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

The EMC® version of Linux®, used as the operating system on the Centera server, is a derivative of Red Hat® 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 Centera server, you agree to the terms and conditions listed therein.

Trademark Information

EMC®, EMC, Symmetrix, Celerra, CLARiiON, CLARAlert, DG, E-Infostructure, HighRoad, Navisphere, PowerPath, ResourcePak, SnapView/IP, SRDF, VisualSAN, where information lives, and The EMC Effect are registered trademarks and EMC Automated Networked Storage, EMC ControlCenter, EMC Developers Program, EMC Enterprise Storage, EMC Enterprise Storage Network, EMC OnCourse, EMC Proven, EMC Snap, Access Logix, AutoAdvice, Automated Resource Manager, AutoSwap, AVALONidm, C-Clip, CacheStorm, Celerra Replicator, Centera, CentraStar, CLAREvent, Connectrix, CopyCross, CopyPoint, CrosStor, Direct Matrix, Direct Matrix Architecture, EDM, E-Lab, Enginuity, FarPoint, FLARE, GeoSpan, InfoMover, MirrorView, NetWin, OnAlert, OpenScale, Powerlink, PowerVolume, RepliCare, SafeLine, SAN Architect, SAN Copy, SAN Manager, SDMS, SnapSure, SnapView, StorageScope, SupportMate, SymmAPI, SymmEnabler, Symmetrix DMX, TimeFinder, Universal Data Tone, and VisualISRM are trademarks of EMC Corporation. All other trademarks used herein are the property of their respective owners.

Sun, the Sun Logo, Solaris, the Solaris logo, the Java compatible logo are trademarks or registered trademarks of Sun Microsystems, Inc.

All SPARC trademarks are trademarks or registered trademarks of SPARC International.

Inc.Linux is a registered trademark of Linus Torvalds.

ReiserFS is a trademark of Hans Reiser and the Naming System.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

Red Hat is a registered trademark of Red Hat Software, Inc.

UNIX is a registered trademark in the United States and other countries and is licensed exclusively through X/Open Company Ltd.

HP-UX is a trademark of Hewlett-Packard Company.

AIX® is a registered trademark of IBM Corporation.

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

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

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

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

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

Finally, nothing in this license shall be interpreted to allow you to fail to fairly credit me, or to remove my credits, without my permission, unless you are an end user not redistributing to others. If you have doubts about how to properly do that, or about what is fair, ask. (Last I spoke with him Richard was contemplating how best to address the fair crediting issue in the next GPL version.)
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>xi</td>
<td>Preface</td>
</tr>
<tr>
<td>1-2</td>
<td>Chapter 1: Product Overview</td>
</tr>
<tr>
<td>1-3</td>
<td>Centera</td>
</tr>
<tr>
<td>1-7</td>
<td>Cluster Topology</td>
</tr>
<tr>
<td>1-7</td>
<td>Centera Interfaces</td>
</tr>
<tr>
<td>1-7</td>
<td>API</td>
</tr>
<tr>
<td>1-7</td>
<td>CLI</td>
</tr>
<tr>
<td>1-7</td>
<td>Centera Viewer</td>
</tr>
<tr>
<td>1-7</td>
<td>CenteraVerify</td>
</tr>
<tr>
<td>1-7</td>
<td>CenteraPing</td>
</tr>
<tr>
<td>1-7</td>
<td>Front Panel</td>
</tr>
<tr>
<td>1-8</td>
<td>Centera Features</td>
</tr>
<tr>
<td>1-9</td>
<td>Content Addressed Storage</td>
</tr>
<tr>
<td>1-11</td>
<td>Content Protection</td>
</tr>
<tr>
<td>1-12</td>
<td>Compliancy</td>
</tr>
<tr>
<td>1-13</td>
<td>Data Retention</td>
</tr>
<tr>
<td>1-14</td>
<td>Regeneration</td>
</tr>
<tr>
<td>1-15</td>
<td>Full Scalability and Serviceability</td>
</tr>
<tr>
<td>1-16</td>
<td>Organic Data Integrity Checking</td>
</tr>
<tr>
<td>1-17</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>1-18</td>
<td>Data Shredding</td>
</tr>
<tr>
<td>1-19</td>
<td>Replication</td>
</tr>
<tr>
<td>1-20</td>
<td>Email Home</td>
</tr>
<tr>
<td>1-22</td>
<td>Centera Compliance Licensing Models</td>
</tr>
<tr>
<td>1-22</td>
<td>Compliance Licensing Models and Retention Periods</td>
</tr>
<tr>
<td>1-23</td>
<td>Compliance Licensing Models and Centera Features</td>
</tr>
</tbody>
</table>
Contents

What’s New? ........................................................................................................... 1-24
  Fast Blob Naming .................................................................................... 1-24
  Monitor API ....................................................................................... 1-24
  New Functions ................................................................................. 1-24

Chapter 2 Setup

System Requirements ...................................................................................... 2-2
  Windows ............................................................................................. 2-2
  Solaris ................................................................................................. 2-2
  Linux .................................................................................................... 2-3
  HP-UX .................................................................................................. 2-3
  AIX ....................................................................................................... 2-4
  IRIX ..................................................................................................... 2-4
CenteraVerify ............................................................................................ 2-6
  Installing CenteraVerify .................................................................. 2-7
  Using CenteraVerify ....................................................................... 2-8
  Configuring CenteraVerify ............................................................... 2-14
CenteraPing ............................................................................................... 2-19
  Running CenteraPing ......................................................................... 2-19

Chapter 3 C API Reference

Function Details ............................................................................................. 3-2
  Function Call ......................................................................................... 3-2
  Parameter List ....................................................................................... 3-2
  API Specific Types ................................................................................. 3-3
Pool Functions ............................................................................................. 3-5
  FPPool_Open/FPPool_OpenW ........................................................... 3-6
  FPPool_GetLastError ......................................................................... 3-10
  FPPool_GetLastErrorInfo ................................................................... 3-11
  FPPool_Close ....................................................................................... 3-12
  FPPool_SetIntOption/FPPool_SetIntOptionW .................................. 3-13
  FPPool_SetGlobalOption/FPPool_SetGlobalOptionW ..................... 3-15
  FPPool_GetPoolInfo ............................................................................ 3-17
  FPPool_GetIntOption/FPPool_GetIntOptionW .................................. 3-19
  FPPool_GetGlobalOption/FPPool_GetGlobalOptionW ..................... 3-20
  FPPool_GetCapability/FPPool_GetCapabilityW ................................ 3-21
  FPPool_GetClusterTime ..................................................................... 3-24
Contents

Clip Functions................................................................................. 3-27
Clip Handling Functions............................................................... 3-27
FPClip_Create/FPClip_CreateW.......................................... 3-28
FPClip_SetName/FPClip_SetNameW ................................ 3-30
FPClip_Open ........................................................................... 3-31
FPClip_Write ........................................................................... 3-33
FPClip_RawRead .................................................................... 3-35
FPClip_RawOpen ................................................................... 3-36
FPClip_SetRetentionPeriod ................................................... 3-38
FPClip_Delete.......................................................................... 3-40
FPClip_Purge........................................................................... 3-42
FPClip_Close ........................................................................... 3-44
FPClip_Exists........................................................................... 3-45
Clip Info Functions ........................................................................ 3-47
FPClip_GetPoolRef ................................................................. 3-48
FPClip_GetNumTags ............................................................. 3-49
FPClip_GetNumBlobs ............................................................ 3-50
FPClip_GetTotalSize............................................................... 3-51
FPClip_GetName/FPClipGetNameW............................... 3-52
FPClip_GetClipID................................................................... 3-53
FPClip_GetCreationDate ....................................................... 3-54
FPClip_GetRetentionPeriod .................................................. 3-55
FPClip_IsModified.................................................................. 3-56
Clip Tag Functions ......................................................................... 3-57
FPClip_GetTopTag ................................................................. 3-58
FPClip_FetchNext ................................................................... 3-59
FPClip_SetDescriptionAttribute/
FPClip_SetDescriptionAttributeW ....................................... 3-61
FPClip_GetDescriptionAttribute/
FPClip_GetDescriptionAttributeW ........................................ 3-63
FPClip_GetNumDescriptionAttributes............................... 3-65
FPClip_GetDescriptionAttributeIndex/
FPClip_GetDescriptionAttributeIndexW............................ 3-66
FPClip_RemoveDescriptionAttribute/
FPClip_RemoveDescriptionAttributeW .................................. 3-68
Tag Functions............................................................................. 3-69
Tag Handling Functions............................................................. 3-69
FPTag_Create/FPTag_CreateW........................................... 3-70
FPTag_Copy ............................................................................ 3-71
FPTag_GetPoolRef ................................................................. 3-73
FPTag_GetClipRef ................................................................. 3-74
FPTag_GetTagName/FPTag_GetTagNameW........................... 3-75
FPTag_GetBlobSize................................................................. 3-77
Contents

FPTag_Close ..............................................................3-78
FPTag_Delete ............................................................3-79
Tag Navigation Functions .............................................3-81
  FPTag_GetFirstChild ...............................................3-83
  FPTag_GetSibling ..................................................3-84
  FPTag_GetPrevSibling ............................................3-85
  FPTag_GetParent ....................................................3-86
Tag Attribute Functions ...............................................3-87
  FPTag_SetStringAttribute/FPTag_SetStringAttributeW ...3-88
  FPTag_SetLongAttribute/FPTag_SetLongAttributeW ......3-90
  FPTag_SetBoolAttribute/FPTag_SetBoolAttributeW ......3-92
  FPTag_GetStringAttribute/FPTag_GetStringAttributeW 3-94
  FPTag_GetLongAttribute/FPTag_GetLongAttributeW ...3-96
  FPTag_GetBoolAttribute/FPTag_GetBoolAttributeW ......3-97
  FPTag_RemoveAttribute/FPTag_RemoveAttributeW ......3-98
  FPTag_GetNumAttributes ........................................3-99
  FPTag_GetIndexAttribute/FPTag_GetIndexAttributeW3-100
Blob Handling Functions ...............................................3-103
  FPTag_BlobWrite ..................................................3-104
  FPTag_BlobRead ..................................................3-108
  FPTag_BlobReadPartial .......................................3-111
  FPTag_BlobPurge ..................................................3-114
  FPTag_BlobExists ...............................................3-116
Stream Functions ........................................................3-118
  Stackable Stream Support .......................................3-118
Generic Stream Operation .............................................3-118
Stream Creation Functions ............................................3-121
  FPStream_CreateGenericStream ................................3-122
  FPStream_CreateFileForInput ...................................3-134
  FPStream_CreateFileForOutput ..................................3-136
  FPStream_CreateWithFile .......................................3-138
  FPStream_CreateBufferForInput ................................3-139
  FPStream_CreateBufferForOutput ...............................3-140
  FPStream_CreateWithBuffer ....................................3-141
  FPStream_CreateToStdio .......................................3-142
  FPStream_CreateToNull ........................................3-143
  FPStream_CreateTemporaryFile .................................3-144
Stream Handling Functions ............................................3-145
  FPStream_Read ..................................................3-146
  FPStream_Write ..................................................3-146
  FPStream_SetMarker ............................................3-146
  FPStream_GetMarker ............................................3-146
  FPStream_PrepareBuffer .......................................3-147
Contents

FPStream_Complete ............................................................. 3-148
FPStream_SetMark ............................................................... 3-149
FPStream_ResetMark ........................................................... 3-150
FPStream_GetInfo................................................................. 3-151
FPStream_Close..................................................................... 3-152
Query Functions........................................................................... 3-153
  FPQuery_Open/FPQuery_OpenW.................................... 3-154
  FPQuery_GetPoolRef ........................................................... 3-157
  FPQuery_FetchResult........................................................... 3-158
  FPQuery_Close...................................................................... 3-160
Monitoring Functions.................................................................. 3-161
  FPMonitor_Open/FPMonitor_OpenW ............................. 3-162
  FPMonitor_Close .................................................................. 3-163
  FPMonitor_GetDiscovery .................................................... 3-164
  FPMonitor_GetDiscoveryStream........................................ 3-166
  FPMonitor_GetAllStatistics ................................................. 3-168
  FPMonitor_GetAllStatisticsStream..................................... 3-170
  FPEventCallback_RegisterForAllEvents ........................... 3-172
  FPEventCallback_Close ....................................................... 3-174
Error Codes ................................................................................... 3-175

Chapter 4 Java API Reference

Classes................................................................................................ 4-2
  Constants.................................................................................... 4-2
  FPPool......................................................................................... 4-3
  FPPool.PoolInfo......................................................................... 4-4
  loadFPLibrary............................................................................ 4-5
  FPClip ......................................................................................... 4-6
  FPTag.......................................................................................... 4-8
  FPStream .................................................................................. 4-11
  FPQuery...................................................................................... 4-12
Errors................................................................................................ 4-13

Chapter 5 Toolbox

Error Handling .................................................................................. 5-2
C Toolbox Functions......................................................................... 5-4
  ClipFile_Store ............................................................................ 5-5
  ClipFile_Retrieve....................................................................... 5-6
  FPError_GetErrorString........................................................... 5-7
  FPError_GetErrorMessage....................................................... 5-8
Appendix A  Monitoring Information

Discovery Information .................................................. A-2
  Syntax Discovery ...................................................... A-2
  Sample Discovery .................................................... A-5
Statistical Information ................................................. A-10
  Syntax Statistics ..................................................... A-10
  Sample Statistics ................................................... A-10
Alert Information ........................................................ A-12
  Syntax Alert ........................................................... A-12
  Sample Alert .......................................................... A-12

Glossary ........................................................................ g-1

Index ............................................................................ i-1
This guide forms part of the EMC Centera™ Software Development Kit (SDK), P/N CNR-SDK, and is for experienced programmers who are developing applications that interface with a Centera cluster. It is intended to be a complete reference guide for both C and Java application development using the Centera API.

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

- Chapter 1 contains an overview of the product and a What’s New section.
- Chapter 2 lists the system requirements and describes two tools that check the connectivity to the cluster.
- Chapter 3 is a reference guide to the C API.
- Chapter 4 is a reference guide to the Java API.
- Chapter 5 describes the toolbox API.

Other Centera publications include:

- Centera Product Guide, P/N069001221
- Centera Programmer’s Guide, P/N069001127
- Centera System Operator’s Guide, P/N069001126
- Centera Replication Guide, P/N069001168
- Cabinet Setup Guide for the 40U Cabinet, P/N014003099
- Site Preparation and Unpacking Guide for the 40U Cabinet, P/N014003100

EMC uses the following conventions for notes, cautions, warnings, and danger notices.
A note presents information that is important, but not hazard-related.

**CAUTION**

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

**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 warning.

EMC uses the following type style conventions in this guide:

- **Boldface**
  - Specific filenames or complete paths.
  - Dialog box names and menu items in text.
  - Selections you can make from the user interface, including buttons, icons, options, and field names.
  - Emphasis in cautions and warnings.

- **Italic**
  - New terms or unique word usage in text.
  - Command line arguments when used in text.

- **Fixed space**
  - Examples of specific command entries that you would type, displayed text, or program listings.
  - For example:
    
    QUERY [CUU=cuu|VOLSER=volser]

- **Fixed italic**
  - Arguments used in examples of command line syntax.
Where to Get Help
For questions about technical support and service, contact your service provider.
If you have a valid EMC service contract, contact the EMC Customer Support Center at:

United States: (800) 782-4362 (SVC-4EMC)
Canada: (800) 543-4782 (543-4SVC)
Worldwide: +1 - (508) 497-7901

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

Sales and Customer Service Contacts
For the list of EMC sales locations, please access the EMC home page at:

http://www.emc.com/contact/

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

http://powerlink.emc.com

Your Comments
Your suggestions will help us continue to improve the accuracy, organization, and overall quality of the user publications. Please send a message to techpub_comments@emc.com with your opinions of this guide.
This chapter introduces EMC Centera™, the world’s first Content Addressed Storage solution designed to store and provide access to fixed content.

The main sections are:

- Centera................................................................................................. 1-2
- Cluster Topology................................................................................ 1-3
- Centera Interfaces .............................................................................. 1-7
- Centera Features................................................................................. 1-8
- Centera Compliance Licensing Models ........................................ 1-22
- What’s New?..................................................................................... 1-24
Centera

Centera is a new, networked storage system specifically designed to store and provide fast, easy access to fixed content (information in its final form). It is the first solution to offer online availability with long-term retention and assured integrity for this fastest-growing category of information.

Centera greatly simplifies the task of managing, sharing, and protecting large-scale fixed content repositories. Centera enables businesses to use this traditionally offline information to cost-effectively support new sources of revenue generation, expanded business models, and increase service levels to users and customers.

Several applications are available as a gateway to a Centera and a fully comprehensive Software Developers Kit (SDK) is provided for application developers to integrate via Centera’s Access Application Program Interface (API).
Cluster Topology

This section describes Centera’s basic hardware elements that are grouped into structures that we call cabinets or clusters. A cabinet is a single Centera unit containing 8, 16, 24, or 32 nodes. A cluster consists of multiple Centera cabinets interconnected to present a single storage area. We currently support up to 4 cabinets in a cluster. In this guide we use the term cluster to refer to both single and multiple cabinets.

Note: All Centera clusters should be installed by qualified EMC personnel only. Attempts by unqualified personnel to set up or power up a Centera may void product warranties. Do NOT change any configuration or cabling or your Centera may not function or be damaged.

Figure 1-2  Centera Topology Overview
A single cabinet consists of the following hardware elements:

- One 40U NEMA rack
- 8, 16, 24, or 32 nodes of which:
  - Each node is connected to both cube switches.
  - 2, 4 or more nodes are connected to the client LAN infrastructure (Access Nodes).
  
  Access Nodes communicate with the customer’s environment and with the cluster’s Storage Nodes. Each Access Node has an external IP address. Access Nodes are essentially the gateway to the cluster for customer’s data.
- The remaining nodes store data (Storage Nodes).
- The ETH2 ports can be used for manageability connections by EMC service personnel only.

### Table 1-1 Node Versions

<table>
<thead>
<tr>
<th>Access Connectivity</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard Drive</td>
<td>160 GB</td>
<td>250 GB</td>
<td>320 GB</td>
</tr>
<tr>
<td>Processor</td>
<td>866 MHz P3</td>
<td>1 GHz P3</td>
<td>2 GHz P4</td>
</tr>
</tbody>
</table>

- Two internal LAN-48 port cube switches that connect to all of the nodes to enable the CentraStar network and provide full redundancy in case one fails.
- Two internal root switches for multi-cabinet clusters (these are optional and not shown in Figure 1-2.)
- Two power distribution units (PDUs).
- One Automatic (AC) Transfer Switch (ATS) to provide power failover so that the cluster continues to operate in case one of the PDUs fails.
- An optional external modem (though two are recommended to maintain redundancy) to support the Email Home feature.
Figure 1-3  Modems Connected to a Node

- Associated cabling and power connections.

**Note**: Do not touch the cabling within the Centera cabinet.
Figure 1-4   A 32-node Centera cabinet with Cube Switch
Centera Interfaces

End users can access a Centera in different ways: via the API, the CLI, Centera Viewer, CenteraVerify, CenteraPing, and the front panel. The sections below describe the Centera interfaces.

API

End users input their data to content management applications that interface with Centera via the Access Application Program Interface (API). The API connects to the Access Nodes using IP addressing; the Storage Nodes process and store the end user data. This guide contains a detailed overview of the Access API.

CLI

The system operator can administer the cluster and monitor its performance using the Centera Command Line Interface (CLI), a set of predefined commands that you enter via a command line. Refer to the Centera System Operator’s Guide, P/N 069001126, for more information on the use of the CLI and specific CLI commands.

Centera Viewer

The system operator can view and save information about the run-time behavior of a Centera using Centera Viewer. Refer to Centera Viewer Online Help for detailed information on how to use this application.

CenteraVerify

CenteraVerify quickly checks the connectivity to a Centera and provides some cluster details. Refer to the Centera System Operator’s Guide, P/N 069001126, for more information on CenteraVerify.

CenteraPing

CenteraPing quickly checks the connectivity to a Centera. Refer to the Centera System Operator’s Guide, P/N 069001126, for more information on CenteraPing.

Front Panel

Every node in a Centera has its own front panel that displays the node status. EMC service personnel use the front panel for node administration.

CAUTION

Under no circumstances should end users administer a node via the front panel. This can cause severe damage to the system.
Centera Features

The following sections describe the Centera features:

- Content Addressed Storage
- Content Protection
- Compliancy
- Data Retention
- Regeneration
- Full Scalability and Serviceability
- Organic Data Integrity Checking
- Garbage Collection
- Data Shredding
- Replication
- Email Home
Content Addressed Storage

Content Addressed Storage (CAS) is a category of automated networked storage established to store large volumes of fixed content. Each data object gets a unique identifier, or Content Address (CA), derived from its digital content. CAS eliminates the need for applications to understand and manage the physical location of information on storage media.

Referencing data based on its actual content presents an additional storage benefit. If multiple clients store the same data object on a cluster, the system needs to store only one copy; each client, however, will receive a unique identifier to retrieve it.

In addition, CAS ensures that stored data cannot be changed or tampered with because the identifier is calculated from the binary content of the data. If the content of the data changes, the identifier changes also.

The sections below describe the basic concepts of the CAS technology.

C-Clip

A C-Clip is a package containing the user’s data and associated metadata. When a user presents a file or a set of files to a Centera server, the system calculates a unique Content Address (CA) for the data and then stores this address in a newly created XML file, the C-Clip Descriptor File (CDF) together with application-specific metadata.

The system then calculates another CA for the CDF, and stores the CDF plus the user’s file in the complete C-Clip package on the cluster. The CA of the CDF is a handle for the C-Clip that the system uses to retrieve the user’s data as shown in Figure 1-5.

![The Creation of a C-Clip](image)
**Tags**

A C-Clip contains both user data and associated metadata in XML tags. Each tag has a name and can have one or more attributes. The tags representing actual data are called *blob tags*.

**Blob**

We use the term *blob* to mean the Distinct Bit Sequence (DBS) of the user's data. Do NOT confuse it with the term Binary Large Object. Every file consists of a unique sequence of bits and bytes. The DBS represents the actual content of a file and is independent of the filename and physical location. A blob is represented in the C-Clip by a blob tag.
Content Protection

Data redundancy is an intrinsic feature of Centera that ensures continuous data availability. Centera offers two protection schemes to store data redundantly:

- Content Protection Mirrored (CPM)
- Content Protection Parity (CPP)

Each cabinet contains two groups of nodes that are powered from separate power rails plugged into the Automatic (AC) Transfer Switch (ATS). To assure continuous data availability and allow self-healing, Centera mirrors (CPM) or fragments (CPP) all data on the cluster. Data copies or fragments are stored on different nodes thus ensuring data redundancy in the event of a disk or node failure. If a power rail fails, all nodes in the cabinet will continue to function due to the ATS.

Centera is a high-availability system, designed for redundancy across its hardware architecture. Consistent with this architecture, Access Nodes and Storage Nodes are added in pairs to ensure that if a single node fails, there is another node remaining to provide fail-over. The paired node architecture is especially important in Centera clusters running CPM.

The size of the data object determines which protection scheme the server uses to store the data.

**Content Protection Mirrored (CPM)**

CPM stores a complete copy of the data object on a mirror node. CPM enhances faster data regeneration and improves performance during normal operation. Object retrieval is improved because the Access Node can select the least-loaded node from which to retrieve a specific object.

**Content Protection Parity (CPP)**

CPP segments a data object into 6 parts and stores each one on a different drive of a different node in the same cluster. It calculates a parity fragment from the stored fragments and stores that as the 7th data segment on yet another node. This mechanism is a derivative of RAID-3 (mostly used in single-user systems with large record applications).
Compliancy

A Centera does more than simply store data. It helps to manage the retention of information and guarantees the authenticity of the information.

The new Centera Compliance and Compliance Plus Editions address regulatory needs and corporate governance. For more information on compliancy, retention periods, and other Centera features refer to Centera Compliance Licensing Models on page 1-22.

Note: A Centera requires additional application software to meet all requirements for a compliant environment, and human processes that review retention requirements and assess special situations (such as audits, litigation, and others).
Data Retention

The retention period is the time that a data object has to be stored before an application is allowed to delete it. Each data object stored on a Centera has a retention period. This can either be the default value set by the server (depending on the Centera licensing model) or the value that has been given to the data object by the SDK before it was stored.

Once data has been stored on a Centera, its retention period cannot be changed. When the retention period of the stored data expires, applications will be allowed to delete the data. The data will not be deleted automatically, it is the responsibility of the application or end user to delete it.

For more information on the Centera compliance licensing models and retention periods, refer to Centera Compliance Licensing Models on page 1-22. This guide and the Centera Programmer’s Guide, P/N 069001127, contain more information on how to set the retention periods using the SDK.
Regeneration prevents data loss by disk and node failures, and provides self-healing functionality. It relies on the existence of mirrored copies (Content Protection Mirrored) or fragmented segments (Content Protection Parity) of the data on different nodes in the cluster.

The system mirrors or segments all data in the cluster to ensure that redundant copies are available. If a disk or node failure occurs and the data objects stored on the failed disk or node are no longer available, the cluster will automatically start the regeneration process, which creates a copy of each of the affected data objects. All nodes in the cluster participate in this process.

There are several levels of regeneration and multiple mechanisms to trigger regeneration tasks. Disk regeneration, for instance, occurs when a node detects a disk failure. The node informs the other nodes in the cluster. This triggers a regeneration task on every node that has a copy of the objects stored on the failed disk.

The system periodically inspects the health status of all nodes in a cluster. When the system cannot reach a node, it triggers a regeneration task for the missing data.

**Note:** The EMC Customer Support Center receives an email notification of any hardware faults, refer to Organic Data Integrity Checking on page 1-16 for more information.
**Full Scalability and Serviceability**

Centera is a *no single point of failure* system. It is highly scalable and non-disruptively serviceable over the course of its life. Centera’s data management technology allows dynamic expansion when more storage is required and self-healing capability if hardware failures occur.

The cluster automatically detects new nodes when they are connected and powered on. The system excludes from the cluster any drives that fail and regenerates their data to ensure that a fully redundant copy of the data is always available. No system outage or restore is required.
Organic Data Integrity Checking

Centera features complete redundancy to protect against component faults. In the event of a disk drive failure, Centera will self-heal by detecting the fault and generating new copies of the original data objects.

A permanent integrity-checking task runs in the background to confirm the state of the data. This task continuously recalculates the Content Addresses for all objects and compares them to the original calculations. It also verifies that mirror copies or segmented fragments exist for every data object.
Garbage Collection

Garbage collection is part of the Centera deletion process. Garbage collection continually runs as a background process and ensures that data deletion will always be successful in the event of a power failure and that storage space is continuously optimized. Garbage collection is disabled by default.

Note: Data shredding must be enabled in order for garbage collection to function and vice versa.
**Data Shredding**

The Centera data shredding process removes all traces of data that has been deleted by an application from a Centera. Data shredding essentially takes disk/file system data blocks that previously held data objects, and overwrites them a number of times with random and/or fixed pattern data to thoroughly remove the media’s magnetic memory of the previously stored data.

Data shredding is disabled by default on a Compliance Edition model and enabled by default on a Compliance Edition Plus model.

*Note:* Garbage collection must be enabled in order for data shredding to function and vice versa.
Replication

A Centera can transparently replicate all stored objects asynchronously to a remote cluster to support disaster recovery. Replication complements Content Protection Mirrored and Content Protection Parity by putting copies of data in geographically separated sites. If a problem renders an entire cluster inoperable, the replica cluster can support the application server until the problem is fixed.

Refer to the Centera Product Guide, P/N 069001221, for more information.
Email Home

Email Home allows Centera to communicate with the EMC Customer Support Center via email. Email Home sends email messages to the EMC Customer Support Center via the customer SMTP infrastructure (VPN) or via a customer workstation with EMC OnAlert™ installed on it.

EMC OnAlert is an application that provides remote support functionality to networked EMC devices and includes Automatic Error Reporting and Remote Technical Support. Refer to the Centera System Operator’s Guide, P/N 069001126, for more information on configuring SMTP and to the EMC OnAlert Product Guide, P/N 300-999-378, for more information on EMC OnAlert and how to install it. If an OnAlert station is used, it must run an SMTP server to accept the health reports.

An on-site EMC engineer can enable or disable Email Home. Table 1-2 shows the three Email Home settings.

<table>
<thead>
<tr>
<th>Email Home Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>None</strong></td>
</tr>
<tr>
<td><strong>Alert</strong></td>
</tr>
<tr>
<td><strong>All</strong></td>
</tr>
</tbody>
</table>

Table 1-3 shows the messages that Email Home can send.

<table>
<thead>
<tr>
<th>Email Home Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Daily health report</strong></td>
</tr>
<tr>
<td><strong>On-demand email to the EMC Customer Support Center</strong></td>
</tr>
<tr>
<td><strong>Email alert</strong></td>
</tr>
</tbody>
</table>
Upon receipt of the email message, the EMC Support Center decides if it is necessary to send a Customer Support Engineer to the customer’s site. If EMC determines that the cluster can be accessed remotely, a support engineer dials into the Centera Alert Station through a modem connection. Once logged in, the engineer uses Centera related tools to analyze the error and implement recovery procedures.

Note: On a Compliance Edition Plus model all remote serviceability procedures are replaced with local support procedures. No modems can be connected to the cluster. The only modem is the one connected to the OnAlert workstation. All dial-in capabilities have been removed. All service calls are on-site visits.
Centera Compliance Licensing Models

There are two Centera compliance licensing models:

- **Compliance Edition**
  
  This model enforces retention periods for stored data and is the option typically required in pharmaceutical, healthcare, and government compliance environments.

- **Compliance Edition Plus**
  
  This model also enforces retention periods but disables remote management and data purging. It also prevents the deletion of any data with unspecified retention periods and meets the requirements of the Securities and Exchange Commission’s (SEC) rule 17a-4 and the Department of Defense’s rule 5015.

Compliance Licensing Models and Retention Periods

The Centera compliance licensing model determines the default value of the retention period that will be used by the server when data is stored without a retention period specified by the SDK.

- A Compliance Edition model sets the default retention period to 0 (no retention period). Applications can specify a different retention period for each data object that they store. The server will only allow the application to delete the data object if the retention period for that object has expired.

- A Compliance Edition Plus model sets the default retention period to -1 (infinite retention). Applications can specify a different retention period for each data object that they store. The server will only allow the application to delete the data object if the retention period for that object has expired. Purging of data is never allowed.

For more information on the difference between delete and purge, refer to the Centera Programmer’s Guide, P/N 069001127.

It is not possible to change a Compliance Edition Plus Centera into a Compliance Edition Centera.

**Note:** The clock settings on the Centera server are factory default and cannot be reset.
Table 1-4 shows how the Centera compliance licensing models affect various Centera features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Compliance Edition</th>
<th>Compliance Edition Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enforced retention periods</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Default retention period</td>
<td>0</td>
<td>-1 (infinite)</td>
</tr>
<tr>
<td>Purge data</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Delete data</td>
<td>Only if retention period has expired</td>
<td>Only if retention period has expired</td>
</tr>
<tr>
<td>Remote management</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Data shredding (default setting)</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>SNMP access</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Unlock nodes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
What's New?

New features in the current version of the SDK include:

Fast Blob Naming

To improve performance Centera version 2.1 provides a new naming scheme to calculate Content Addresses for small files: fast blob naming. If fast blob naming is enabled, all files smaller than the defined threshold will have a unique Content Address that is similar to the CA that will be calculated when Collision Avoidance has been enabled. Refer to `FCClip_Write` on page 3-33 for an example.

Fast blob naming significantly improves the speed of write operations at the cost of single instance storage, identical content may be stored multiple times.

Only qualified Centera service personnel can enable fast blob naming and the maximum file size (threshold) for which the scheme is used. Fast blob naming is disabled by default. The threshold is 250 KB by default and can be changed when enabling fast blob naming.

The system operator can view if fast blob naming is enabled by using the CLI command `show feature`. This feature also shows the threshold. Refer to the Centera System Operator’s Guide, P/N 069001126, for more information.

Monitor API

SDK version 2.1 and higher offers the Monitor API (MoPI). With this API you can gather monitoring information from the Centera server. The MoPI can be used on both a Compliance Edition and Compliance Edition Plus Model.

Refer to Chapter 3, C API Reference, for more information on the monitoring functions.

New Functions

The following functions and options have been added to the API. Refer to Chapter 3, C API Reference for a detailed function description. Refer to Chapter 4, Java API Reference for their Java mappings.
- PPClip_SetDescriptionAttribute/PPClip_SetDescriptionAttributeW
- PPClip_GetDescriptionAttribute/PPClip_GetDescriptionAttributeW
- PPClip_RemoveDescriptionAttribute/PPClip_RemoveDescriptionAttributeW
- PPClip_GetNumDescriptionAttributes
- PPClip_GetDescriptionAttributeIndex/PPClip_GetDescriptionAttributeIndexW
- PPClip_SetName/PPClip_SetNameW
- FPTag_GetPrevSibling
- FPTag_BlobExists
- FPMonitor_Open
- FPMonitor_Close
- FPMonitor_GetDiscovery
- FPMonitor_GetDiscoveryStream
- FPMonitor_GetAllStatistics
- FPMonitor_GetAllStatisticsStream
- FPEventCallback_RegisterForAllEvents
- FPEventCallback_Close
- PP_OPTION_CLUSTER_NON_AVAIL_TIME (FPPool_SetGlobalOption)
- PP_OPTION_PREFETCH_SIZE (FPPool_SetIntOption)
This chapter describes the system requirements for the installation of the Centera software on a client or application server and the steps needed to connect to a cluster.

The main sections are:

- System Requirements ................................................................. 2-2
- CenteraVerify .................................................................................. 2-6
- CenteraPing .......................................................... 2-19
System Requirements

The minimum system requirements for a client or application server are detailed below. Note that requirements may change depending on the nature of the system’s applications. Refer to the release notes for installation information and latest updates to the product.

Windows

The minimum Windows® system requirements are:

- Microsoft Windows NT® 4.0, SP6
- Microsoft Windows® 2000, SP2
- Microsoft Windows® XP Professional, SP1
- Intel Pentium® III, 550 MHz
- 512 Mbyte RAM
- 100 Mbps Ethernet network card

The supported Windows® compilers are:

- Metrowerks® CodeWarrior® 4.1
- Microsoft Visual Studio® 6.0
- Sun® JDK 1.3.1
- GNU gcc compiler, version 3.2

Solaris

The minimum Solaris™ system requirements are:

- Sun® Solaris™ 8 and 9/Solaris 2.6
- Sun® UltraSPARC™ IIe
- 512 Mbyte RAM
- 100 Mbps Ethernet network card
The supported Solaris™ compilers are:

- For C/C++:
  - Solaris 8 and 9: Sun® Forte™ C/C++, Enterprise Edition 6, update 2
  - Solaris 2.6: GNU gcc 2.95.3

- For 32-bit Java:
  - Sun® JDK 1.3.1

Below is an example of a 32-bit compilation using the Sun® Forte™ suite (CC in this case) that supports multi-threading:

```
CC -mt -DPOSIX -I/opt/Centera_SDK/include
-L/opt/Centera_SDK/lib/32 -o test_api test_api.c
-lFPLibrary
```

---

**Linux**

The minimum Linux® system requirements are:

- Linux® Red Hat® 7.2
- Intel Pentium® III, 550 MHz
- 512 Mbyte RAM
- 100 Mbps Ethernet network card

The supported Linux® compilers are:

- For C/C++:
  - GNU gcc compiler, version 3.0.4

- For 32-bit Java:
  - SUN® JDK 1.3.1

---

**HP-UX**

The minimum HP-UX™ system requirements are:

- HP-UX 11i: 32 bit in portability mode; 64 bit in non-portability mode
- HP-UX 11.0: 32 bit in portability mode; 64 bit in non-portability mode
- HP-UX11i: HP 9000/B2600 workstation, 500 MHz
Setup

- HP-UX 11.0: HPC 3000 workstation
- 512 Mbyte RAM
- 100 Mbps Ethernet network card

The supported HP® 11i compilers are:
- For C/C++
  - HP aC++, version A.03.33
- For 32-bit Java:
  - Sun® JDK 1.3.1

AIX

The minimum AIX® system requirements are:
- IBM® AIX® version 5.2
- IBM® AIX® 5L version 5.1
- IBM® AIX® version 4.3.3
- PowerPC - Power 3,333 MHz
- 512 Mbyte RAM
- 100 Mbps Ethernet network card

The supported AIX® compilers are:
- For C/C++:
  - IBM® VisualAge® C++/ C version 6.0 for AIX
- For 32-bit Java:
  - IBM® Developer’s kit for Java version 1.3.1

IRIX

The minimum IRIX® system requirements are:
- SGI® IRIX® 6.5.17m
- SGI® IP35 500 Mhz
- 512 Mbyte RAM
- 100 Mbps Ethernet network card
The supported IRIX® compilers are:

- For C/C++:
  SGI® MIPSpro 7.3.1.2m compiler
- For 32-bit Java:
- Java™ 2 v 1.3.1 APIs for SGI® IRIX®
CenteraVerify

CenteraVerify is used to efficiently determine the general health of a Centera cluster. CenteraVerify tests the connection to a Centera cluster, displays cluster information, displays information about the capabilities, and writes to, retrieves from and deletes files on a cluster.

**Note:** Ensure that the read, write, delete, and clip-configuration capabilities are enabled in order to use all functionalities of CenteraVerify.

More detailed health information or analysis of a Centera cluster is available through the use of Centera Viewer but CenteraVerify can also be used when Centera Viewer is not available.

The following sections describe how to install CenteraVerify and how to use and configure it on a Windows and a Unix platform.

**Note:** All output in this chapter is sample output and may differ from the actual output.
Installing CenteraVerify

CenteraVerify should have been installed with your cluster by the EMC Customer Engineer.

However, you may require a second installation. Insert the SDK CD-ROM in the CD-ROM drive. Follow the onscreen instructions to install CenteraVerify.

![CenteraVerify User Interface](image)

To check which version of CenteraVerify is running, click on the top left corner of the CenteraVerify screen and select **About CenteraVerify**....
Setup

Figure 2-2  About CenteraVerify

Using CenteraVerify

CenteraVerify performs the following steps:

Step 1:
- Connects to Access Nodes listed in the List IP addresses field.

Step 2:
- Displays the cluster ID, the available capacity, the CentraStar™ software version, and the replica address of the Centera cluster.
- Displays information about the write, read, delete and purge capabilities.

Step 3:
- Writes files to, and reads them from the cluster.
- Deletes files from the cluster.

Please note the IP addresses of Access Nodes in the cluster. You will need to refer to these addresses in any application that interfaces with the cluster.

The default port number for the Centera cluster is 3218. Ensure that this port is open for import and export through your firewall (contact your network administrator for help).

Note: Response time varies according to the LAN speed. Verify the IP address and network connectivity with the network administrator before contacting the EMC Customer Support Center.
CenteraVerify Step 1

- Click `Start/Programs/Centera Tools 2.1/CenteraVerify 2.1` to launch CenteraVerify.

- Enter the IP address of the Access Node(s) to be tested in the `List IP addresses` box. The maximum number of IP addresses that you can enter is 16.

- Check the `Authentication` box to enter the authentication string in the `Authentication` field. If you use the default EMC PAI module, this string is the full path to the Pool Entry Authorization (PEA) file. If you use your own custom PAI module, complete required fields. If you do not specify an authentication string, CenteraVerify connects to the cluster using the Anonymous profile.

- Click `Run`.

The `Status` box displays `Success` if a connection is made to the Access Node and `Failure` if not.
Click **Stop** to abort the current tests.

Click **Clear Results** to clear all fields on the screen (except the list of Access Node IP addresses).

Click **Update Ini** to save the list of Access Nodes to the INI file. Refer to Configuring CenteraVerify on page 2-14 for more information on the INI file.

### CenteraVerify Step 2

The **Health Report** section shows the cluster ID, the available capacity, the CentraStar™ software version, the capabilities associated with the current connection to the cluster and the replica address (if applicable) of the Centera cluster. CenteraVerify displays the data from the first Access Node to respond to Step 1.

**Note:** Some fields are left blank, depending on the configuration of your cluster.

![Figure 2-5 CenteraVerify Step 2 Health Report](image)

- The available capacity is shown as free usable capacity and total capacity.

  **Note:** 1 Gbyte = \(10^9\) bytes.

- Step 3’s **Write Read Test** is not performed if the **Write** capability is disabled.

- The following message is displayed if the **Read** capability is disabled: Unable to read test files with Read capability disabled. Proceed with test? Yes/No.
Click Yes to proceed with the Write Read Test, however the files will not be read.

Click No and the Write Read Test is not performed.

The following message is displayed if the Delete capability is disabled:

Unable to delete test files with Delete capability disabled. Proceed with this test. Yes/No.

Click Yes to proceed with the Write Read Test, however the C-Clips and blobs will not be removed from the server.

Click No and the Write Read Test is not performed.

**CenteraVerify Step 3**

Step 3’s Write Read Test shows the status of the Write, Read and Delete operations. The Success or Failure status is displayed in the Status field next to the corresponding Write, Read or Delete row. CenteraVerify selects the first Access Node to respond to Step 1 and uses that IP address to write to, read from, and delete from the cluster.

**Figure 2-6 CenteraVerify Step 3 Write Read Test**

CenteraVerify calculates the throughput rate of the Write and Read operations by adding up each Write and Read time and dividing that sum by the total bytes transferred. CenteraVerify specifies the throughput of the Write and Read operations in Mbytes per second. The throughput is displayed in the ThroughPut field next to the corresponding Write, Read or Delete row. All displayed throughput rates are average values.

*Note:* CenteraVerify is not intended as a performance measurement tool and all data returned in the ThroughPut fields is purely for informational purposes.
The **Status** field displays the current status. This enables you to monitor the progress of the test. The possible status values are:

- Access Node validated
- Retrieving Pool Information
- Writing File
- Reading File
- Deleting file
- File Deleted
- Finished

![Figure 2-7 CenteraVerify Status](image)

Once the test is completed, a **Test Completion** message box appears. This **Test Completion** message box also displays the log file path created during the process.

![Figure 2-8 CenteraVerify Completion Message Box](image)

- Click **Exit** to quit CenteraVerify.

**CenteraVerify Log File**

The CenteraVerify log file contains the generation date, a list of all the runtime parameters, the results of the tests, the errors (if applicable), and the total throughput of the **Write** and **Read** operations.
Figure 2-9  CenteraVerify Log File
Launching CenteraVerify automatically overwrites any existing log files. If you perform more than one test without exiting CenteraVerify, the log file displays the results of each test.

It is possible to view more detailed error descriptions than those displayed in the log file of CenteraVerify. Create an empty FPLibrary.log in the same directory as the CenteraVerify.exe and the FPLibrary before launching CenteraVerify. This FPLibrary.log will now record detailed error descriptions.

**Note:** CenteraVerify may give additional feedback on an operation. A dialog box appears if, for example, the INI file cannot be found or if you exceed the maximum number of Access Nodes.

### Configuring CenteraVerify

Click **Start**, **Programs**, **Centera Tools 2.1**, **CenteraVerify 2.1** to launch CenteraVerify. CenteraVerify automatically creates default runtime values, a default INI file, and then displays the CenteraVerify screen (refer to Figure 2-1).

You can also customize CenteraVerify to your own needs by modifying the default INI file with values of your choice, whereby -1 denotes the default SDK value. The INI file configures the runtime options. Restart CenteraVerify for changes to take effect.

By default, **CenteraVerify.ini** is located in the same folder as **CenteraVerify.exe**. The INI file is opened and read every time CenteraVerify starts.

If CenteraVerify cannot find the INI file, it uses the default values of each option, creates a new INI file and saves it as **CenteraVerify.ini** in your default folder.
There are two sections in the default INI file: [Dialog] and [FileSizeSec]. The file sizes (in Kbytes) are placed in the [FileSizeSec] section and the rest are placed in the [Dialog] section.

The number of entries for FileSizeN should match the value for NumFiles.

Example: If NumFiles=6, then there should be entries for FileSize1 … FileSize6.

There is a third optional section for the INI file: [AccessNodes]. This is a list of Access Nodes that are to be used in the List IP addresses box (refer to Figure 2-3). The Key values must be listed as AccessNode1, AccessNode2, … AccessNodeN.

The Update Ini button (refer to Figure 2-3) automatically creates the [AccessNodes] section and adds all Access Nodes listed on the CenteraVerify screen.

<table>
<thead>
<tr>
<th>Configurable Runtime Options</th>
<th>Description</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumFiles</td>
<td>The number of files to create, write and read.</td>
<td>3</td>
</tr>
<tr>
<td>restoreFolder</td>
<td>Folder used to place restored files.</td>
<td>.\Restore</td>
</tr>
<tr>
<td>LogFile</td>
<td>File name and path of the log file.</td>
<td>.\CenteraVerify.log</td>
</tr>
<tr>
<td>GenFileFolder</td>
<td>Folder containing temporary data files.</td>
<td>.</td>
</tr>
<tr>
<td>Retries</td>
<td>The number of retries in response to API errors.</td>
<td>6</td>
</tr>
<tr>
<td>RetrySleepInterval</td>
<td>Sleep duration between API errors.</td>
<td>2000 ms</td>
</tr>
<tr>
<td>FileSizesSec</td>
<td>Size in Kbytes of each of the files.</td>
<td>10, 100, 1000 (values in Kbytes)</td>
</tr>
<tr>
<td>AccessNodes</td>
<td>List of the Access Node IP addresses.</td>
<td>This cannot be a default value.</td>
</tr>
</tbody>
</table>
Setup

Figure 2-10 CenteraVerify.ini

If you have entered Access Nodes in your INI file, these are automatically displayed in the **List IP addresses** box of CenteraVerify (refer to Figure 2-3). You can override the values in the INI file by modifying, deleting, or adding IP addresses in the **List IP addresses** box.

---

**CenteraVerify for Unix**

The Unix version of CenteraVerify runs in command line mode.

1. The syntax of CenteraVerify is:

   ```
   usage: centeraverify -address node1,node2,... [-log logfilename] [-fileSize(kb) filesize1,filesize2,...] [-store directory] [-restore directory] [-retries numRetries] [-retrySleep retrySleep]
   ```

2. Arguments:
   - `-address`:
     A comma delimited list of Access Node names or IP addresses for the Centera cluster.
     **Example:** 10.15.54.101,10.15.54.102.

**Note:** If you want to use Application Authentication, append a semicolon after the address followed by the authentication string, that is the full path to the Pool Entry Authorization (PEA) file. If there is no authentication string specified, CenteraVerify connects to the cluster by using the anonymous profile.
-log:
The name and location of the log file.
Example: C:\temp\CenteraVerify.log.

-fileSize:
A comma delimited string of whole numbers representing the size of each file to be written and read.
Example: 10, 100, 1000.

-store:
The name of the directory to create files to write to Centera.

-restore:
The name of the directory in which to restore files which have been read from Centera.

-retries:
A whole number indicating how many times the software should retry each Centera API call.

-retrySleep:
A whole number representing how much time (in ms) should pass before attempting the retry.

3. Argument rules:

The specific order of the arguments is not important as long as you adhere to the following rules.

- The -address argument and a value are required.
• The very first argument must be a switch, but it cannot be a value. A switch is defined as an argument that has a - (dash) preceding it.
  **Example:** -address

• A value must always follow a switch.
  **Example:** -address 127.0.0.1
  ( -address is the switch and 127.0.0.1 is a value)

• Two values that immediately follow each other are not allowed.

---

**Note:** The only required elements to the command line are the -address switch and the IP address of the cluster.
CenteraPing

Use CenteraPing to quickly check the health status of a Centera cluster. This tool is an executable that is run from a command line and which can easily be integrated into your system. It requires the IP address of one of the Access Nodes of the cluster that you want to examine. You can copy CenteraPing.exe from the SDK CD. The following section describes the usage of CenteraPing and its return values.

Running CenteraPing

To run CenteraPing, open a command line and type:

```
CenteraPing –address <IP address>
```

where `<IP address>` is a single IP address of one of the cluster’s Access Nodes.

CenteraPing will now try to make a connection to the given address to retrieve pool information. If successful, it returns `<IP address> is accessible`, otherwise it returns `<IP address> Open Error: -10020 No connection with pool`.

If the wrong syntax is used then the following is returned `usage: CenteraPing –address node1,node2,...`

The same output applies when typing CenteraPing –help
This chapter is a reference guide for application developers who are working with the Centera Access API (C).

The main sections in this chapter are:

- Function Details ................................................................. 3-2
- Pool Functions ...................................................................... 3-5
- Clip Functions ................................................................. 3-27
- Clip Handling Functions ...................................................... 3-27
- Clip Info Functions .......................................................... 3-47
- Clip Tag Functions ......................................................... 3-57
- Tag Functions ................................................................... 3-69
- Tag Handling Functions ..................................................... 3-69
- Tag Navigation Functions .................................................. 3-81
- Tag Attribute Functions ..................................................... 3-87
- Blob Handling Functions ................................................... 3-103
- Stream Functions ......................................................... 3-118
- Stream Creation Functions ............................................. 3-121
- Stream Handling Functions ............................................. 3-145
- Query Functions ............................................................. 3-153
- Monitoring Functions ...................................................... 3-161
- Error Codes ................................................................. 3-175
Function Details

This section details the syntax of Centera function calls.

Function Call

Function calls consist of a prefix followed by the actual function. The prefix refers to the category to which the function belongs.

The API functions are divided into the following categories:

1. **Pool** functions – prefix is **FPPool**
2. **Clip** functions – prefix is **FPClip**
3. **Tag** functions – prefix is **FPTag**
4. **Stream** functions – prefix is **FPStream**
5. **Query** functions – prefix is **FPQuery**
6. **Monitoring** functions – prefix is **FPMonitor** or **FPEventCallback**

**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 the letter “p”.

Commas separate parameters in the parameter list:

(parameter_type parameter1, parameter_type parameter2, ...)

Some parameter types are API specific. Refer to *API Specific Types* on page 3-3.

**Example**

FPPoolRef pPool is a reference to a pool.

We use **void** for functions that do not require a parameter and for functions with no return value:

FPPool_GetLastError (void)
API Specific Types

- `FPLong` is a 64-bit signed integer.
- `FPInt` is a 32-bit signed integer.
- `FPSShort` is a 16-bit signed integer.
- `FPClipID` is a string, used to identify a C-Clip.
- `FPErrorInfo` is a 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 3-1.
- `FPPoolInfo` is a structure that specifies pool information. The `FPPoolInfo` structure is detailed in Table 3-2.
- `FPStreamInfo` is a stream structure that passes information to and from callback functions. The `FPStreamInfo` structure is detailed in Table 3-3.

Table 3-1 FPErrorInfo Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FPInt error</code></td>
<td>The last FPLibrary error that occurred on this thread.</td>
</tr>
<tr>
<td><code>FPInt systemError</code></td>
<td>The last system error that occurred on this thread.</td>
</tr>
<tr>
<td><code>char* trace</code></td>
<td>The function trace for the last error that occurred.</td>
</tr>
<tr>
<td><code>char* message</code></td>
<td>The message associated with the FPLibrary error.</td>
</tr>
<tr>
<td><code>char* errorString</code></td>
<td>The error string associated with the FPLibrary error.</td>
</tr>
</tbody>
</table>
| `FPShort errorClass` | The class of message:  
1: Network error  
2: Server error  
3: Client error |
### Table 3-2 FPPoolInfo Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPIInt poolInfoVersion</td>
<td>The current version of this structure (2).</td>
</tr>
<tr>
<td>FPLong capacity</td>
<td>The total capacity of the pool, in bytes.</td>
</tr>
<tr>
<td>FPLong freeSpace</td>
<td>The total free usable space of the pool, in bytes.</td>
</tr>
<tr>
<td>char clusterID[128]</td>
<td>The cluster identifier of the pool.</td>
</tr>
<tr>
<td>char clusterName[128]</td>
<td>The name of the cluster.</td>
</tr>
<tr>
<td>char version[128]</td>
<td>The version of the pool server software.</td>
</tr>
<tr>
<td>char replicaAddress[256]</td>
<td>The pool address (refer to FPPool_Open()) where the C-Clips are replicated to; empty if there is no replication.</td>
</tr>
</tbody>
</table>

### Table 3-3 FPStreamInfo Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short mVersion</td>
<td>The current version of FPStreamInfo.</td>
</tr>
<tr>
<td>void *mUserData</td>
<td>Application-specific data, untouched by Generic Streams.</td>
</tr>
<tr>
<td>FPLong mStreamPos</td>
<td>The current position.</td>
</tr>
<tr>
<td>FPLong mMarkerPos</td>
<td>The position of the marker.</td>
</tr>
<tr>
<td>FPLong mStreamLen</td>
<td>The length of the stream, if known, else –1.</td>
</tr>
<tr>
<td>Boolean mAtEOF</td>
<td>If the end of stream has been reached.</td>
</tr>
<tr>
<td>Boolean mReadFlag</td>
<td>Read/write indicator, true on import, false on export.</td>
</tr>
<tr>
<td>void *mBuffer</td>
<td>The databuffer supplied by the application.</td>
</tr>
<tr>
<td>FPLong mTransferLen</td>
<td>The number of bytes to be transferred or actually transferred</td>
</tr>
</tbody>
</table>
Pool Functions

The pool functions are a set of function calls that operate at pool level. A pool normally consists of multiple clusters or nodes, each with their own IP address or DNS name and port number(s).

The system must establish a connection to the pool before performing a pool operation, with the exception of FPPool_SetGlobalOption(W) and FPPool_GetComponentVersion(W). The application should provide one or more addresses of the available Access Nodes to make a connection to a pool.

The pool functions are thread safe.

Note: You must close the connection to a pool if that connection is no longer needed.
FPPool_Open/FPPool_OpenW

Syntax:
FPPool_Open (const char *pPoolAddress)
FPPool_OpenW (const wchar_t *pPoolAddress)

Return Value:
FPPoolRef

Input Parameter:
const char *pPoolAddress/const wchar_t *pPoolAddress

Concurrency Requirement:
This function is thread safe.

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

The pool address is a comma-separated string of IP addresses or DNS names of available Access Nodes. The pool probes all addresses in this string. Each Access Node returns:

- all Access Nodes of the cluster
- the cluster ID
- the cluster time
- server features (segment size and cluster idle time)
- the replica server address (if available)

The pool recursively probes the addresses of the replica servers. It thus builds a list of clusters (addresses of Access Nodes are grouped into clusters by their cluster ID). The first cluster in the list that replies to the probe becomes the 'primary' cluster. A replication relationship is not required between clusters.

Suppose we have four clusters: N1, N2, O1, and O2 and there is a replication relationship between N1 and N2, and between O1 and O2. When reading data after FPPool_Open ("N1,O1") the clusters will be accessed in the following order: N1, N2, O1, O2.

Due to the overhead on every FPPool_Open(), we recommend reducing the number of calls to FPPool_Open() and FPPool_Close().
Note: When your local Centera cluster has been configured to replicate to a non-existing or unreachable Centera cluster, the FPPool_Open() call will take 1 minute to complete because the application tries to reach an unreachable cluster.

Every transaction uses a separate connection to the pool. This connection will not be closed when the transaction finishes but will remain available for other transactions. The pool object maintains a list of open connections. You can globally specify how many connections can be made to a pool by using the function FPPool_SetGlobalOption. The default setting is 100 and the maximum value is 999.

If replication is enabled, the IP address of the replica cluster will automatically be used for multicluster failover. Read operations will automatically failover to the replica cluster in case the source cluster cannot be reached or if the blob cannot be found. The application has to restart write and query operations by calling a new FPPool_Open in order to failover to another cluster. Delete operations will not failover.

Note: In case of a multi-threaded application, sharing one FPPoolRef over all threads gives better performance than calling FPPool_Open per thread.

Note: Use the FPPool_Close function to close the pool. Be sure to close all pool connections that are no longer needed in order to avoid performance loss and resource leakage.

Parameter:

const char *pPoolAddress / const wchar_t *pPoolAddress

pPoolAddress is a comma-separated string containing one or more addresses of the available Access Nodes 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.

Since version 2.0 of the SDK, the pool address is augmented with info for the PAI module that is in use by the application, refer to the example below. Refer to the Centera Programmer’s Guide, P/N 069001127, for more information on PAI modules.

**Note:** The maximum length of the DNS name is 256 characters.

**Note:** pPoolAddress influences the load balancing that is performed with every FPPool_Open(). Refer to the Centera Programmer’s Guide, P/N069001127, for more information on Load Balancing.

Based on the given pPoolAddress, a connection to the pool will be established as follows:

- **1 IP address:** If the SDK can establish a connection to the IP address, it automatically detects the IP addresses of all available Access Nodes in the same cluster and of the replica cluster. The disadvantage of this scenario is that if the IP address provided cannot be reached, none of the other Access Nodes can be reached and no connection will be established.

- **2 or more IP addresses of the same cluster** (recommended scenario): The SDK tries to connect to all IP addresses. If it fails to connect to one address this will not disrupt establishing a connection. The IP address of the replica cluster is automatically detected once a connection is made.

  If the first Access Node is down and the pool is not closed within 10 minutes, the SDK might return an internal error after 10 minutes. If needed, the application can increase this time by using FPPool_SetGlobalOption (FP_OPTION_CLUSTER_NON_AVAIL_TIME).

- **2 or more IP addresses of different clusters:** The SDK tries to connect to all IP addresses. It will detect which IP address belongs to which cluster. The disadvantage of this scenario is that if none of the IP addresses of the primary cluster can be reached, the secondary cluster (which can be a replica cluster) is used as the primary cluster and the application is not notified of this. Performance is affected when a secondary cluster is accessed via a WAN.
Example: Opening a pool and refer to the PEA file.

MyPoolAddress = "152.62.69.153";
MyPool = FPPool_Open (MyPoolAddress?c:\centera\rwe.pea)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_PARAM_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_PROTOCOL_ERR (internal error)
- PP_NO_SOCKET_AVAIL_ERR (network error)
- PP_PROBEPACKET_ERR (internal error)
- PP_NO_POOL_ERR (network error)
- PP_ACCESSNODE_ERR (network error)
- PP_AUTHENTICATION_FAILED_ERR (server error)
FPPool_GetLastError

Syntax:  
FPPool_GetLastError (void)

Return Value:  
FPInt

Concurrency Requirement:  
This function is thread safe.

Description:  
This function retrieves the error status of the last FPLibrary function call on the same thread and returns the number of the error. This can be an FPLibrary specific error or an OS-specific error. Refer to Error Codes on page 3-175 for a complete list of FPLibrary specific errors. If no errors are generated, the return value is ENOERR (zero).

Use FPPool_GetLastErrorInfo() to retrieve further information about the error.

Note: Check the error status after each function call, because every function call resets the error status to ENOERR before executing.
**FPPool_GetLastErrorInfo**

**Syntax:**

```c
FPPool_GetLastErrorInfo (FPErrorInfo *pErrorInfo)
```

**Return Value:**

`void`

**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. Refer to Table 3-1 on page 3-3 for details of the FPErrorInfo structure.

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

**Note:** Check the error status after each function call, because every function call resets the error status before executing.

**Note:** Do not modify the contents of the pErrorInfo structure. Do not deallocate the string member variables.
**FPPool_Close**

**Syntax:**
FPPool_Close (const FPPoolRef pPool)

**Return Value:**
void

**Input Parameter:**
const FPPoolRef pPool

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function closes the connection to the given pool and frees resources associated with the connection. Be aware that using this function on a closed pool 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.

**Parameter:**
const FPPoolRef pPool

The reference to a pool opened by FPPool_Open function. The reference may also be NULL.

**Example:**
FPPool_Close (MyPool)

**Error Handling:**
FPPool_GetLastErr() returns ENOERR (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_OBJECTINUSE_ERR (client error)
FPool_SetIntOption/FPPool_SetIntOptionW

**Syntax:**
FPPool_SetIntOption (const FPPoolRef pPool, const char *pOptionName, const FPInt pOptionValue)
FPPool_SetIntOptionW (const FPPoolRef pPool, const wchar_t *pOptionName, const FPInt pOptionValue)

**Return Value:**
void

**Input Parameters:**
const FPPoolRef pPool, const char *pOptionName/const wchar_t *pOptionName, const FPInt pOptionValue

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function sets the options for the given pool. To change global pool settings, refer to FPPool_SetGlobalOption/FPPool_SetGlobalOptionW on page 3-15.

Use the FPPool_Open function, to open and set the options for that pool.

**Parameters:**
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
- const char *pOptionName/const wchar_t *pOptionName
  pOptionName is a string with the name of the option to be set. The following initial pool options can be set:
  - FP_OPTION_BUFFERSIZE ("buffersize") – The size of internal C-Clip buffers in bytes. The default value is 16*1024. The minimum value is 1 Kbyte, the maximum value is 10 Mbyte.
  - FP_OPTION_TIMEOUT ("timeout") – 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 ("multiclusterfailover") – The option to turn automatic cluster failover for read operations on or off. By default this option is on (true or 1). To turn this option off, type false or 0.
  - FP_OPTION_DEFAULT_COLLISION_AVOIDANCE ("defaultcollisionavoidance") – 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 will use an additional blob
discriminator for read and write operations of C-Clips and blobs. Refer to the Centera Programmer’s Guide, P/N069001127, 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_BlobWrite on page 3-104.

- **FP_OPTION_PREFETCH_SIZE** ("prefetchsize") – The size of the prefetch buffer. This buffer is used to determine the size of the blob. The default size is 32 KB, the maximum size is 1 MB.

- const FPInt pOptionValue
  
  pOptionValue 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 (zero) if successful or:

- **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)
FPool_SetGlobalOption/FPPool_SetGlobalOptionW

**Syntax:**

FPPool_SetGlobalOption (const char *pOptionName, const FPInt pOptionValue)
FPPool_SetGlobalOptionW (const wchar_t *pOptionName, const FPInt pOptionValue)

**Return Value:**

void

**Input Parameters:**

const char *pOptionName/const wchar_t *pOptionName, const FPInt pOptionValue

**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function sets a library-wide option. 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.

**Parameters:**

- const char *pOptionName/const wchar_t *pOptionName

  pOptionName is a string with the name of the option to be set. The following global options can be set:

  - FP_OPTION_MAXCONNECTIONS ("maxconnections") – The maximum number of sockets that you can use globally. The default value is 100, the maximum value is 999. You can adapt this setting to your system settings and the needs of other applications.

  - FP_OPTION_RETRYCOUNT ("retrycount") – The number of times a retry must be executed. The default value is 6. If the first execution of the function fails, the system will retry the function 6 times. In total the function executes 7 times. The maximum value is 99. If you do not want functions to retry, set the retrycount to 0.

  - FP_OPTION_RETRYSLEEP ("retrysleep") – The time to wait before the failed API function call should be retried, in milliseconds. The maximum value is 100000 ms. If no

**Note:** Refer to the Centera Programmer’s Guide, P/N 069001127, for more information on the retry mechanism.
retrysleep has been defined, the SDK uses an exponential backoff scheme. The sleep time then increases after each retry, starting at 1 second, and doubles after each retry.

- **FP_OPTION_CLUSTER_NON_AVAIL_TIME**
  ("clusternonavailtime") – The time in seconds that the cluster will be marked as not available before retrying with a probe. The default setting is 10 minutes, the minimum is 0 and the maximum is 10 hours.

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

Example:

```c
const FPInt pOptionValue
pOptionValue is the value for the given option.
```

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

Error Handling:

- **FPPool_GetLastError()** returns ENOERR (zero) if successful or:
  - **FP_PARAM_ERR** (program logic error)
  - **FP_WRONG_REFERENCE_ERR** (program logic error)
  - **FP_OUT_OF_BOUNDS_ERR** (program logic error)
  - **FP_UNKNOWN_OPTION** (program logic error)
FPPool_GetPoolInfo

**Syntax:**
FPPool_GetPoolInfo (const FPPoolRef pPool, FPPoolInfo *pPoolInfo)

**Return Value:**
void

**Input Parameter:**
const FPPoolRef pPool

**Output Parameter:**
FPPoolInfo *pPoolInfo

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves information about the given pool and saves it into pPoolInfo.

**Parameters:**
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
- FPPoolInfo *pPoolInfo
  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 of the pool (in bytes) of all online nodes

**Note:** The CLI and CenteraViewer report 1000 bytes as 1Kbyte. We recommend 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 pool

**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 pool (max 128 char)
- clusterName [string]: the cluster name of the pool (max 128 char)
- **version** [string]: the version of the pool server software (max 128 char)
- **replicaAddress** [string]: the pool address of the cluster where the C-Clips are replicated to, refer to `FPPool_Open/FPPool_OpenW` on page 3-6. The maximum number is 256 characters and it will be empty if there is no replication address set.

**Note:** All pool information can only be set using the CLI. Refer to the *Centera System Operator's Guide*, P/N 069001126, for more information.

**Example:**
```
FPPoolInfo PoolInfo;
FPPool_GetPoolInfo(MyPool, &PoolInfo)
```

**Error Handling:**
`FPPool_GetLastError()` returns ENOERR (zero) if successful or:
- 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)
FPool_GetIntOption/FPPool_GetIntOptionW

**Syntax:**

- FPPool_GetIntOption (const FPPoolRef pPool, const char *pOptionName)
- FPPool_GetIntOptionW (const FPPoolRef pPool, const wchar_t *pOptionName)

**Return Value:**

FPInt

**Input Parameter:**

- const FPPoolRef, const char/wchar_t *pOptionName

**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function retrieves the value of pOptionName that is set by FPPool_SetIntOption or FPPool_SetIntOptionW.

**Parameters:**

- const FPPoolRef pPool
  - The reference to a pool opened by FPPool_Open.
- const char *pOptionName
  - Refer to FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13 for the option names and their possible values.

**Example:**

FPPool_GetIntOption(MyPool,
FP_OPTION_ENABLE_MULTICLUSTER_FAIOVER)

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_POOLCLOSED_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_UNKNOWN_OPTION (program logic error)
### FPPool_GetGlobalOption/FPPool_GetGlobalOptionW

**Syntax:**
- FPPool_GetGlobalOption (const char *pOptionName)
- FPPool_GetGlobalOptionW (const wchar_t *pOptionName)

**Return Value:**
FPInt

**Input Parameter:**
const char/wchar_t *pOptionName

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the value of pOptionName that is set by FPPool_SetGlobalOption or FPPool_SetGlobalOptionW.

**Parameter:**
- const char* pOptionName
  
  Refer to FPPool_SetGlobalOption/FPPool_SetGlobalOptionW on page 3-15 for the option names and their possible values.

**Example:**
FPPool_GetGlobalOption(FP_OPTION_RETRYCOUNT)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_UNKNOWN_OPTION (program logic error)
**FPPool_GetCapability/FPPool_GetCapabilityW**

**Syntax:**

FPPool_GetCapability (const FPPoolRef pPool, const char *pCapabilityName, const char *pCapabilityAttributeName, char *pCapabilityValue, FPInt *pCapabilityValueLen)

FPPool_GetCapabilityW (const FPPoolRef pPool, const wchar_t *pCapabilityName, const wchar_t *pCapabilityAttributeName, wchar_t *pCapabilityValue, FPInt *pCapabilityValueLen)

**Return Value:**

void

**Input Parameter:**

const FPPoolRef pPool, const char/wchar_t *pCapabilityName, const char/wchar_t *pCapabilityAttributeName, char/wchar_t *pCapabilityValue, FPInt *pCapabilityValueLen

**Output Parameter:**

char/wchar_t *pCapabilityValue, FPInt *pCapabilityValueLen

**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function returns the attribute value(s) for the given attribute name(s) within a capability tag. The capabilities are part of the application profile and refer to the operations that the SDK is allowed to perform on the server, refer to the Centera Programmer’s Guide, P/N 069001127, for more information.

**Parameters:**

- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.

- const char/wchar_t *pCapabilityName
  The name of the capability tag. Refer to Table 3-4, Capability Names and Attributes for the capability names, attribute names, and attribute values that represent the server capabilities.
### Table 3-4 Capability Names and Attributes

<table>
<thead>
<tr>
<th>Capability Name</th>
<th>Attribute Name</th>
<th>Attribute Value</th>
<th>Default Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPClip_Open, FPTag_BlobRead (Partial) are not allowed if attribute &lt;&gt; true.</td>
</tr>
<tr>
<td>write</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPClip_Write, FPTag_BlobWrite are not allowed if attribute &lt;&gt; true.</td>
</tr>
<tr>
<td>purge</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPClip_Purge, FPTag_BlobPurge are not allowed if attribute &lt;&gt; true. This attribute &lt;&gt; true on a Compliance Plus model.</td>
</tr>
<tr>
<td>delete</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPClip_Delete is not allowed if attribute &lt;&gt; true.</td>
</tr>
<tr>
<td>exist</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPClip_Exists and FPTag_BlobExists are not allowed if attribute &lt;&gt; true.</td>
</tr>
<tr>
<td>clip-constant</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>FPQuery_Open is not allowed if attribute &lt;&gt; true.</td>
</tr>
</tbody>
</table>
Pool Functions

C API Reference

const char/wchar_t *pCapabilityAttributeName
The name of the capability attribute.

char/wchar_t *pCapabilityValue
pCapabilityValue is the memory buffer that will receive the value of the capability upon successful completion of the function.

FPInt *pCapabilityValueLen
On input pCapabilityValueLen specifies the length, in characters, of the pCapabilityValue buffer. On output it specifies the actual length of the string in characters, including the end-of-string character.

Example:
Check if the SDK is allowed to purge a blob:

char vCapability[256]
FPInt vCapabilityLen = sizeof (vCapability);
FPPool_GetCapability(vPool, FP_PURGE, FP_ALLOWED, vCapability, &vCapabilityLen);

Error Handling:
FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_PARAM_ERR (program logic error)
- PP_ATTR_NOT_FOUND_ERR (program logic error)

---

Table 3-4 Capability Names and Attributes

<table>
<thead>
<tr>
<th>Capability Name</th>
<th>Attribute Name</th>
<th>Attribute Value</th>
<th>Default Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>retention</td>
<td>default</td>
<td>integer</td>
<td>0</td>
<td>If the CDF does not specify a retention period, then this value is used. 0 on a Basic Compliance model, -1 on a Compliance Plus model.</td>
</tr>
<tr>
<td>blobnaming</td>
<td>supported-schemes</td>
<td>string</td>
<td>MDS, MG</td>
<td>The supported naming schemes.</td>
</tr>
<tr>
<td>monitor</td>
<td>allowed</td>
<td>true/false</td>
<td>true</td>
<td>If true, the server supports the FPMonitor_xxx calls.</td>
</tr>
</tbody>
</table>

---
**FPPool_GetClusterTime**

**Syntax:**
FPPool_GetClusterTime (const FPPoolRef pPool, char *pClusterTime, FPInt *pClusterTimeLen)

**Return Value:**
void

**Input Parameter:**
const FPPoolRef pPool, FPInt *pClusterTimeLen

**Output Parameter:**
char *pClusterTime, FPInt *pClusterTimeLen

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the current cluster time.

**Parameters:**
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
- char *pClusterTime
  pClusterTime is the memory buffer that will store the cluster time. The time is specified in YYYY.MM.DD hh.mm.ss GMT format.
- FPInt *pClusterTimeLen
  On input pClusterTimeLen specifies the length, in characters, of the pClusterTime buffer. On output it specifies the actual length of the string in characters.

**Example:**
```c
char vClusterTime[256] ;
FPInt vClusterTimeLen ;
FPPool_GetClusterTime(vPool, vClusterTime, &vClusterTimeLen);
```

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_PARAM_ERR (program logic error)

**Syntax:**
FPPool_GetComponentVersion (const FPInt pComponent, char *pVersion, FPInt *pVersionLen)/
FPPool_GetComponentVersionW (const FPInt pComponent, wchar_t *pVersion, FPInt *pVersionLen)

**Return Value:**
void

**Input Parameter:**
const FPInt pComponent, FPInt *pVersionLen

**Output Parameter:**
char/wchar_t *pVersion, FPInt *pVersionLen

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the version of SDK components that are currently in use. Use FPPool_GetPoolInfo to retrieve the version from the server.

**Parameters:**
- const FPInt pComponent
  pComponent refers to the component queried for its version. Use one of the following numbers:
  1 = FP_VERSION_FPLIBRARY_DLL
  2 = FP_VERSION_FPLIBRARY_JAR (Java only)

**Note:** Since SDK 2.0 it is not possible to check the version of FPToolbox.dll and FPToolbox.jar. Trying this will return error -10006, FP_PARAM_ERR.

- char/wchar_t *pVersion
  pVersion is the memory buffer that will store the version number.

- FPInt *pVersionLen
  On input pVersionLen specifies the length, in characters, of the pVersion buffer. On output it specifies the actual length, in characters, of the string including the end-of-string character.

**Example:**
```c
char vVersion[128] ;
FPInt vVersionLen ;
FPPool_GetComponentVersion(FP_VERSION_FPLIBRARY_DLL, vVersion, &vVersionLen);
```
Error Handling: `FPPool_GetLastError()` returns ENOERR (zero) if successful or:

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

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

- clip handling
- clip info
- 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. This means that several threads can perform blob and tag operations within a C-Clip simultaneously.

Clip Handling Functions

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

- FPClip_Create
- FPClip_Open
- FPClip_Write
- FPClip_RawOpen
- FPClip_RawRead
- FPClip_SetRetentionPeriod
- FPClip_Purge
- FPClip_Delete
- FPClip_Close
- FPClip_Exists
**FPClip_Create/FPClip_CreateW**

**Syntax:**

- `FPClip_Create (const FPPoolRef pPool, const char *pName)`
- `FPClip_CreateW (const FPPoolRef pPool, const wchar_t *pName)`

**Return Value:**

`FPClipRef`

**Input Parameters:**

`const FPPoolRef pPool, const char *pName`/`const wchar_t *pName`

**Concurrency Requirement:**

This function is thread safe.

**Description:**

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

During the execution of `FPClip_Create`, the SDK looks for the environment variable `CENTERA_CUSTOM_METADATA`. This variable contains 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). No error is reported if the function cannot access a metadata item. The metadata can be retrieved using `FPClip_GetDescriptionAttribute()`.

If, for example, the variable contains:

```plaintext
CENTERA_CUSTOM_METADATA USER, APPLICATION, HOSTNAME
USER Doe
APPLICATION RWE Exerciser
HOSTNAME QA Test 15
```

the SDK adds the following information to the CDF:

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

**Parameters:**

- `const FPPoolRef pPool`
  The reference to a pool opened by `FPPool_Open`.
- `const char *pName`/`const wchar_t *pName`
  `pName` is a string holding the name of the C-Clip. If `pName` is NULL, the name of the C-Clip is `untitled`.

**Example:**

```c
MyClip = FPClip_Create (MyPool, "anotherclip")
```
Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_PARAM_ERR (program logic error)
- PP_SECTION_NOT_FOUND_ERR (internal error)
- PP_TAGTREE_ERR (internal error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_TAG_NOT_FOUND_ERR (internal error)
- PP_POOLCLOSED_ERR (program logic error)
**FPClip_SetName/FPClip_SetNameW**

**Syntax:**

- `FPClip_SetName (const FPClipRef pClip, const char *inClipName)`
- `FPClip_SetNameW (const FPClipRef pClip, const wchar_t *inClipName)`

**Return Value:**

`FPClipRef`

**Input Parameters:**

- `const FPClipRef pClip`
- `const char *inClipName`/`const wchar_t *inClipName`

**Concurrency Requirement:**

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

**Description:**

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

**Parameters:**

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

**Example:**

```
MyClip = FPClip_Create (MyPool, "anotherclip")
```

**Error Handling:**

`FPPool_GetLastError()` returns `ENOERR` (zero) if successful or:

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

**Syntax:**
FPClip_Open (const FPPoolRef pPool, const FPClipID pClipID, const FPInt pOpenMode)

**Return Value:**
FPClipRef

**Input Parameters:**
const FPPoolRef pPool, const FPClipID pClipID, const FPInt pOpenMode

**Concurrency Requirement:**
This function is thread safe.

**Description:**
With FPClip_Open, you can open a stored C-Clip either in tree mode structure or as a flat structure. This function returns a reference to the opened C-Clip and reads the CDF into the memory of the application server.

When using large string attributes (> 100 Kbyte) in a C-Clip, FPClip_Open() might take several minutes to execute.

**Note:**
The server will allow the application to perform this call if the server capability "read" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

**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 pPool**
  The reference to a pool opened by FPPool_Open.
- **const FPClipID pClipID**
  The C-Clip ID returned by FPClip_Write.
- **const FPInt pOpenMode**
  The method of opening a C-Clip. Current options 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.

- **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 very useful for reading C-Clips that do not fit in memory.

**Example:**

```c
MyClip = FPClip_Open (MyPool, MyClipID, FP_OPEN_ASTREE)
```

**Error Handling:**

- **FP_POOL_GETLASTERROR()** returns **ENOERR** (zero) if successful or:
  - **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_SEGMENT_ERR** (internal error)
  - **FP_BLOBIDMISMATCH_ERR** (server error)
  - **FP_SERVER_NOTREADY_ERR** (server error)
  - **FP_OPERATION_NOT_ALLOWED** (client error)
FPClip_Write

Syntax: FPClip_Write (const FPClipRef pClip, FPClipID pClipID)

Return Value: void

Input Parameter: const FPClipRef pClip

Output Parameter: FPClipID pClipID

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). Since version 2.0 this address is 64 bytes.

If collision avoidance is enabled at pool level, refer to FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13, this function returns: <C-CLIPID><REFID>. For example: 42L0M726P04T2e7QU2445E81QBK7QU2445E81QBK42L0M726P04T2. Refer to the Centera Programmer’s Guide, P/N069001127, for more information on collision avoidance.

If fast blob naming is enabled, files smaller than the threshold (by default 250 KB) will have a Content Address similar to the one that is created when Collision Avoidance has been enabled.

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

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

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 pClip
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.
• FPClipID pClipID
  The C-Clip ID that the function returns.

  **Example:** FPClip_Write (MyClip, MyClipID)

  **Error Handling:** FPPool_GetLastError() returns ENOERR (zero) if successful or:

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

**Syntax:**
FPClip_RawRead (const FPClipRef pClip, const FPStreamRef pStream)

**Return value:**
void

**Input Parameters:**
const FPClipRef pClip, const FPStreamRef pStream

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

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

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 pClip
  The reference to a C-Clip returned by FPClip_Open.
- const FPStreamRef pStream
  The reference to a stream that has been created by an FPStreamCreateXXX function or a generic stream for writing. The stream does not have to support marking.

**Example:**
FPClip_RawRead (MyClip, MyStream)

Refer to Duplicate Address Detection in the Centera Programmer’s Guide, P/N 0690001127, for a code example.

**Error Handling:**
- PP_PARAM_ERR (internal error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_STREAM_ERR (client error)
- stream related errors (refer to Stream Functions on page 3-118 for more information)
**FPClip_RawOpen**

**Syntax:**
FPClip_RawOpen (const FPPoolRef pPool, const FPClipID pClipID, const FPStreamRef pStream, const FPLong pOptions)

**Return Value:**
FPClipRef

**Input Parameters:**
const FPPoolRef pPool, const FPClipID pClipID, const FPStreamRef pStream, const FPLong pOptions

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function reads the content of pStream and creates a new C-Clip. The new C-Clip ID has to match the given C-Clip ID. If collision avoidance has been enabled at pool level, refer to FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13, the new C-Clip ID has to match the given C-Clip ID and the RefID of the new C-Clip ID has to match the RefID inside the given C-Clip, refer to FPClip_Write on page 3-33.

If the fast blob naming scheme is used to create the new C-Clip this function returns FP_OPERATION_NOT_ALLOWED when used against C-Clips from a Centera prior to version 2.1.

If the fast blob naming scheme is used to create the new C-Clip this function returns FP_OPERATION_NOT_ALLOWED when used against C-Clips from a Centera prior to version 2.1.

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

**Parameters:**
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
- const FPClipID pClipID
  The C-Clip used to reference the C-Clip that has to be read from the stream.
- const FPStreamRef pStream
  The reference to an input stream. Marking support is not necessary.
- const FPLong Options
  This option is currently not in use but reserved for future expansion. Set this option to 0.
Example: \[\text{NewClip} = \text{FPClip\_RawOpen} \text{ \{MyPool, MyClipID, MyStream, 0\}}\]

Error Handling: \[\text{FPPool\_GetLastError()} \text{ returns ENOERR (zero) if successful or:}\]

- \text{FP\_PARAM\_ERR (internal error)}
- \text{FP\_WRONG\_REFERENCE\_ERR (program logic error)}
- \text{FP\_BLOBIDMISMATCH\_ERR (server error)}
- \text{FP\_STREAM\_ERR (client error)}
- \text{FP\_VERSION\_ERR (internal error)}
- stream related errors (refer to \text{Stream Functions} on page 3-118 for more information)
FPClip_SetRetentionPeriod

Syntax: 
FPClip_SetRetentionPeriod (const FPClipRef pClip, const FPLong pRetentionSecs)

Return Value: 
void

Input Parameters: 
const FPClipRef pClip, const FPLong pRetentionSecs

Concurrency Requirement: 
This function is thread safe.

Description: 
This function sets the retention period, in seconds, of the given C-Clip. The retention period specifies how long a C-Clip has to be stored before you are allowed to delete it. When the CDF of the C-Clip does not contain a retention period, the C-Clip will be stored with the retention period that is specified by the server (refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for the possible server settings). The retention period set by the SDK overrules the retention period set by the server. You cannot change the retention period of a C-Clip once the C-Clip has been stored. Modifying the retention period of a stored C-Clip will generate a new C-Clip.

Note: You might not be able to delete a file immediately after its retention period has expired due to the drift in the server clock. This delay can take up to 1 day over the span of 20 years.

Note: Be aware that the retention period is calculated from the date that the C-Clip was created (the creation date) and not from the last modification date. If you write a C-Clip, open it, modify something and write another C-Clip, both C-Clips will have the same creation date. The retention period of the second C-Clip will be based on the creation date and not on the modification date.

Parameters:
- const FPClipRef pClip
  The reference to a C-Clip that was opened by the FPClip_Open or FPClip_Create function.
- const FPLong pRetentionSecs
  pRetentionSecs is the retention period in seconds or one of the following options:
• FP_NO_RETENTION_PERIOD (0) – the C-Clip can always be deleted
• FP_INFINITE_RETENTION_PERIOD (-1) – the C-Clip can never be deleted
• FP_DEFAULT_RETENTION_PERIOD (-2) – the retention period depends on the cluster setting (for a Compliance Plus model the default retention period is -1 or infinite, for a Basic Compliance model it is 0)

Note: The value of the retention period set by the SDK overrules the retention period set by the server.

Example: FPClip_SetRetentionPeriod(vClip, FP_INFINITE_RETENTION_PERIOD);

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:
• FP_PARAM_ERR (internal error)
• FP_WRONG_REFERENCE_ERR (program logic error)
FPClip_Delete

**Syntax:**
FPClip_Delete (const FPPoolRef pPool, const FPClipID pClipID)

**Return Value:**
void

**Input Parameters:**
const FPPoolRef pPool, const FPClipID pClipID

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function deletes the given CDF 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 true. Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities and to FPClip_SetRetentionPeriod on page 3-38 for more information on retention periods.

C-Clips will only be deleted when all the parity fragments or mirror copies of the C-Clip have been found and successfully removed. When some nodes are down or when network problems occur, the delete or purge will fail and error -10156, FP_TRANSACTION_FAILED_ERR, will be returned. Try again later.

This function returns FP_OPERATION_NOT_ALLOWED if the retention period is not yet expired and/or if the server capability "delete" is false. In that case, the CDF will not be deleted.

Note that this only deletes associated blobs as well if this is the only C-Clip that references the associated blobs. To delete blobs, refer to FPTag_BlobPurge on page 3-114. Refer to the Centera Programmer’s Guide, P/N 069001127, for more information on deleting data.

**Note:** The server will allow the application to perform this call if the server capability "delete" is true. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

**Note:** Following the successful deletion of the C-Clip, associated blobs may become unreferenced until they are removed by Garbage Collection.
Clip Handling Functions

Parameters:
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
- const FPClipID pClipID
  The ID of a C-Clip.

Example:
FPClip_Delete (MyPool, MyClipID)

Error Handling:
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_PARAM_ERR (program logic error)
- PP_NO_POOL_ERR (network error)
- PP_NO_SOCKET_AVAIL_ERR (network error)
- PP_PROTOCOL_ERR (internal error)
- PP_PROBEPACKET_ERR (internal error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_SERVER_ERR (server error)
- PP_CONTROLFIELD_ERR (server error)
- PP_NUMLOC_FIELD_ERR (server error)
- PP_UNKNOWN_OPTION (internal error)
- PP_CLIP_NOT_FOUND_ERR (program logic error)
- PP_VERSION_ERR (internal error)
- PP_NOT_RECEIVE_REPLY_ERR (network error)
- PP_SEGDATA_ERR (internal error)
- PP_SERVER_NOTREADY_ERR (server error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_BLOBBUSY_ERR (server error)
- PP_OPERATION_NOT_ALLOWED (client error)
- PP_TRANSACTION_FAILED_ERR (server error)
FPClip_Purge

Earlier versions of the Centera Access API enabled this function to purge a CDF and associated blob(s) from the pool. From version 2.0 onwards this function only purges the CDF from the first writable cluster in the given pool but not its associated blob(s). This function does not take the retention period into account and removes the CDF, regardless of the C-Clip’s expiration date.

Although this function is still available, we recommend using the function FPTag_BlobPurge() to delete blob data, and FPClip_Delete() to delete a CDF. You can also use FPTag_Delete() to delete a tag without deleting the blob data. Refer to the Centera Programmer’s Guide, P/N 069001127, for more information on deleting data.

**Note:** The server will allow the application to perform this call only if the server capability "purge" is true. A Compliance Plus model will not allow this call and will return the error FP_OPERATION_NOT_ALLOWED.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

**Note:** Errors will occur if an application tries to read a C-Clip that no longer exists. The application developer is responsible for writing code that can handle this scenario.

C-Clips will only be purged when all the parity fragments or mirror copies of the C-Clip have been found and successfully removed. When some nodes are down or when network problems occur, the delete or purge will fail and error -10156, FP_TRANSACTION_FAILED_ERR, will be returned. Try again later.

**Error Handling:** FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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)
- PP_CONTROLFIELD_ERR (server error)
- PP_NUMLOC_FIELD_ERR (server error)
- PP_UNKNOWN_OPTION (internal error)
- PP_CLIP_NOT_FOUND_ERR (program logic error)
- PP_VERSION_ERR (internal error)
- PP_NOT_RECEIVE_REPLY_ERR (network error)
- PP_SEGDATA_ERR (internal error)
- PP_SERVER_NOTREADY_ERR (server error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_BLOBBUSY_ERR (server error)
- PP_OPERATION_NOT_ALLOWED (client error)
- PP_TRANSACTION_FAILED_ERR (server error)
**FPClip_Close**

**Syntax:**

FPClip_Close (const FPClipRef pClip)

**Return Value:**

void

**Input Parameter:**

const FPClipRef pClip

**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. Be aware that using this function on a C-Clip that has already been closed may produce unwanted results.

*Note:* All changes will be lost, if the C-Clip has been modified since it was opened but has not yet been written to the pool when you close it.

**Parameter:**

const FPClipRef pClip

The reference to a C-Clip that was opened by the FPClip_Open or FPClip_Create function. The reference may also be NULL.

**Example:**

FPClip_Close (MyClip)

**Error Handling:**

FPool_GetLastError() returns ENOERR (zero) if successful or:

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

Syntax:

FPClip_Exists (const FPPoolRef pPool, const FPClipID pClipID)

Return Value:

Boolean

Input Parameters:

const FPPoolRef pPool, const FPClipID pClipID

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. This function will not failover.

Note: The server will allow the application to perform this call if the server capability "exist" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

Parameters:

• const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.

• const FPClipID pClipID
  The ID of a C-Clip.

Example:

FPClip_Exists (MyPool, MyClipID)

Error Handling:

FPPool_GetLastError() returns ENOERR (zero) if successful or:

• PP_PARAM_ERR (program logic error)
• PP_NO_POOL_ERR (network error)
• PP_NO_SOCKET_AVAIL_ERR (network error)
• PP_PROTOCOL_ERR (internal error)
• PP_PROBEPACKET_ERR (internal error)
• PP_WRONG_REFERENCE_ERR (program logic error)
• PP_SERVER_ERR (server error)
• PP_NUMLOC_FIELD_ERR (server error)
• PP_NOT_RECEIVE_REPLY_ERR (network error)
• PP_CONTROLFIELD_ERR (server error)
• PP_CLIP_NOT_FOUND_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
- FP_OPERATION_NOT_ALLOWED (client error)
Clip Info Functions

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

- FPClip_GetPoolRef
- FPClip_GetNumTags
- FPClip_GetNumBlobs
- FPClip_GetTotalSize
- FPClip_GetClipID
- FPClip_GetName/FPClip_GetNameW
- FPClip_GetCreationDate
- FPClip_GetRetentionPeriod
- FPClip_IsModified
**FPClip_GetPoolRef**

**Syntax:**
FPClip_GetPoolRef (const FPClipRef pClip)

**Return Value:**
FPPoolRef

**Input Parameter:**
const FPClipRef pClip

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the reference to the pool in which the given C-Clip has been opened. The application should either close the pool reference returned by FPPool_Open or by this function, but not both.

**Parameter:**
const FPClipRef pClip
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

**Example:**
MyPool = FPClip_GetPoolRef (MyClip)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
- FP_CLIPCLOSED_ERR (program logic error)
### FPClip_GetNumTags

**Syntax:**

```
FPClip_GetNumTags (const FPClipRef pClip)
```

**Return Value:**

FPInt

**Input Parameter:**

const FPClipRef pClip

**Concurrency Requirement:**

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

**Description:**

This function retrieves the number of tags that are stored in the given C-Clip. Only tags that are visible to the user are taken into account. This function returns -1, when an error occurs. C-Clips created with an SDK version lower than 1.2 have to be opened as tree to retrieve the number of tags, otherwise the error FP_ATTR_NOT_FOUND_ERR will be returned.

**Parameter:**

const FPClipRef pClip  
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

**Example:**

```
NumTags = FPClip_GetNumTags (MyClip)
```

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_CLIPCLOSED_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
**FPClip_GetNumBlobs**

**Syntax:**
FPClip_GetNumBlobs (const FPClipRef pClip)

**Return Value:**
FPInt

**Input Parameter:**
const FPClipRef pClip

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

**Description:**
This function retrieves the number of blobs that are stored in the given C-Clip.

**Parameter:**
const FPClipRef pClip
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

**Example:**
NumBlobs = FPClip_GetNumBlobs (MyClip)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- 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_GetTotalSize**

**Syntax:**
```
FPClip_GetTotalSize (const FPClipRef pClip)
```

**Return Value:**
FPLong

**Input Parameter:**
const FPClipRef pClip

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

**Description:**
This function retrieves the total size (in bytes) of all blobs stored in the given C-Clip.

**Parameter:**
const FPClipRef pClip
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

**Example:**
```
ClipSize = FPClip_GetTotalSize (MyClip)
```

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_SECTION_NOT_FOUND_ERR (internal error)
- PP_WRONG_REFERENCE_ERR (internal error)
- PP_TAG_NOT_FOUND_ERR (internal error)
- PP_POOLCLOSED_ERR (program logic error)
FPClip_GetName/FPClip_GetNameW

**Syntax:**

FPClip_GetName (const FPClipRef pClip, char *pName, FPInt *pNameLen)

FPClip_GetNameW (const FPClipRef pClip, wchar_t *pName, FPInt *pNameLen)

**Return Value:**

void

**Input Parameters:**

const FPClipRef pClip, FPInt *pNameLen

**Output Parameters:**

char *pName/wchar_t *pName, FPInt *pNameLen

**Concurrency Requirement:**

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

**Description:**

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

**Parameters:**

- const FPClipRef pClip
  The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

- char *pName/wchar_t *pName
  pName is the buffer that will store the name of the C-Clip. The name will be truncated to the buffer length as specified by pNameLen.

- FPInt *pNameLen
  Input: The reserved length, in characters, of the buffer for pName.  
  Output: The actual length, in characters, of the name, including the end-of-string character.

**Example:**

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

**Error Handling:**

FPProfiler_GetLastError() returns ENOERR (zero) if successful or:

- 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_GetClipID**

**Syntax:**
```
FPClip_GetClipID (const FPClipRef pClip, FPClipID pClipID)
```

**Return Value:**
void

**Input Parameter:**
const FPClipRef pClip

**Output Parameter:**
FPClipID pClipID

**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 pClipID.
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 pClip**
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.
- **FPClipID pClipID**
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 (zero) if successful or:
- **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:**
```c
FPClip_GetCreationDate (const FPClipRef pClip, char *pDate, FPInt *pDateLen)
```

**Return Value:**
void

**Input Parameters:**
const FPClipRef pClip, FPInt *pDateLen

**Output Parameters:**
char *pDate, FPInt *pDateLen

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

**Description:**
This function retrieves the creation date of the C-Clip. The time is specified in UTC (Coordinated Universal Time, also known as GMT – Greenwich Mean Time) and is based on the cluster time.

February 21st, 2002 is expressed as: 2002.02.21 10:46:32 GMT

**Parameters:**
- const FPClipRef pClip
  The reference to a C-Clip opened by FPClip_Open or FPClip_Create.
- char *pDate
  pDate 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 pDateLen.
- FPInt *pDateLen
  Input: The reserved length, in characters, of the buffer for pDate. Output: The actual length of the date string, including the end-of-string character.

**Example:**
```c
FPInt datesize;
datesize = MAX_DATE_SIZE;
char date[MAX_DATE_SIZE];
FPClip_GetCreationDate (MyClip, date, &datesize)
```

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- 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_GetRetentionPeriod

Syntax: FPClip_GetRetentionPeriod (const FPClipRef pClip)

Return Value: FPLong

Input Parameters: const FPClipRef pClip

Concurrency Requirement: This function can only access one C-Clip reference.

Description: This function retrieves the retention period in seconds of the given C-Clip, that has been set by FPClip_SetRetentionPeriod, or one of the default retention periods. The value returned by this function is the retention period as it was defined when the C-Clip was stored. Refer to FPClip_SetRetentionPeriod on page 3-38 for more information on the possible retention values.

Parameters:
- const FPClipRef pClip
  The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

Example: vNumSeconds = FPClip_GetRetentionPeriod(vClip);

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_WRONG_REFERENCE_ERR (program logic error)
**FPClip_IsModified**

**Syntax:**
FPClip_IsModified (const FPClipRef pClip)

**Return Value:**
Boolean

**Input Parameter:**
const FPClipRef pClip

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

**Description:**
This function retrieves the modification status of a C-Clip that you have already opened or created. This function returns true if the C-Clip has been modified locally 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 pClip
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

**Example:**
FPClip_IsModified (MyClip)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
Clip Tag Functions

This section describes the functions that manipulate a single tag from a C-Clip. Because these functions operate on the level of a C-Clip – and not on the level of a tag as the tag functions described in Tag Functions on page 3-69 – they are listed as clip functions.

- FPClip_GetTopTag
- FPClip_FetchNext
- FPClip_SetDescriptionAttribute(W)
- FPClip_GetDescriptionAttribute(W)
- FPClip_GetNumDescriptionAttributes
- FPClip_GetDescriptionAttributeIndex(W)
- FPClip_RemoveDescriptionAttribute(W)
FPClip_GetTopTag

Syntax:  
FPClip_GetTopTag (const FPClipRef pClip)

Return Value:  
FPTagRef

Input Parameter:  
const FPClipRef pClip

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

Description:  
This function retrieves a reference to the top-level tag in a C-Clip that has been opened in tree mode. If the tag structure is empty, you can use the returned tag as the parent tag to create the first tag in the tree.

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

This function retrieves a reference to the first tag in a C-Clip that has been opened in flat mode. You can then only use FPClip_FetchNext to retrieve following tags.

Note: You are not allowed to call FPTag_GetTagName on a top-level tag.

The top tag is not a tag as other tags, it has no name, and has no associated attributes or blobs.

Parameter:  
const FPClipRef pClip  
The reference to a C-Clip opened by FPClip_Open or FPClip_Create.

Example:  
MyTag = FPClip_GetTopTag (MyClip)

Error Handling:  
FP Pool GetLast Error () returns ENOERR (zero) if successful or:
- 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)
FPClip_FetchNext

Syntax:  
FPClip_FetchNext (const FPClipRef pClip)

Return Value:  
FPTagRef

Input Parameter:  
const FPClipRef pClip

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

Description:  
This function retrieves the first tag in the C-Clip structure of the given C-Clip if this is the first call to the function. Subsequent function calls retrieve the next tags in the tag structure.

If the C-Clip has been opened in tree mode, the traversal order is depth-first. If the tree has been opened in flat mode, the function returns the next tag as it has been written to the CDF. In both cases the result will be the same.

Note: You can get access to a tag structure of a C-Clip only by using this function or FPClip_GetTopTag. Use FPClip_GetTopTag if the C-Clip has been opened in tree mode.

This function returns NULL if it does not find any tag or if it encountered the last tag.

Note: Do not forget to close the C-Clip and FPTag after you have used this function.

Parameter:  
const FPClipRef pClip  
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 (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_TAGTREE_ERR (internal error)
- FP_SECTIONS_NOT_FOUND_ERR (internal error)
- FP_POOLCLOSED_ERR (program logic error)
**FPClip_SetDescriptionAttribute/FPClip_SetDescriptionAttributeW**

**Syntax:**

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

FPClip_SetDescriptionAttributeW (const FPClipRef inClip, const wchar_t *inAttrName, const wchar_t *inAttrValue)

**Return Value:**

void

**Input Parameters:**

const FPClipRef inClip, const char *inAttrName/const wchar_t *inAttrName, const char *inAttrValue/const wchar_t *inAttrValue

**Concurrency Requirement:**

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

**Description:**

This function adds the given attribute to the CDF of the given C-Clip. The attribute is added to the user tag in the CDF section that contains the C-Clip metadata. You can also add user-defined attributes using the environment variable CENTERA_CUSTOM_METADATA. This variable will be read during FPClip_Create().

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 the FPClip_Open or FPClip_Create function.

- const char *inAttrName/const wchar_t *inAttrName
  
inAttrName is the buffer that contains the name of the attribute that needs to be added.

- const char *inAttrValue/const wchar_t *inAttrValue
  
inAttrValue is the buffer that contains the value of the attribute that needs to be added.

**Example:**

MyClip = FPClip_Create(MyPool, "test");
FPClip_SetDescriptionAttribute(MyClip, "company", "com.acme")

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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)
FPClip_GetDescriptionAttribute/FPClip_GetDescriptionAttributeW

**Syntax:**

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

FPClip_GetDescriptionAttributeW (const FPClipRef inClip, const wchar_t *inAttrName, const wchar_t *outAttrValue, FPInt *ioAttrValueLen)

**Return Value:**

void

**Input Parameters:**

const FPClipRef inClip, const char *inAttrName/const wchar_t *inAttrName, FPInt *ioAttrValueLen

**Output Parameters:**

const char *outAttrValue/const wchar_t *outAttrValue, FPInt *ioAttrValueLen

**Concurrency Requirement:**

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

**Description:**

This function retrieves the value of the given attribute from the given C-Clip. The attribute-value pair resides in the user tag in the CDF section that contains the C-Clip metadata.

**Parameters:**

- const FPClipRef inClip
  The reference to a C-Clip that was opened by the FPClip_Open or FPClip_Create function.

- const char *inAttrName/const wchar_t *inAttrName
  inAttrName is the buffer that contains the name of the attribute of which the value needs to be retrieved.

- const char *outAttrValue/const wchar_t *outAttrValue
  outAttrValue is the buffer that will hold the attribute value.

- FPInt *ioAttrValueLen
  Input: the length in wide characters of outAttrValue. At most (*pNameLen-1) characters will be written to the buffer plus the end_of_string character.
  Output: the actual number of characters in outAttrValue, including the end_of_string character.

**Example:**

```c
char vBuffer[1024];
FPInt l=sizeof(vBuffer);
FPClip_GetDescriptionAttribute(MyClip, "company", vBuffer, &l);
```
Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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_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 retrieves the number of the user-defined and standard description attributes of the given C-Clip. These attributes reside in the user tag in the CDF section that contains the C-Clip metadata.

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

Example: 
FPInt vNum;
vNum = FPClip_GetDescriptionAttributes(MyClip);

Error Handling: 
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_WRONG_REFERENCE_ERR (program logic error)
**FPClip_GetDescriptionAttributeIndex/FPClip_GetDescriptionAttributeIndexW**

**Syntax:**

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

FPClip_GetDescriptionAttributeIndexW (const FPClipRef inClip, const FPInt inIndex, wchar_t *outAttrName, FPInt *ioAttrNameLen, wchar_t *outAttrValue, FPInt *ioAttrValueLen)

**Return Value:** void

**Input Parameters:**

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

**Output Parameters:**

- char *outAttrName/wchar_t *outAttrName, FPInt *ioAttrNameLen
- char *outAttrValue/wchar_t *outAttrValue, FPInt *ioAttrValueLen

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

**Description:** This function retrieves the name and the value of an attribute from the given C-Clip according to the given index. The attribute-value pair resides in the user tag in the CDF section that contains the C-Clip metadata.

**Parameters:**

- const FPClipRef inClip
  The reference to a C-Clip that was opened by the FPClip_Open or FPClip_Create function.

- const FPInt inIndex
  The index number (zero based) of the attribute that has to be retrieved.

- char *outAttrName/wchar_t *outAttrName
  outAttrName is the buffer that will hold the attribute name.

- char *outAttrValue/wchar_t *outAttrValue
  outAttrValue is the buffer that will hold the attribute value.

- FPInt *ioAttrNameLen
  Input: the length in characters of outAttrName. At most (*pNameLen-1) characters will be written to the buffer plus the end_of_string character.
Output: the actual number of characters in `outAttrName`, including the end_of_string character.

- `FPInt *ioAttrValueLen`
  Input: the length in wide characters of `outAttrValue`. At most
  `(*pNameLen-1)` characters will be written to the buffer plus the
  end_of_string character.
  Output: the actual number of characters in `outAttrValue`, including the end_of_string character.

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

**Error Handling:**
`FPPool_GetLastError()` returns ENOERR (zero) if successful or:

- `PP_PARAM_ERR` (program logic error)
- `PP_WRONG_REFERENCE_ERR` (program logic error)
- `PP_ATTR_NOT_FOUND_ERR` (internal error)
**FPClip_RemoveDescriptionAttribute/FPClip_RemoveDescriptionAttributeW**

**Syntax:**
```c
FPClip_RemoveDescriptionAttribute (const FPClipRef inClip, const char *inAttrName)
FPClip_RemoveDescriptionAttributeW (const FPClipRef inClip, const wchar_t *inAttrName)
```

**Return Value:** void

**Input Parameters:**
- const FPClipRef inClip
- const char *inAttrName
- const wchar_t *inAttrName

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

**Description:**
This function removes the given attribute from the CDF of the given C-Clip. The attribute resides in the `user` tag in the CDF section that contains the C-Clip metadata.

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

**Example:**
```c
FPClip_RemoveDescriptionAttribute(MyClip, "company");
```

**Error Handling:**
- FPPool_GetLastError() returns ENOERR (zero) if successful or:
  - FP_PARAM_ERR (program logic error)
  - FP_WRONG_REFERENCE_ERR (program logic error)
  - FP_ATTR_NOT_FOUND_ERR (internal error)
Tag Functions

The tag functions operate at the level of a C-Clip tag. 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 the tag. Do not forget to close the tag afterwards.

Tag Handling Functions

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

- `FPTag_Create/FPTag_CreateW`
- `FPTag_Copy`
- `FPTag_GetPoolRef`
- `FPTag_GetClipRef`
- `FPTag_GetTagName/FPTag_GetTagNameW`
- `FPTag_GetBlobSize`
- `FPTag_Close`
- `FPTag_Delete`
FPTag_Create/FPTag_CreateW

**Syntax:**

FPTag_Create (const FPTagRef pParent, const char *pName)
FPTag_CreateW (const FPTagRef pParent, const wchar_t *pName)

**Return Value:**

FPTagRef

**Input Parameters:**

const FPTagRef pParent, const char *pName/const wchar_t *pName

**Concurrency Requirement:**

This function is thread safe.

**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 to the maximum size of a C-Clip: 100 Mbyte. The maximum number of nested tags is 64.

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

**Parameters:**

- const FPTagRef pParent
  The reference to the parent tag of the new tag that you are creating.
- const char *pName/const wchar_t *pName
  pName is the buffer that holds the name of the new tag. The value cannot be NULL.

**Note:** The name must be xml compliant.

**Example:**

MyTag = FPTag_Create (Parent, "tagname")

**Error Handling:**

FPool_GetLastError() returns ENOERR (zero) if successful or:

- 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_Copy

**Syntax:**
FPTag_Copy (const FPTagRef pTag, const FPTagRef pNewParent, const FPInt pOptions)

**Return Value:**
FPTagRef

**Input Parameters:**
const FPTagRef pTag, const FPTagRef pNewParent, const FPInt pOptions

**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 should belong to a C-Clip that resides on the same cluster. When copying multiple children of the same tag, the order of the tags will be 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 only supported 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 clusters, the call will return the FP_OPERATION_NOT_SUPPORTED error.

**Parameters:**
- const FPTagRef pTag
  The reference to the tag that you want to copy.
- const FPTag pNewParent
  The reference to the new destination tag. To copy a tag, a reference to an existing destination tag is required.
- const FPInt pOptions
  You can use one or more of the following options:
  - FP_OPTION_NO_COPY_OPTIONS (0) – only the tag and its attributes are copied.
  - FP_OPTION_COPY_BLOBDATA (1) – the tag attributes and the blob data are copied.
FP_OPTION_COPY_CHILDREN (2) – the children of the tag are copied. You have to specify this option, if pTag is the top tag. The top tag itself will then not be copied. If pTag is a parent of pNewParent then the error FP_OUT_OF_BOUNDS_ERR will be returned.

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

```c
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 (zero) if successful or:

- 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_GetPoolRef

Syntax: FPTag_GetPoolRef (const FPTagRef pTag)

Return Value: FPPoolRef

Input Parameter: const FPTagRef pTag

Concurrency Requirement: This function is thread safe.

Description: This function retrieves the reference to the pool in which the given tag was opened. The application should either close the pool reference returned by FPPool_Open or by this function, but not both.

Parameter: const FPTagRef pTag

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 (zero) if successful or:
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_CLIPCLOSED_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
FPTag_GetClipRef

Syntax: FPTag_GetClipRef (const FPTagRef pTag)

Return Value: FPClipRef

Input Parameter: const FPTagRef pTag

Concurrence Requirement: This function is thread safe.

Description: This function retrieves the reference to the C-Clip in which the given tag was opened. The application should either close the C-Clip reference returned by FPClip_Open/FPClip_Create or by this function, but not both.

Parameter: const FPTagRef pTag
The reference to a tag that any of the tag functions have opened or created.

Example: MyClip = FPTag_GetClipRef (MyTag)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_CLIPCLOSED_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
**FPTag_GetTagName/FPTag_GetTagNameW**

**Syntax:**

FPTag_GetTagName (const FPTagRef pTag, char *pName, FPInt *pNameLen)

FPTag_GetTagNameW (const FPTagRef pTag, wchar_t *pName, FPInt *pNameLen)

**Return Value:**
void

**Input Parameters:**
const FPTagRef pTag, FPInt *pNameLen

**Output Parameters:**
char *pName/wchar_t *pName, FPInt *pNameLen

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the name of the given tag in the opened C-Clip. This function requires a buffer for the name (pName) and the buffer length (pNameLen).

**Note:** You cannot call this function on a top tag, that is, on a tag that is returned by FPClip_GetTopTag or a tag of which FPTag_GetParent returns NULL.

**Parameters:**

- const FPTagRef pTag
  The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetFirstChild, FPTag_GetSibling, FPTag_GetPrevSibling, or FPClip_FetchNext).

- char *pName/wchar_t *pName
  pName is the buffer that will hold the name of the tag. The name will be truncated to the buffer length as specified by pNameLen.

- FPInt *pNameLen
  Input: The length, in characters, of the buffer for pName. 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 (zero) if successful or:

- 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)
FPTag_GetBlobSize

Syntax: FPTag_GetBlobSize (const FPTagRef pTag)

Return Value: FPLong

Input Parameters: const FPTagRef pTag

Concurrency Requirement: This function is thread safe.

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

Parameter: const FPTagRef pTag

The reference to a tag (as returned from the functions 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 (zero) if successful or:

• FP_WRONG_REFERENCE_ERR (program logic error)
FPTag_Close

Syntax: FPTag_Close (const FPTagRef pTag)

Return Value: void

Input Parameter: const FPTagRef pTag

Concurrency Requirement: This function is thread safe.

Description: This function closes the given tag. It also closes all structures and frees up all memory allocated to that tag without deleting the tag from the C-Clip. Be aware that using this function on a tag that has already been closed may produce unwanted results.

Parameter: const FPTagRef pTag
The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetFirstChild, FPTag_GetSibling, FPTag_GetPrevSibling, FPClip_GetTopTag or FPClip_FetchNext). The reference may also be NULL.

Example: FPTag_Close (MyTag)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

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

Syntax: FPTag_Delete (const FPTagRef pTag)

Return Value: void

Input Parameter: const FPTagRef pTag

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 tag refers to data (a blob tag), it will not delete that data (refer to FPTag_BlobPurge on page 3-114).

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 pTag becomes invalid. Any function call to the tag (for example the FPTag_Close function) will result in an FP_WRONG_REFERENCE_ERR error.

Parameter: const FPTagRef pTag

The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetFirstChild, FPTag_GetSibling or FPTag_GetPrevSibling).

Example: FPTag_Delete (MyTag)

Error Handling: FPPool_GetLastErrorCode() returns ENOERR (zero) if successful or:

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

Figure 3-1 shows how the tags are structured hierarchically and how the tag navigation functions operate if the C-Clip opens in tree mode.
Figure 3-1  Tag Structure and Navigation
FPTag_GetFirstChild

Syntax: FPTag_GetFirstChild (const FPTagRef pTag)

Return Value: FPTagRef

Input Parameter: const FPTagRef pTag

Concurrency Requirement: This function is thread safe.

Description: This function retrieves the first child tag of the given tag. This means 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 3-1). This function returns NULL if no child tag can be found.

Parameter: const FPTagRef pTag

The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPClip_GetTopTag, FPTag_GetFirstChild, FPTag_GetSibling or FPTag_GetPrevSibling).

Example: MyChildTag = FPTag_GetFirstChild (MyTag)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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 pTag)

Return Value:  
FPTagRef

Input Parameter:  
const FPTagRef pTag

Concurrency Requirement:  
This function is thread safe.

Description:  
This function retrieves the sibling tag of the given tag. This means 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 3-1 on page 3-82).

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.

Parameter:  
const FPTagRef pTag
The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetSibling, FPTag_GetPrevSibling, or FPTag_GetFirstChild).

Example:  
MySibling = FPTag_GetSibling (MyTag)

Error Handling:  
FP.Pool_GetLastError() returns ENOERR (zero) if successful or:

- 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 pTag)

**Return Value:**
FPTagRef

**Input Parameter:**
const FPTagRef pTag

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the previous sibling tag of the given tag. This means 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 3-1 on page 3-82).

This function returns NULL, if the system cannot find a sibling tag.

**Parameter:**
const FPTagRef pTag
The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetSibling, FPTag_GetPrevSibling, or FPTag_GetFirstChild).

**Example:**
MySibling = FPTag_GetPrevSibling (MyTag)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_OPERATION_NOT_SUPPORTED (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
FPTag_GetParent

Syntax: FPTag_GetParent (const FPTagRef pTag)

Return Value: FPTagRef

Input Parameter: const FPTagRef pTag

Concurrence Requirement: This function is thread safe.

Description: This function retrieves the parent tag of the given tag. This means 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 3-1). This function returns NULL if the system cannot find a parent tag.

Parameter: const FPTagRef pTag

The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetFirstChild, FPTag_GetSibling, or FPTag_GetPrevSibling).

Example: MyParent = FPTag_GetParent (MyTag)

Error Handling: FPPool_GetLastErr() returns ENOERR (zero) if successful or:

- 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 will create 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 maximum number of attributes within a tag is 64.
**FPTag.SetStringAttribute/FPTag.SetStringAttributeW**

**Syntax:**
FPTag.SetStringAttribute (const FPTagRef pTag, const char *pAttrName, const char *pAttrValue)
FPTag.SetStringAttributeW (const FPTagRef pTag, const wchar_t *pAttrName, const wchar_t *pAttrValue)

**Return Value:**
void

**Input Parameters:**
const FPTagRef pTag, const char *pAttrName/const wchar_t *pAttrName, const char *pAttrValue/const wchar_t *pAttrValue

**Concurrency Requirement:**
This function is thread safe.

**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 (pTag), the attribute name (pAttrName), and the attribute value (pAttrValue).

**Note:** The maximum number of attributes within one tag is 64.

**Parameters:**
- const FPTagRef pTag
  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 *pAttrName/const wchar_t *pAttrName
  pAttrName is the buffer containing the name of the attribute to be created or updated.

  **Note:** The name must be xml compliant.

- const char *pAttrValue/const wchar_t *pAttrValue
  pAttrValue contains the value of the attribute that will be assigned. The value cannot be NULL. If the value is larger than 10K, we recommend writing the value as a separate blob to the pool in order to increase performance.

**Example:**
FPTag.SetStringAttribute (MyTag, "name", "value")
Error Handling: `FPPool_GetLastError()` returns `ENOERR` (zero) if successful or:

- `PP_INVALID_NAME` (program logic error)
- `PP_PARAM_ERR` (program logic error)
- `PP_TAG_READONLY_ERR` (program logic error)
- `PP_WRONG_REFERENCE_ERR` (program logic error)
- `PP_SECTION_NOT_FOUND_ERR` (internal error)
- `PP_OUT_OF_BOUNDS_ERR` (program logic error)
- `PP_CLIPCLOSED_ERR` (program logic error)
- `PP_POOLCLOSED_ERR` (program logic error)
FPTag_SetLongAttribute/FPTag_SetLongAttributeW

Syntax:
FPTag_SetLongAttribute (const FPTagRef pTag, const char *pAttrName, const FPLong pAttrValue)
FPTag_SetLongAttributeW (const FPTagRef pTag, const wchar_t *pAttrName, const FPLong pAttrValue)

Return Value:
void

Input Parameters:
const FPTagRef pTag, const char *pAttrName/const wchar_t *pAttrName, const FPLong pAttrValue

Concurrency Requirement:
This function is thread safe.

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 (pTag), the attribute name (pAttrName), and the attribute value (pAttrValue).

Note: the maximum number of attributes within one tag is 64.

Parameters:
- const FPTagRef pTag
  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 *pAttrName/const wchar_t *pAttrName
  pAttrName is a string containing the name of the attribute to be created or updated.
  Note: The name must be xml compliant.
- const FPLong pAttrValue
  pAttrValue contains the value of the attribute to be assigned.

Example:
FPLong pAttrValue;
pAttrValue = 100;
FPTag_SetLongAttribute (MyTag, "name", pAttrValue)
Error Handling:  

FP::Pool::GetLastError() returns ENOERR (zero) if successful or:

- PP_INVALID_NAME (program logic error)
- PP_PARAM_ERR (program logic error)
- PP_TAG_READONLY_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_SECTION_NOT_FOUND_ERR (internal error)
- PP_OUT_OF_BOUNDS_ERR (program logic error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
FPTag_SetBoolAttribute/FPTag_SetBoolAttributeW

Syntax:
FPTag_SetBoolAttribute (const FPTagRef pTag, const char *pAttrName, const Boolean pAttrValue)
FPTag_SetBoolAttributeW (const FPTagRef pTag, const wchar_t *pAttrName, const Boolean pAttrValue)

Return Value:
void

Input Parameters:
const FPTagRef pTag, const char *pAttrName/const wchar_t *pAttrName, const Boolean pAttrValue

Concurrency Requirement:
This function is thread safe.

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 (pTag), the attribute name (pAttrName) and the attribute value (pAttrValue).

Note: the maximum number of attributes within one tag is 64.

Parameters:

• const FPTagRef pTag
  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 *pAttrName/const wchar_t *pAttrName
  pAttrName is the buffer that will hold the name of the attribute to be created or updated.

  Note: The name must be xml compliant.

• const Boolean pAttrValue
  pAttrValue is the value of the attribute to be assigned.

Example:
Boolean pAttrValue;
pAttrValue = TRUE;
FPTag_SetBoolAttribute (MyTag, "name", pAttrValue)
Error Handling:  

FPGetLastError() returns ENOERR (zero) if successful or:

- PP_INVALID_NAME (program logic error)
- PP_PARAM_ERR (program logic error)
- PP_TAG_READONLY_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_SECTION_NOT_FOUND_ERR (internal error)
- PP_OUT_OF_BOUNDS_ERR (program logic error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
**Syntax:**
```c
FPTag_GetStringAttribute (const FPTagRef pTag, const char *pAttrName, char *pAttrValue, FPInt *pAttrValueLen)
FPTag_GetStringAttributeW (const FPTagRef pTag, const wchar_t *pAttrName, char *pAttrValue, FPInt *pAttrValueLen)
```

**Return Value:**
void

**Input Parameters:**
const FPTagRef pTag, const char *pAttrName / const wchar_t *pAttrName

**Output Parameters:**
char *pAttrValue, FPInt *pAttrValueLen

**Concurrency Requirement:**
This function is thread safe.

**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 pTag`
  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 *pAttrName / const wchar_t *pAttrName`
  `pAttrName` is the buffer containing the attribute name.

- `char *pAttrValue`
  `pAttrValue` is the buffer that will hold the attribute value. The value will be truncated to the buffer length as specified by `pAttrValueLen`.

- `FPInt *pAttrValueLen`
  Input: The length, in characters, of the buffer for `pAttrValue`.
  Output: The length of the attribute value, in characters, including the end_of_string character.

**Example:**
```c
char pAttrValue[MAX_NAME_SIZE];
namesize = MAX_NAME_SIZE;
FPTag_GetStringAttribute (MyTag, "name", pAttrValue, &namesize)
```
Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_PARAM_ERR (program logic error)
- PP_ATTR_NOT_FOUND_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_SECTION_NOT_FOUND_ERR (internal error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_TAG_READONLY_ERR (program logic error)
**FPTag_GetLongAttribute/FPTag_GetLongAttributeW**

**Syntax:**
FPTag_GetLongAttribute (const FPTagRef pTag, const char *pAttrName)
FPTag_GetLongAttributeW (const FPTagRef pTag, const wchar_t *pAttrName)

**Return Value:**
FPLong

**Input Parameters:**
const FPTagRef pTag, const char *pAttrName/const wchar_t *pAttrName

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves an FPLong attribute of an existing tag of a C-Clip.

**Parameters:**
- const FPTagRef pTag
  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 *pAttrName/const wchar_t *pAttrName
  pAttrName is the buffer containing the name of the attribute.

**Example:**
MyValue = FPTag_GetLongAttribute (MyTag, "attribute_name")

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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_GetBoolAttribute/FPTag_GetBoolAttributeW

Syntax:
FPTag_GetBoolAttribute (const FPTagRef pTag, const char *pAttrName)
FPTag_GetBoolAttributeW (const FPTagRef pTag, const wchar_t *pAttrName)

Return Value:
Boolean

Input Parameters:
const FPTagRef pTag, const char *pAttrName/const wchar_t *pAttrName

Concurrency
Requirement:
This function is thread safe.

Description:
This function retrieves a Boolean attribute of an existing tag of a C-Clip.

Parameters:
• const FPTagRef pTag
  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 *pAttrName/const wchar_t *pAttrName
  pAttrName is the buffer containing the name of the attribute.

Example:
FPTag_GetBoolAttribute (MyTag, "attribute_name")

Error Handling:
FPPool_GetLastError() returns ENOERR (zero) if successful or:
• PP_PARAM_ERR (program logic error)
• PP_ATTR_NOT_FOUND_ERR (program logic error)
• PP_WRONG_REFERENCE_ERR (program logic error)
• PP_SECTION_NOT_FOUND_ERR (internal error)
• PP_CLIPCLOSED_ERR (program logic error)
• PP_POOLCLOSED_ERR (program logic error)
• PP_TAG_READONLY_ERR (program logic error)
**FTag_RemoveAttribute/FTag_RemoveAttributeW**

**Syntax:**
- `FTag_RemoveAttribute (const FPTagRef pTag, const char *pAttrName)`
- `FTag_RemoveAttributeW (const FPTagRef pTag, const wchar_t *pAttrName)`

**Return Value:** void

**Input Parameters:**
- `const FPTagRef pTag`, `const char *pAttrName`/`const wchar_t *pAttrName`

**Concurrency Requirement:**
- This function is thread safe.

**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 pTag`
  - 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 *pAttrName`/`const wchar_t *pAttrName`
  - `pAttrName` is the buffer containing the name of the attribute that has to be removed.

**Example:**
- `FTag_RemoveAttribute (MyTag, "attribute_name")`

**Error Handling:**
- `FPPool_GetLastError()` returns ENOERR (zero) if successful or:
  - `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_GetNumAttributes

Syntax: FPTag_GetNumAttributes (const FPTagRef pTag)

Return Value: FPInt

Input Parameter: const FPTagRef pTag

Concurrency Requirement: This function is thread safe.

Description: This function returns the number of attributes in a tag.

Parameter: const FPTagRef pTag
The reference to a tag (as returned from the functions 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 (zero) if successful or:

- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_PARAM_ERR (program logic error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
FPTag_GetIndexAttribute/FPTag_GetIndexAttributeW

**Syntax:**
FPTag_GetIndexAttribute (const FPTagRef pTag, const FPInt pIndex, char *pAttrName, FPInt *pAttrNameLen, char *pAttrValue, FPInt *pAttrValueLen)
FPTag_GetIndexAttributeW (const FPTagRef pTag, const FPInt pIndex, wchar_t *pAttrName, FPInt *pAttrNameLen, wchar_t *pAttrValue, FPInt *pAttrValueLen)

**Return Value:**
void

**Input Parameters:**
const FPTagRef pTag, const FPInt pIndex, FPInt *pAttrNameLen, FPInt *pAttrValueLen

**Output Parameters:**
char *pAttrName/wchar_t *pAttrName, FPInt *pAttrNameLen, char *pAttrValue/wchar_t *pAttrValue, FPInt *pAttrValueLen

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function returns an attribute name and value of an existing tag in a C-Clip using the given index number.

**Parameters:**
- **FPTagRef pTag**
The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetParent, FPTag_GetFirstChild, FPTag_GetSibling, FPTag_GetPrevSibling, or FPClip_FetchNext).
- **FPInt pIndex**
PIndex is the index number (zero based) of the tag attribute that has to be retrieved.
- **char *pAttrName**
PAttrName is the buffer that will hold the name of the attribute. The name will be truncated to the buffer length as specified by pAttrNameLen.
- **FPInt *pAttrNameLen**
Input: The length of the buffer pAttrName. Output: The actual length of the attribute name, including the end-of-string character.
- `char *pAttrValue`
  
pAttrValue is the buffer that will hold the value of the attribute. The value will be truncated to the buffer length as specified by pAttrValueLen.

- `FPInt *pAttrValueLen`
  
  Input: The length, in characters, of the buffer for pAttrValue.
  
  Output: The actual length, in characters, of the attribute value, including the end-of-string character.

**Example:**

```c
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 (zero) if successful or:

- `PP_PARAM_ERR` (program logic error)
- `PP_ATTR_NOT_FOUND_ERR` (program logic error)
- `PP_WRONG_REFERENCE_ERR` (program logic error)
- `PP_CLIPCLOSED_ERR` (program logic error)
- `PP_POOLCLOSED_ERR` (program logic error)
Blob Handling Functions

This section describes the following functions that manipulate a blob (a tag referring to actual data):

- FPTag_BlobWrite
- FPTag_BlobRead
- FPTag_BlobReadPartial
- FPTag_BlobPurge
- FPTag_BlobExists
**FPTag_BlobWrite**

**Syntax:**

FPTag_BlobWrite (const FPTagRef pTag, const FPStreamRef pStream, const FPLong pOptions)

**Return Value:**

void

**Input Parameters:**

const FPTagRef pTag, const FPStreamRef pStream, const FPLong pOptions

**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 3-118).

This function opens a new blob, reads bytes from the stream object, writes the bytes to the pool, closes the blob, and associates the calculated Content Address (CA) with the given tag. The given parameters determine whether the 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. Be sure 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 will allow the application to perform this call if the server capability "write" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the **Centera System Operators Guide**, P/N069001126.

If collision avoidance is enabled at pool level, refer to FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13, this call will use an additional blob discriminator during write and read operations of the blob. If you want to disable collision avoidance for this call – to be compatible with a Centera v1.2 cluster – use FP_OPTION_DISABLE_COLLISION_AVOIDANCE. If collision avoidance is disabled at pool level, you can enable it for this call using FP_OPTION_ENABLE_COLLISION_AVOIDANCE.
If this function is used to restore data from a stream to the pool, the data will only be exported to the pool if \texttt{pTag} already has data associated to 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 \texttt{prepareBufferProc} callback should not take longer than 1 minute to execute.

The following applies to SDK 2.1 and higher: If the generic stream returns more data than asked, the SDK will enlarge the buffer to accommodate the additional data. If the stream cannot provide the data within 1 minute, set \texttt{mTransferLen} to -1 and \texttt{mAtEOF} to false. This forces \texttt{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 will only work 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:**

- \texttt{const FPTagRef pTag}
  The reference to a tag (as returned from the functions \texttt{FPTag_Create}, \texttt{FPTag_GetFirstChild}, \texttt{FPTag_GetSibling}, \texttt{FPTag_GetPrevSibling}, \texttt{FPClip_FetchNext} and \texttt{FPTag_GetParent}).

- \texttt{const FPStreamRef pStream}
  The reference to an input stream (as returned from the \texttt{FPStream_CreateXXX} functions or \texttt{FPStream_CreateGenericStream}), refer to *Stream Functions* on page 3-118.

- \texttt{const FPLong pOptions}
  A numeric code that identifies how the Content Address should be calculated. You must use one of the following options:

  - \texttt{FP_OPTION_CLIENT_CALCID} – The client calculates the address before sending the data to the cluster. The client will not send the data, if the cluster already contains the data.
Note: When writing large files (10 Mbytes or larger) or when using many threads, we recommend not to calculate Content Addresses before writing to the cluster for performance reasons.

- **FP_OPTION_CLIENT_CALCID_STREAMING** – The client calculates the address while sending the data to the server. This option is equal to **FP_OPTION_DEFAULT_OPTION**.
- **FP_OPTION_SERVER_CALCID_STREAMING** – The server calculates the address (as the application server sends the data). This option has been deprecated since SDK 2.1.

The following options may be used in conjunction with any of the options above:

- **FP_OPTION_CALCID_NOCHECK** – There is no end-to-end checking of the Content Address on the client and the server to verify that it matches the expected address. Using this option increases performance. Be aware that you will not know if the server has received the data correctly. This option has been deprecated since SDK 2.1, the CA of all blobs will be checked.
- **FP_OPTION_ENABLE_DUPLICATE_DETECTION** – This option only applies to SDK versions prior to 2.1. If this option is true, the system checks, when writing data to the pool, if the new Content Address already exists for data stored on the cluster. If the system finds the same CA, it will return the error **FP_DUPLICATE_FILE_ERR** (-10010). This option only checks if the calculated CA already exists, not if the CA refers to the same data. Use this option if you want to be sure that the system never calculates the same CA for different data. When Collision Avoidance is enabled there will be no duplicate CA’s by default.

Note: When a colliding CA is found for a segment of a large file (more than 100 Mbyte) using **FP_OPTION_ENABLE_DUPLICATE_DETECTION**, the whole file will be considered as a duplicate.

- **FP_OPTION_ENABLE_COLLISION_AVOIDANCE** – If collision avoidance is disabled at pool level, you can enable it using this option. This option is only valid for this instance of FPTag_BlobWrite. For more information refer to the Centera Programmer’s Guide, P/N069001127, and FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13.
• **FP_OPTION_DISABLE_COLLISION_AVOIDANCE** – If collision avoidance is enabled at pool level, you can disable it with this option. This option is only valid for this instance of FPTag_BlobWrite. For more information refer to the Centera Programmer’s Guide, P/N069001127, and FPPool_SetIntOption/FPPool_SetIntOptionW on page 3-13.

**Example:**

The following example calculates the Content Address on the client before sending the data to the cluster.

FPTag_BlobWrite (MyTag, MyStream, FP_OPTION_CLIENT_CALCID)

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- **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_BlobRead

Syntax:
FPTag_BlobRead (const FPTagRef pTag, const FPStreamRef pStream, const FPLong pOptions)

Return Value:
void

Input Parameters:
const FPTagRef pTag, const FPStreamRef pStream, const FPLong pOptions

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 3-118).

Note: Refer to Stream Creation Functions on page 3-121 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 true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.
Parameters:
- const FPTagRef pTag
  The reference to a tag (as returned from the functions
  FPTag_GetFirstChild, FPTag_GetSibling,
  FPTag_GetPrevSibling, FPClip_FetchNext and
  FPTag_GetParent).
- const FPStreamRef pStream
  The reference to a stream (as returned from the functions
  FPStream_CreateXXX or FPStream_CreateGenericStream).
- const FPLong pOptions
  A numeric code that identifies the option to be used. You can use
  the following options:
  - PP_OPTION_DEFAULT_OPTIONS – The default is to verify the data
    read from the Centera server.
  - PP_OPTION_CALC_ID_NOCHECK – The client does not calculate a
    Content Address (CA) for the data to verify that it matches the
    expected CA. Using this option increases performance. Be aware
    that you will not know if the server has sent the data correctly.
    This option has been deprecated since SDK 2.1, the CA of all
    blobs will be checked.

Example:
FPTag_BlobRead (MyTag, MyStream, 0)

Error Handling:
FPPool_GetLastErr () returns ENOERR (zero) if successful or:
- PP_PARAM_ERR (program logic error or internal error; verify your
  code before contacting the EMC Customer Support Center)
- PP_ATTR_NOT_FOUND_ERR (internal error)
- PP_TAG_HAS_NO_DATA_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_FILE_NOT_STORED_ERR (program logic error)
- PP_NO_POOL_ERR (network error)
- PP_NO_SOCKET_AVAIL_ERR (network error)
- PP_PROTOCOL_ERR (internal error)
- PP_UNKNOWN_OPTION (internal error)
- PP_SERVER_ERR (server error)
- PP_CONTROLFIELD_ERR (server error)
- PP_NOT_RECEIVE_REPLY_ERR (network error)
- PP_SEGDATA_ERR (internal error)
- PP_BLOBIDMISMATCH_ERR (server error)
- PP_BLOBBUSY_ERR (server error)
- PP_SERVER_NOTREADY_ERR (server error)
- PP_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 pTag, const FPStreamRef pStream, const FPLong pOffset, const FPLong pReadLength, const FPLong pOptions)

Return Value:  
void

Input Parameters:  
const FPTagRef pTag, const FPStreamRef pStream, const FPLong pOffset, const FPLong pReadLength, const FPLong pOptions

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 3-118). This function reads the data in chunks of 16 Kbyte.

Note:  
The server allows the application to perform this call if the server capability “read” is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

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 will be added to the output stream and the function will return ENOERR. Use FPTag_GetBlobSize() to verify that you are not reading past the end of the blob.

Refer to Stream Creation Functions on page 3-121 for more information on how a stream is opened.

Note:  
Due to server limitations, the stream should not exceed 1 minute per iteration to store the data received from the server. This means the completeProc callback should not take longer than 1 minute to execute.
Parameters:

- `const FPTagRef pTag`  
The reference to a tag (as returned from the functions `FPTag_GetFirstChild`, `FPTag_GetSibling`, `FPTag_GetPrevSibling`, `FPClip_FetchNext` and `FPTag_GetParent`).

- `const FPStreamRef pStream`  
The reference to a stream (as returned from the functions `FPStream_CreateXXX` or `FPStream_CreateGenericStream`).

- `const FPLong pOffset`  
The starting offset of the read operation.

- `const FPLong pReadLength`  
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 pOptions`  
A numeric code that identifies the option to be used. The following options can be used:

  - `FP_OPTION_DEFAULT_OPTIONS` – The default is to verify the data read from the Centera server if `pOffset = 0` and `pLength = -1`.
  
  - `FP_OPTION_CALC_ID_NOCHECK` – The client does not calculate a Content Address (CA) of the data read/written to verify that it matches the expected CA. Using this option increases performance. Be aware that you will not know if the server has sent the data correctly. This option has been deprecated since SDK 2.1. The CA of all blobs are checked if `pOffset = 0` and `pLength = -1`.

Example:

Read 8 Kbytes of the blob, starting at the first byte.

```c
FPTag_BlobReadPartial(MyTag, MyStream, 0, 8192, FP_OPTION_DEFAULT_OPTIONS);
```

Read 8 Kbytes of the blob, starting at offset 32 Kbytes.

```c
FPTag_BlobReadPartial(MyTag, MyStream, 32768, 8192, FP_OPTION_DEFAULT_OPTIONS);
```

Read the entire blob, equivalent to `FPTag_BlobRead(MyTag, MyStream, 0)`.

```c
FPTag_BlobReadPartial(MyTag, MyStream, 0, -1, FP_OPTION_DEFAULT_OPTIONS);
```

Error Handling:

`FPool_GetLastErr()` returns `ENOERR` (zero) if successful or:

- `FP_PARAM_ERR` (program logic error or internal error; verify your
Blob Handling Functions

- PP_ATTR_NOT_FOUND_ERR (internal error)
- PP_TAG_HAS_NO_DATA_ERR (program logic error)
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_FILE_NOT_STORED_ERR (program logic error)
- PP_NO_POOL_ERR (network error)
- PP_NO_SOCKET_AVAIL_ERR (network error)
- PP_PROTOCOL_ERR (internal error)
- PP_UNKNOWN_OPTION (internal error)
- PP_SERVER_ERR (server error)
- PP_CONTROLFIELD_ERR (server error)
- PP_NOT_RECEIVE_REPLY_ERR (network error)
- PP_SEGDATA_ERR (internal error)
- PP_BLOBIDMISMATCH_ERR (server error)
- PP_BLOBBUSY_ERR (server error)
- PP_SERVER_NOTREADY_ERR (server error)
- PP_PROBEPACKET_ERR (internal error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_OPERATION_NOT_ALLOWED (client error)
### FPTag_BlobPurge

**Syntax:**
```
FPTag_BlobPurge (FPTagRef pTag)
```

**Return Value:**
```
void
```

**Input Parameter:**
```
FPTagRef pTag
```

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function purges a blob from the given tag in an opened C-Clip and removes the reference to the blob from the C-Clip. The tag itself and the C-Clip containing that tag will not be deleted. Once a blob has been purged it does not exist anymore and cannot be retrieved. 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 will allow the application to perform this call if the server capability "purge" is true. A Compliance Plus model will not allow this call and will return the error `FP_OPERATION_NOT_ALLOWED`.

Refer to `FPPool_GetCapability/FPPool_GetCapabilityW` on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

**Note:** If you purge all blobs from a C-Clip, the C-Clip will still exist and can be queried without errors. Errors will occur if an application tries to read a blob that no longer exists. The application developer is responsible for writing code that can handle this scenario.

**Parameter:**
```
FPTagRef pTag
```

The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetFirstChild, FPTag_GetSibling, FPTag_GetPrevSibling, FPClip_FetchNext or FPTag_GetParent).

**Example:**
```
FPTag_BlobPurge (MyTag)
```

**Error Handling:**
`FPPool_GetLastErr()` returns `ENOERR` (zero) if successful or:
- `FP_NO_POOL_ERR` (network error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
• PP_NO_SOCKET_AVAIL_ERR (network error)
• PP_PARAM_ERR (internal error)
• PP_PROTOCOL_ERR (internal error)
• PP_WRONG_REFERENCE_ERR (program logic error)
• PP_SERVER_ERR (server error)
• PP_CONTROLFIELD_ERR (server error)
• PP_NUMLOC_FIELD_ERR (server error)
• PP_BLOBBUSY_ERR (server error)
• PP_SERVER_NOTREADY_ERR (server error)
• PP_NOT_RECEIVE_REPLY_ERR (network error)
• PP_BLOBIDMISMATCH_ERR (server error)
• PP_PROBEPACKET_ERR (internal error)
• PP_CLIPCLOSED_ERR (program logic error)
• PP_POOLCLOSED_ERR (program logic error)
• PP_OPERATION_NOT_ALLOWED (client error)
• PP_TRANSACTION_FAILED_ERR (server error)
FPTag_BlobExists

Syntax:  FPTag_BlobExists (const FPTagRef pTag)

Return Value:  FPInt

Input Parameter:  const FPTagRef pTag

Concurrency Requirement:  This function is thread safe.

Description:  This function checks if the blobdata of the given tag exists. If the blobdata of the given tag exists, the function returns 1. If the blobdata is segmented then all segments must exist. If the blobdata does not exist, the function returns 0. If the tag has no associated blobdata, the function returns -1.

This function will not failover.

Note: The server will allow the application to perform this call if the server capability "exist" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

Parameter:  FPTagRef pTag
The reference to a tag (as returned from the functions FPTag_Create, FPTag_GetFirstChild, FPTag_GetSibling, FPTag_GetPrevSibling, FPClip_FetchNext or FPTag_GetParent).

Example:  FPTag_BlobExists (MyTag)

Error Handling:  FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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)
- PP_NOT_RECEIVE_REPLY_ERR (network error)
- PP_SEGDATA_ERR (internal error)
- PP_BLOBIDMISMATCH_ERR (server error)
- PP_BLOBBUSY_ERR (server error)
- PP_SERVER_NOTREADY_ERR (server error)
- PP_PROBEPACKET_ERR (internal error)
- PP_CLIPCLOSED_ERR (program logic error)
- PP_TAGCLOSED_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_OPERATION_NOT_ALLOWED (client error)
- PP_WRONG_STREAM_ERR (client error)
Stream Functions

Streams are generalized input/output channels similar to the HANDLE mechanism used in the Win32 API for files, the C++ iostream functionality, the C standard I/O FILE routines, or the Java InputStream/OutputStream facility.

All bulk data movement in the API is achieved by first associating a stream with a data sink or source (for example, a file, an in-memory buffer, or the standard output channel) and then performing operations on that stream.

The API stream functions implement streams generically. When creating a stream, the application developer has to use method pointers to indicate how subsequent stream functions should handle the stream. For more information on the available method pointers for stream functions, refer to FPStream_CreateGenericStream on page 3-122.

Stackable Stream Support

A stackable stream is a stream that calls another stream as part of its operation. An example is a stream that prepends bytes to another stream, or a stream that compresses stream data. To support stackable streams, the following functions are available to the API: FPStream_PrepareBuffer, FPStream_Complete, FPStream_SetMark, and FPStream_ResetMark.

Generic Stream Operation

In versions of the SDK prior to v1.2, generic streams that were used for output (for example with FPTag_BlobRead) received a buffer with data which the completion callback function had to process. For SDK v1.2 and higher, the application has the possibility to provide a buffer to the output stream which the SDK will fill. In this case, the stream remains the owner of the output buffer.

Foreign Pointer Mode

The SDK remains the owner of the pointer to the data buffer (mBuffer). The stream completion function is only allowed to read data from it. This mode is compatible with pre 1.2 SDK releases.
The following rules apply:

- Do not declare `prepareBufferProc`. If declared, `mBuffer` should be set to `NULL`.
- This function `completeProc` can read the data from `mBuffer` for `mTransferLen` bytes but it is not allowed to change these fields in the `StreamInfo` structure.

If the end of the stream has been reached, then `mAtEOF` is true. It is possible that this last call to the `completeProc` does not pass any data (`mTransferLen` is 0, and `mBuffer` remains `NULL`).

### Stream Buffer Mode

The stream provides a buffer that the SDK will fill. `prepareBufferProc` is responsible for preparing this buffer and to set `mBuffer` to it. `mTransferLen` contains the number of bytes that can be transferred. `completeProc` will process this buffer. `mTransferLen` then contains the number of bytes actually transferred into the buffer. The value of `mBuffer` does not change.

The following rules apply:

- `prepareBufferProc` should set `mBuffer` to point to the buffer that it manages and it should set `mTransferLen` to the maximum number of bytes that the stream can receive (typically `mTransferLen` is the length of `mBuffer`).
- `completeProc` can process the data in `mBuffer`. `mTransferLen` number of bytes are actually copied into `mBuffer`.

The algorithm is as follows:

```c
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
```

*Stream Functions*
If the end of the stream has been reached, then `mAtEOF` is true. It is possible that this last call to `completeProc` does not pass any data (`mTransferLen` is 0).
Stream Creation Functions

This section describes the following functions used to create a stream:

- PPStream_CreateGenericStream
- PPStream_CreateFileForInput
- PPStream_CreateFileForOutput
- PPStream_CreateWithFile
- PPStream_CreateBufferForInput
- PPStream_CreateBufferForOutput
- PPStream_CreateWithBuffer
- PPStream_CreateToStdio
- PPStream_CreateToNull
- PPStream_CreateTemporaryFile
FPStream_CreateGenericStream

**Syntax:**


**Return Value:**

FPStreamRef

**Input Parameters:**


**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function allocates a stream data structure, declares its methods and returns a reference to the created stream. This function returns NULL, if the stream has not been created.

If you want to extend the created Generic Stream, you will have to supply at least one or more function pointers.

**Parameters:**

const void *pUserData

Any data that the application wants to pass to the method pointers.

The callback functions are of type FPStreamProc:

```c
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 has to ensure that the callback function is thread safe and does not take more than 1 minute to execute.

In the following callback function descriptions, **input stream** refers to a stream from which the SDK has to read data, for example when writing a blob to the cluster. **Output stream** refers to a stream to which the SDK has to write data, for example when reading a blob from the cluster.
**FPStreamProc pPrepareBufferProc**

This method prepares a buffer that the stream can use. If the stream is an input buffer, the callback method prepares a buffer that contains the data. The mBuffer field of FPStreamInfo contains a pointer to the data. If the stream is an output buffer, the function does nothing.

If you name your function MyPrepareBuffer then you would declare it like this:

```c
static long MyPrepareBuffer (FPStreamInfo *pStreamInfo)
```

pStreamInfo is a pointer to an FPStreamInfo structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 3-5, **FPStreamInfo fields for MyPrepareBuffer** for a description of all possible fields of FPStreamInfo.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mVersion</td>
<td>short</td>
<td>Version of the FPStreamInfo structure. Currently the value of this field is 3. Check this field for backward compatibility.</td>
</tr>
<tr>
<td>mUserData</td>
<td>void*</td>
<td>Parameter passed unchanged from FPStream_CreateGenericStream to each callback function. You can use this to pass (a pointer to) MyStream-specific data to the callback instead of relying on global variables.</td>
</tr>
<tr>
<td>mStreamPos</td>
<td>FPLong</td>
<td>The current position in the stream where input occurs. This field should be updated by MyPrepareBuffer if needed.</td>
</tr>
<tr>
<td>mStreamLen</td>
<td>FPLong</td>
<td>The total length of the stream. This is –1 (unknown) by default and should be updated by MyPrepareBuffer. Preferably already initialized before the first call to prepareBuffer.</td>
</tr>
<tr>
<td>mAEEOF</td>
<td>Boolean</td>
<td>Indicates if MyStream has reached the end of stream or not. MyPrepareBuffer should return true in this field if the last segment is read.</td>
</tr>
</tbody>
</table>
The SDK calls MyPrepareBuffer prior to transferring data from the input stream to the Centera server. The SDK expects the callback function to make the data available and provides a pointer to it in mBuffer. The pointer to this data will be 'owned' by the SDK until the completeProc is called. The prepareBufferProc is also called before data is transferred from the Centera server to the SDK.

MyPrepareBuffer could then be used to prepare the output stream to receive data. In many cases however, it is not necessary to implement this function for output streams (pass NULL in FPStream_CreateGenericStream()). If used for output streams,

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mReadFlag</td>
<td>Boolean</td>
<td>Indicates if MyStream is used for reading (input) or writing (output). The behavior of MyPrepareBuffer might depend on the value of this field. In case of an input stream: Must be set to true. In case of an output stream: Must be set to false.</td>
</tr>
<tr>
<td>mBuffer</td>
<td>void*</td>
<td>In case of an input stream: MyPrepareBuffer should make mBuffer point to a buffer containing data. Possibly MyPrepareBuffer must allocate memory, read data from a device into that memory, and set mBuffer to point to it. In case of an output stream: This field should either be set to a stream-managed buffer or it should be set to NULL.</td>
</tr>
<tr>
<td>mTransferLen</td>
<td>FPLong</td>
<td>In case of an input stream: mTransferLen indicates the maximum number of bytes that the SDK wants to receive from the input. MyPrepareBuffer returns the actual number of bytes in mBuffer in this field. If the buffer contains no data, then mTransferLen = 0. If mTransferLen = 0 and mAtEOF is true, then the end of the data stream has been reached. In case of an output stream: If MyPrepareBuffer manages the output buffer, then mTransferLen should be set to the size of that output buffer. The SDK will transfer the number of bytes into the output buffer that equals its size. If the stream does not manage the output buffer, then this field is unused.</td>
</tr>
</tbody>
</table>
MyPrepareBuffer can put a pointer to the buffer that it manages in mBuffer and its length in mTransferLen.

**FPStreamProc pCompleteProc**
The generic stream calls this method when the buffer that has been prepared with pPrepareBufferProc is no longer needed. If the stream was an input stream, this means the data has been processed successfully. If the stream was an output stream, this means the buffer contains the requested data and it can be written to an output device.

If you name your function MyComplete then you would declare it like this:

```c
static long MyComplete (FPStreamInfo *pStreamInfo)
```

pStreamInfo is a pointer to an FPStreamInfo structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 3-6, FPStreamInfo fields for MyComplete for a description of all possible fields of FPStreamInfo.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mVersion</td>
<td>short</td>
<td>Version of the FPStreamInfo structure. Currently the value of this field is 3. Check this field for backward compatibility.</td>
</tr>
<tr>
<td>mUserData</td>
<td>void*</td>
<td>Parameter passed unchanged from FPStream_CreateGenericStream to each callback function. You can use this to pass (a pointer to) MyStream-specific data to the callback instead of relying on global variables.</td>
</tr>
<tr>
<td>mStreamPos</td>
<td>FPLong</td>
<td>The current position in the stream where input or output occurs. This field should be updated by MyComplete if needed. This should preferably already have been initialized when MyStream is created.</td>
</tr>
<tr>
<td>mStreamLen</td>
<td>FPLong</td>
<td>The total length of the stream. This is –1 (unknown) by default and should be updated by MyComplete. Preferably already initialized when MyStream is created.</td>
</tr>
<tr>
<td>mAtEOF</td>
<td>Boolean</td>
<td>In case of an output stream: It indicates that the last buffer of data will now be written. If mTransferLen = 0, then no data will actually be passed.</td>
</tr>
</tbody>
</table>
In case of an output stream, MyComplete actually transfers the data pointed to by mBuffer to the output device. In case of a file, this might translate into a fwrite() operation. The mBuffer pointer can either be provided by the stream or by the SDK.

Note: If the SDK provides the mBuffer pointer to the stream to read the output data, the callback function should never change this data.

In case of an input stream, the callback function notifies that the SDK has finished with the input buffer (mBuffer). It can then unlock or deallocate the buffer if necessary. In many cases, this callback is NULL for input streams.

In case of an output stream, the end of the stream behaves as follows:

- If the application allocated its buffer in prepareBufferProc, then the last completeProc callback has mTransferLen >= 0 and mAtEOF is true.
- If the application did not allocate its buffer, then the last call to completeProc always has mTransferLen = 0 and mAtEOF is true.

### Table 3-6 FPStreamInfo fields for MyComplete

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mReadFlag</td>
<td>Boolean</td>
<td>Indicates if MyStream is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. The behavior of MyComplete might depend on the value of this field.</td>
</tr>
<tr>
<td>mBuffer</td>
<td>void*</td>
<td>In case of an input stream: Contains a pointer to the data that has been read (as provided by an earlier call from MyPrepareBuffer). In case of an output stream: Contains the address of a memory buffer that holds the data that has to be written. If this pointer is owned by the SDK (refer to prepareBufferProc) then the callback function is allowed to access this memory during the execution of MyComplete only.</td>
</tr>
<tr>
<td>mTransferLen</td>
<td>FPLong</td>
<td>In case of an input stream: Contains the number of bytes that have been read (as provided by an earlier call from MyPrepareBuffer). In case of an output stream: Contains the number of bytes that mBuffer points to. This value can be 0 (in which case mAtEOF is true).</td>
</tr>
</tbody>
</table>
**FPStreamProc pSetMarkerProc**

This method instructs the generic stream to mark the current position in the stream. If the stream supports marking, the function can use the mMarkerPos field to indicate the current position.

If you name your function `MySetMarker` then you would declare it like this:

```c
static long MySetMarker (FPStreamInfo *pStreamInfo)
```

`pStreamInfo` is a pointer to an `FPStreamInfo` structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 3-7, `FPStreamInfo fields for MySetMarker` for a description of all possible fields of `FPStreamInfo`.

### Table 3-7 FPStreamInfo fields for MySetMarker

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mVersion</td>
<td>short</td>
<td>Version of the <code>FPStreamInfo</code> structure. Currently the value of this field is 3. Check this field for backward compatibility.</td>
</tr>
<tr>
<td>mUserData</td>
<td>void*</td>
<td>Parameter passed unchanged from <code>FPStream_CreateGenericStream</code> to each callback function. You can use this to pass a pointer to <code>MyStream</code>-specific data to the callback instead of relying on global variables.</td>
</tr>
<tr>
<td>mStreamPos</td>
<td>FPLong</td>
<td>The current position in the stream where input or output occurs. This field should be updated by <code>MySetMarker</code> to reflect the current position in the stream. In case of a file-based stream, this is often a <code>ftell()</code> call.</td>
</tr>
<tr>
<td>mReadFlag</td>
<td>Boolean</td>
<td>Indicates if <code>MyStream</code> is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. This callback function usually does not use this field.</td>
</tr>
<tr>
<td>mMarkerPos</td>
<td>FPLong</td>
<td>This field is updated by GenericStreams to reflect <code>mStreamPos</code> after the callback function returns.</td>
</tr>
</tbody>
</table>

The SDK sometimes needs to return to an earlier position in the stream. To do this, it sets a ‘mark’ at the current position in the stream (how the current position is defined depends on the stream implementation). If necessary, the SDK can return later to that position by calling the `resetMarkerProc`. 
If a stream does not support marking, then pass NULL in 
`FPStream_CreateGenericStream`.

**Note:** This callback function returns an error if unsuccessful or `ENOERR` if successful. Any error from `MySetMarker` is currently ignored by the SDK as it is not a fatal condition.

**FPStreamProc pResetMarkerProc**
This method tells the stream to go back to the marked position in the stream (`mMarkerPos`).

If you name your function `MyResetMarker` then you would declare it like this:

```c
static long MyResetMarker (FPStreamInfo *pStreamInfo)
```

`pStreamInfo` is a pointer to an `FPStreamInfo` structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 3-8, `FPStreamInfo fields for MyResetMarker` for a description of all possible fields of `FPStreamInfo`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mVersion</td>
<td>short</td>
<td>Version of the <code>FPStreamInfo</code> structure. Currently the value of this field is 3. Check this field for backward compatibility.</td>
</tr>
<tr>
<td>mUserData</td>
<td>void*</td>
<td>Parameter passed unchanged from <code>FPStream_CreateGenericStream</code> 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.</td>
</tr>
</tbody>
</table>
When the SDK needs to return to an earlier position in the stream (indicated by calling `setMarkerProc`), it calls `resetMarkerProc`.

If a stream does not support marking, then pass `NULL` in `FPStream_CreateGenericStream`. When the SDK needs this functionality, it exits with an `FP_OPERATION_REQUIRES_MARK` error.

In some cases, the stream cannot return to an arbitrary position in the stream, but can only go back to the beginning of it (for example if the data needs to be calculated). If the callback returns `FP_OPERATION_REQUIRES_MARK` in the first case, the SDK will retry with `mMarkerPos 0`. The stream can then reset itself and the SDK will retry the complete operation.

### FPStreamProc pCloseProc

This method informs that the application has performed its operations on the stream and that the stream can clean up the resources that it has allocated.

If you name your function `MyClose` then you would declare it like this:

```c
static long MyClose (FPStreamInfo *pStreamInfo)
```

This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: for input or for output. Refer to Table 3-9, `FPStreamInfo fields for MyClose` for a description of all possible fields of `FPStreamInfo`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| mStreamPos   | FPLong   | The current position in the stream where input or output occurs. This field is updated by GenericStreams to the value of `mMarkerPos` after the call to `MyResetMarker`.
| mReadFlag    | Boolean  | Indicates if `MyStream` is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. This callback function does not usually use this field. |
| mMarkerPos   | FPLong   | If the stream implementation uses an offset field to remember the marked position, then `mMarkerPos` contains the position of the mark. `MyResetMarker` should set the stream to that position. In case of file-based streams, `MyResetMarker` should do a `fseek()` on `mMarkerPos`. |
When the SDK has finished its operations on a stream, it calls the `closeProc` callback function. `MyClose` then has the chance to close any opened resources (for example a file or a network socket) and to deallocate any memory if necessary. If no close function is needed, then `NULL` can be passed to `FPStream_CreateGenericStream`.

**Example:**

The sample code shows how to build a stream implementation on top of Generic Streams. Error handling, parameter checking and special cases are omitted for clarity. Refer to *Multiple Clusters and Write Failover* in the *Centera Programmer’s Guide*, P/N 069001127, for another example.

**File Input**

This example creates a File Input stream. The parameters are a path to the file, the file permissions and the buffersize. A memory buffer will be allocated to read the file data into. 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.

```c
EXPORT FPStreamRef FPStream_CreateFileForInput (const char *pFilePath, const char *pPerm, const long pBuffSize)
{
  TStreamFileInfo *vFileInfo = NULL;
  char            *vBuffer    = NULL;
  FPStreamRef      vResult    = NULL;

  vFileInfo = (TStreamFileInfo*) calloc (1, sizeof (TStreamFileInfo)) ;
  vBuffer    = (char*) malloc (pBuffSize) ;
  vFileInfo->mBufferLen = pBuffSize ;

  // build the access rights from the pPerm parameter here ..
```

### Table 3-9 FPStreamInfo fields for MyClose

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mVersion</td>
<td>short</td>
<td>Version of the FPStreamInfo structure. Currently the value of this field is 3. Check this field for backward compatibility.</td>
</tr>
<tr>
<td>mUserData</td>
<td>void*</td>
<td>Parameter passed unchanged from <code>FPStream_CreateGenericStream</code> 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.</td>
</tr>
</tbody>
</table>
vFileInfo->mFile = CreateFile (pFilePath, 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 ;
This method **deallocates the buffer** (allocated in the stream creation function) and closes the file.

```c
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.

```c
EXPORT FPStreamRef FPStream_CreateFileForOutput (const char *pFilePath, const char *pPerm)
{
    TStreamFileInfo *vFileInfo = NULL ;
    FPStreamRef vResult ;
    vFileInfo = (TStreamFileInfo*) calloc (1, sizeof (TStreamFileInfo)) ;
    // get the access rights from the pPerm parameter..
    vFileInfo->mFile = CreateFile (pFilePath, 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.

```c
static long fileOutComplete (FPStreamInfo *pStreamInfo)
```
C API Reference

Stream Creation Functions

{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData ;
unsigned long    l ;

if (WriteFile (vFileInfo->mFile, pStreamInfo->mBuffer,
   pStreamInfo->mTransferLen, &l, NULL) == 0)
{ return FP_OPERATION_NOT_SUPPORTED1 ;
}
pStreamInfo->mTransferLen = l ;
pStreamInfo->mStreamPos  += l ;

return ENOERR ;
}

Error Handling:  FPPool_GetLastError () returns ENOERR (zero) if successful.

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

Syntax: FPStream_CreateFileForInput (const char *pFilePath, const char *pPerm, const long pBuffSize)

Return Value: FPStreamRef

Input Parameters: const char *pFilePath, const char *pPerm, const long pBuffSize

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 (pPerm), you can use the stream with most of the stream handling functions (refer to Page 3-145 for more information).

Parameters:
- const char *pFilePath
  pFilePath is the buffer that contains the path name of the file for which the stream must be created.

- const char *pPerm
  pPerm is the buffer that contains the open permission for the file. You can only 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 will return an error.

- const long pBuffSize
  pBuffSize is the size of the buffer (in bytes) that is used when reading the file. The value has to be greater than 0.

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

```c
{ 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 (zero) if successful or:

- `PP_PARAM_ERR` (program logic error)
- `PP_FILESYS_ERR` (program logic error)
FPStream_CreateFileForOutput

Syntax:   FPStream_CreateFileForOutput (const char *pFilePath, 
          const char *pPerm)

Return Value:   FPStreamRef

Input Parameters:   const char *pFilePath, const char *pPerm

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 (pPerm), you can use the stream with all stream handling functions (refer to Page 3-145 for more information).

Parameters:   

- const char *pFilePath
  pFilePath is the buffer that contains the pathname of the file for which the stream must be created.

- const char *pPerm
  pPerm is the buffer that contains the open permission for the file, for writing this is usually wb. You can use one of the following permissions:

  wb: opens the given file for writing. This function overwrites existing content of the file.

  ab: opens the file for writing at the end of the file. If the file does not exist, the function creates the file.

  rb+: opens the given file for both reading and writing. If the file does not exist or the system cannot find the file, the function returns an error.

  wb+: opens the given file for both reading and writing. If the given file exists, the function overwrites the content.

  ab+: opens the given file for reading and writing at the end of the file. If the file does not exist, the function creates the file.
Example: This example shows how to write data from a pool to a stream.

```c
FPStreamRef vStream = FPStream_CreateFileForOutput (pPath, "wb") ;
// create a new binary file for write
if (vStream == 0)
    return FPPool_GetLastError() ;
FPTag_BlobRead (pTag, vStream, 0) ;
// read stream has highest performance for downloading
FPStream_Close (vStream) ;
// close file stream
}
```

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_PARAM_ERR (program logic error)
- PP_FILESYS_ERR (program logic error)
Earlier versions of the Centera Access API provided this function to create a stream that handles output to/input from a file. Although this function is still available, we recommend using the `FPStream>CreateFileForInput` and `FPStream>CreateFileForOutput` functions to create streams that write to or read from a file.

**Note:** The write permissions `a`, `a+`, `ab` and `ab+` (that were supported in earlier versions of the API) for this function, are no longer supported.
FPStream_CreateBufferForInput

**Syntax:**
```
FPStream_CreateBufferForInput (char *pBuffer, const unsigned long pBuffLen)
```

**Return Value:**
FPStreamRef

**Input Parameters:**
char *pBuffer, const unsigned long pBuffLen

**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 *pBuffer
  pBuffer is the memory buffer containing the data for the stream.
- const unsigned long pBuffLen
  pBuffLen is the buffer length (in bytes) of pBuffer.

**Example:**
```
char MyDataSource[A_BIG_SIZE];
MyStream = FPStream_CreateBufferForInput (MyDataSource, A_BIG_SIZE);
```

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_PARAM_ERR (program logic error)
FPStream_CreateBufferForOutput

**Syntax:**
FPStream_CreateBufferForOutput (char *pBuffer, const unsigned long pBuffLen)

**Return Value:**
FPStreamRef

**Input Parameters:**
const char *pBuffer, const unsigned long pBuffLen

**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 3-122 for more information on pStreamInfo.

**Parameters:**
- const char *pBuffer  
  pBuffer is the memory buffer containing the data for the stream.
- const unsigned long pBuffLen  
  pBuffLen is the buffer length (in bytes) of pBuffer.

**Example:**
char MyDataSource[A_BIG_SIZE];  
MyStream = FPStream_CreateBufferForOutput (MyDataSource, A_BIG_SIZE);

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- FP_PARAM_ERR (program logic error)
FPStream_CreateWithBuffer

Earlier versions of the Centera Access API provided this function to create a stream that handles input from a buffer. Although this function is still available, we recommend using the FPStream_CreateBufferForInput function to create a stream that reads from a buffer.
**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.

**Parameter:**
void

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful.
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.

Parameter: void

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful.
FPStream_CreateTemporaryFile

Syntax: FPStream_CreateTemporaryFile (const long pMemBuffSize)

Return Value: FPStreamRef

Input Parameter: const long pMemBuffSize

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 pMemBuffSize, the overflow is flushed to a temporary file in the platform-specific temporary directory. This temporary file will be automatically deleted when the stream closes.

Parameter: const long pMemBuffSize
pMemBuffSize 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 (zero) if successful or:

- FP_PARAM_ERR (program logic error)
Stream Handling Functions

This section describes the following functions that handle a stream:

- PPStream_Read
- PPStream_Write
- PPStream_SetMarker
- PPStream_GetMarker
- PPStream_PrepareBuffer
- PPStream_Complete
- PPStream_SetMark
- PPStream_ResetMark
- PPStream_GetInfo
- PPStream_Close
**FPStream_Read**

Earlier versions of the Centera Access API provided this function to read from a stream. This functionality is supported for compatibility reasons only and will be removed from later versions of the SDK.

**FPStream_Write**

Earlier versions of the Centera Access API provided this function to write from a stream. This functionality is supported for compatibility reasons only and will be removed from later versions of the SDK.

**FPStream_SetMarker**

Earlier versions of the Centera Access API provided this function to mark a position in a stream. This functionality is supported for compatibility reasons only and will be removed from later versions of the SDK.

**FPStream_GetMarker**

Earlier versions of the Centera Access API provided this function to return to a marked position in a stream. This functionality is supported for compatibility reasons only and will be removed from later versions of the SDK.
FPStream_PrepareBuffer

Syntax: FPStream_PrepareBuffer (const FPStreamRef pStream)

Return Value: FPStreamInfo*

Input Parameters: const FPStreamRef pStream

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.

Parameter: const FPStreamRef pStream
The reference to a stream as created by FPStream_CreateGenericStream.

Example: Refer to Storing an Encrypted File in the Centera Programmer’s Guide, P/N 069001127, for a code example.

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- Any error that is returned by the prepareBufferProc callback function.
### FPStream_Complete

**Syntax:**

```c
FPStream_Complete (const FPStreamRef pStream)
```

**Return Value:**

`FPStreamInfo*`

**Input Parameters:**

`const FPStreamRef pStream`

**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`.

**Parameter:**

`const FPStreamRef pStream`

The reference to a stream as created by `FPStream_CreateGenericStream`.

**Example:**

Refer to Storing an Encrypted File in the Centera Programmer's Guide, P/N 069001127, for a code example.

**Error Handling:**

`FPPool_GetLastError()` returns `ENOERR` (zero) if successful or:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- Any error that is returned by the `completeProc` callback function.
**FPStream_SetMark**

**Syntax:**
FPStream_SetMark (const FPStreamRef pStream)

**Return Value:**
void

**Input Parameters:**
const FPStreamRef pStream

**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.

**Parameter:**
const FPStreamRef pStream
The reference to a stream as created by FPStream_CreateGenericStream.

**Example:**
Refer to Storing an Encrypted File in the Centera Programmer’s Guide, P/N 069001127, for a code example.

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_OPERATION_REQUIRES_MARK (program logic error)
- Any error that is returned by the setMarkerProc callback function.
**FPStream_ResetMark**

**Syntax:**

FPStream_ResetMark (const FPStreamRef pStream)

**Return Value:**

void

**Input Parameters:**

const FPStreamRef pStream

**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).

**Parameter:**

const FPStreamRef pStream

The reference to a stream as created by FPStream_CreateGenericStream.

**Example:**

Refer to Storing an Encrypted File in the Centera Programmer’s Guide, P/N 069001127, for a code example.

**Error Handling:**

FPool_GetLastError() returns ENOERR (zero) if successful or:

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_OPERATION_REQUIRES_MARK (program logic error)
- Any error that is returned by the setMarkerProc callback function.
FPStream_GetInfo

Syntax: FPStream_GetInfo (const FPStreamRef pStream)

Return Value: FPStreamInfo*

Input Parameter: const FPStreamRef pStream

Concurrency Requirement: This function is thread safe.

Description: This function returns information about the given stream.

Parameter: const FPStreamRef pStream
The reference to a stream.

Example: Refer to the example section of FPStream_CreateGenericStream on page 3-122 for an example of FPStream_GetInfo.

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:
• FP_WRONG_REFERENCE_ERR (program logic error)(internal error)
FPStream_Close

Syntax: FPStream_Close (const FPStreamRef pStream)

Return Value: void

Input Parameter: const FPStreamRef pStream

Concurrency Requirement: This function is thread safe.

Description: This function closes the given stream. Be aware that using 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.

Parameter: const FPStreamRef pStream
The reference to a stream (as returned from the functions FPStream_CreateXXX or FPStream_CreateGenericStream). The reference may also be NULL.

Example: FPStream_Close (MyStream)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:
• FP_WRONG_REFERENCE_ERR (program logic error)(internal error)
Query Functions

The Access API provides functionality to query C-Clips that have been stored on a Centera cluster. A special timestamp indicates when a C-Clip has been stored. This timestamp provides a fast lookup of C-Clips and allows queries within a given time frame.

This chapter describes the following query functions:

- FPQuery_Open/FPQuery_OpenW
- FPQuery_GetPoolRef
- FPQuery_FetchResult
- FPQuery_Close
FPQuery_Open/FPQuery_OpenW

Syntax:
FPQuery_Open (const FPPoolRef pPool, const FPLong pStartTime, const FPLong pStopTime, const char *pReserved)
FPQuery_OpenW (const FPPoolRef pPool, const FPLong pStartTime, const FPLong pStopTime, const wchar_t *pReserved)

Return Value: FPQueryRef

Input Parameters:
const FPPoolRef pPool, const FPLong pStartTime, const FPLong pStopTime, const char/wchar_t *pReserved

Concurrency Requirement: This function is thread safe.

Description:
This function opens a query stream to the given pool and queries the C-Clips that have been stored in the pool from the given start time to the given stop time. This function returns a reference to the opened stream.

Note: The server will allow the application to perform this call only if the server capability "clip-enumeration" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be returned.

This function is intended for backup applications and not as a regular application feature. Applications should expect that the clip-enumeration capability has been disabled by the System Operator for security reasons since FPQuery_Open() will return the list of all the C-Clips on the cluster, including C-Clips written by other applications.

Refer to FPPool_GetCapability/FPPool_GetCapabilityW on page 3-21 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the Centera System Operators Guide, P/N069001126.

Note: The maximum number of parallel queries to a Centera cluster is 10.

Parameters:
- const FPPoolRef pPool
  The reference to a pool opened by FPPool_Open.
• const FPLong pStartTime
  pStartTime specifies from what time the C-Clips that you query have been written to the pool. The time has to be given in milliseconds counted from midnight January 1, 1970 and is based on the UTC (Coordinated Universal Time, also known as GMT – Greenwich Mean Time) of the system clock of the Centera server that has stored the C-Clips. If pStartTime is 0, the function queries all C-Clips that have been stored on the pool until (not inclusive) pStopTime.

• const FPLong pStopTime
  pStopTime specifies until what time (not inclusive) the C-Clips that you query have been written to the pool. The time has to be given in milliseconds counted from midnight January 1, 1970 and is based on UTC (Coordinated Universal Time, also known as GMT – Greenwich Mean Time) of the system clock of the Centera server that has stored the C-Clips. pStopTime must be before or the same as the time when this function is called. pStopTime = -1 refers to the current time.

  Note: Because pStartTime is inclusive and pStopTime is not, you cannot use the same values for these parameters. If you do, this function will return the FP_PARAM_ERR error.

• const char/wchar_t *pReserved
  pReserved is currently not in use, set it to NULL.

Example:

This example shows how to query all data that has been stored on a pool.

```c
vPoolRef = FPPool_Open (POOL_ADDR) ;
FPQueryRef vQuery = FPQuery_Open (vPoolRef, 0, -1, NULL) ;
while (true)
{ FPLong vTimestamp ;
  FPInt vCode = FPQuery_FetchResult (vQuery, vNewID2, &vTimestamp, 1000) ;
  printf ("query result code=%ld clip=%s err=%ld\n", (long) vCode, vNewID2,
          (long) errno) ;
  if (vCode == FP_QUERY_RESULT_CODE_END)
    break ;
}
FPQuery_Close (vQuery) ;
FPPool_Close (vPoolRef) ;
```
Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:

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

**Syntax:**
FPQuery_GetPoolRef (const FPQueryRef pQuery)

**Return Value:**
FPPoolRef

**Input Parameter:**
const FPQueryRef pQuery

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves the reference to the pool for which the given query has been opened. The application should either close the pool reference returned by FPPool_Open or by this function, but not both.

**Parameter:**
const FPQueryRef pQuery
The reference to a query stream as returned from the FPQuery_Open function. The reference may also be NULL.

**Example:**
MyPool = FPQuery_GetPoolRef (MyQuery)

**Error Handling:**
FPPool_GetLastError() returns ENOERR (zero) if successful or:
- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_POOLCLOSED_ERR (program logic error)
- PP_QUERYCLOSED_ERR (program logic error)
**FPQuery_FetchResult**

**Syntax:**
```c
FPQuery_FetchResult (const FPQueryRef pQuery, FPClipID pResultClip, FPLong *pTimestamp, const FPInt pTimeout)
```

**Return Value:**
FPInt

**Input Parameters:**
- const FPQueryRef pQuery
- const FPInt pTimeout

**Output Parameters:**
- FPClipID pResultClip
- FPLong *pTimestamp

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function fetches the C-Clips individually – in time ascending order – that FPQuery_Open has queried and returns a value that indicates the status of the function.

**Note:** Only return value 0 indicates that the function has fetched a valid C-Clip ID that can be processed by the application. All other return values indicate that the fetched C-Clip ID is not valid and cannot be properly processed by the application.

This function can return the following values:

- 0 (FP_QUERY_RESULT_CODE_OK) – The function has fetched a valid C-Clip ID that can be processed by the application.
- 1 (FP_QUERY_RESULT_CODE_INCOMPLETE) – The returned C-Clip ID is not valid. The query results that follow may be incomplete because one or more nodes could not be queried. We recommend restarting the query from the last valid pTimestamp before FP_QUERY_RESULT_CODE_INCOMPLETE was returned.
- 2 (FP_QUERY_RESULT_CODE_COMPLETE) – This value will always be 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.
- 3 (FP_QUERY_RESULT_CODE_END) – The query is finished, no more query results are expected.
• 4 (FP_QUERY_RESULT_CODE_ABORT) – The query has been aborted due to a problem on the server side or because pStartTime used with FPQuery_Open() is later than the current server time. Check pStartTime and retry the query.

• 99 (FP_QUERY_RESULT_CODE_PROGRESS) – The query is in progress.

• -1 (FP_QUERY_RESULT_CODE_ERROR) - An error has been detected during the execution of this call. Check FPPool_GetLastError() to get the Centera error code. In case error FP_SOCKET_ERR (-10101), has been returned, call FPPool_GetLastErrorInfo() to check the OS-dependent error code. If this refers to a timeout error, for example 10060 on Windows, we recommend restarting this call. Otherwise return the error code when the call has been executed.

Parameters:

• const FPQueryRef pQuery
  The reference to the query stream opened by FPQuery_Open.

• FPClipID pResultClip
  The C-Clip ID of the C-Clip that is returned to the function.

• FPLong *pTimestamp
  pTimestamp refers to the time when the returned C-Clip has been stored on the pool.

• const FPInt pTimeout
  pTimeout refers to the time in milliseconds that the function will wait for the next result. If pTimeout = -1, the function uses the default timeout of 120000 ms (2 minutes).

Example:

Refer to the example of FPQuery_Open/FPQuery_OpenW on page 3-154 and the Centera Programmer’s Guide, P/N 069001127, Cookbook Backing up C-Clips to an External Device.

Error Handling:

FPPool_GetLastError() returns ENOERR (zero) if successful or:

• PP_PARAM_ERR (program logic error)
• PP_WRONG_REFERENCE_ERR (program logic error)(internal error)
FPQuery_Close

Syntax: FPQuery_Close (const FPQueryRef pQuery)

Return Value: void

Input Parameters: const FPQueryRef pQuery

Concurrency Requirement: This function is thread safe.

Description: This function closes a query stream that has been opened by FPQuery_Open.

Parameters: const FPQueryRef pQuery
The reference to a query stream as returned from the FPQuery_Open function. The reference may also be NULL.

Example: FPQuery_Close (NewQuery)

Error Handling: FPPool_GetLastError() returns ENOERR (zero) if successful or:
• FP_WRONG_REFERENCE_ERR (program logic error)(internal error)
Monitoring Functions

SDK version 2.1 and higher offers the Monitor API (MoPI). With this API you can gather monitoring information from the Centera server.

The MoPI is part of the FPLibrary shared library. The FPMonitor library authenticates itself to the server using the PAI module.

The information retrieved by the MoPI is in xml format and can be basic, statistical, or event-related. Refer to Appendix A, Monitoring Information for more details.

These are the MoPI functions:

- FPMonitor_Open
- FPMonitor_Close
- FPMonitor_GetDiscovery
- FPMonitor_GetDiscoveryStream
- FPMonitor_GetAllStatistics
- FPMonitor_GetAllStatisticsStream
- FPEventCallback_RegisterForAllEvents
- FPEventCallback_Close

When a monitoring transaction fails, the monitoring function will failover to another Access Node – either one from the parameter list or one from the probe. If none of the Access Nodes respond, the function will return an error.
**FPMonitor_Open/FPMonitor_OpenW**

**Syntax:**

FPMonitor_Open (const char *inClusterAddress)/
FPMonitor_OpenW (const wchar_t *inClusterAddress)

**Return Value:**

FPMonitorRef

**Input Parameter:**

const char *inClusterAddress/const wchar_t *inClusterAddress

**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function creates a new object to monitor the state of the Centera server. The monitoring functions will operate on the first available IP address in the given list of cluster addresses.

This function checks the availability of an Access Node 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 Access Nodes 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 = "152.62.69.153?c:\centera\rwe.pea";
MyMonitor = FPMonitor_Open (MyClusterAddress)

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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)
**FPMonitor_Close**

**Syntax:**

FPMonitor_Close (const FPMonitorRef inMonitor)

**Return Value:**

void

**Input Parameter:**

const FPMonitorRef inMonitor

**Concurrency Requirement:**

This function is thread safe.

**Description:**

This function closes the given monitor object and frees all related (memory) resources.

**Parameters:**

const FPMonitorRef inMonitor

The reference to a monitor object as returned from the FPMonitor_Open function. The reference may also be NULL.

**Example:**

FPMonitor_Close(vMonitor);

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_WRONG_REFERENCE_ERR (program logic error)
- PP_OBJECTINUSE_ERR (client error)
FPMonitor_GetDiscovery

Syntax:  FPMonitor_GetDiscovery (const FPMonitorRef inMonitor, char *outData, FPInt *ioDataLen)

Return Value:  void

Input Parameters:  const FPMonitorRef inMonitor, FPInt *ioDataLen

Output Parameters:  char *outData, FPInt *ioDataLen

Concurrency Requirement:  This function is thread safe.

Description:  This function retrieves discovery information about the Centera cluster in xml format and writes it to the given buffer. The monitor object has been opened with the FPMonitor_Open function.

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 will allow the application to perform this call if the server capability "monitor" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be 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 the FPMonitor_Open function. The reference may also be NULL.
- char *outData
  outData is the memory buffer that will store the discovery information.
- FPInt *ioDataLen
  On input ioDataLen specifies the length, in characters, of the outData buffer. On output it specifies the actual length of the string in characters.

Example:  char vBuffer[10*1024];
          FPInt l=sizeof(vBuffer);
FPMonitor_GetDiscovery(vMonitor, vBuffer, &l);

**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- **PP_PARAM_ERR** (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- **PP_ATTR_NOT_FOUND_ERR** (internal error)
- **PP_TAG_HAS_NO_DATA_ERR** (program logic error)
- **PP_Wrong_REFERENCE_ERR** (program logic error)
- **PP_FILE_NOT_STORED_ERR** (program logic error)
- **PP_NO_POOL_ERR** (network error)
- **PP_NO_SOCKET_AVAIL_ERR** (network error)
- **PP_PROTOCOL_ERR** (internal error)
- **PP_UNKNOWN_OPTION** (internal error)
- **PP_SERVER_ERR** (server error)
- **PP_CONTROLFIELD_ERR** (server error)
- **PP_NOT_RECEIVE_REPLY_ERR** (network error)
- **PP_BEGDATA_ERR** (internal error)
- **PP_BEGBUSY_ERR** (server error)
- **PP_SERVER_NOTREADY_ERR** (server error)
- **PP_PROBEPACKET_ERR** (internal error)
- **PP_CLIPCLOSED_ERR** (program logic error)
- **PP_POOLCLOSED_ERR** (program logic error)
- **PP_OPERATION_NOT_ALLOWED** (client error)
- **PP_WRONG_STREAM_ERR** (client error)
FPMonitor_GetDiscoveryStream

Syntax:
FPMonitor_GetDiscoveryStream (const FPMonitorRef inMonitor, FPStreamRef inStream)

Return Value:
void

Input Parameters:
const FPMonitorRef inMonitor, FPStreamRef inStream

Concurrency Requirement:
This function is thread safe.

Description:
This function retrieves discovery information about the Centera cluster in xml format and writes it to the given stream. The monitor object has been opened with the FPMonitor_Open function.

Note: The server will allow the application to perform this call if the server capability "monitor" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be 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 the FPMonitor_Open function. 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_CreateToStudio();
FPMonitor_GetDiscoveryStream(vMonitor, vStream);
FPStream_Close(vStream);

Error Handling:
FPool_GetLastError() returns ENOERR (zero) if successful or:
- 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)
- `PP_PROTOCOL_ERR` (internal error)
- `PP_UNKNOWN_OPTION` (internal error)
- `PP_SERVER_ERR` (server error)
- `PP_CONTROLFIELD_ERR` (server error)
- `PP_NOT_RECEIVE_REPLY_ERR` (network error)
- `PP_SEGDATA_ERR` (internal error)
- `PP_BLOBBUSY_ERR` (server error)
- `PP_SERVER_NOTREADY_ERR` (server error)
- `PP_PROBEPACKET_ERR` (internal error)
- `PP_CLIPCLOSED_ERR` (program logic error)
- `PP_POOLCLOSED_ERR` (program logic error)
- `PP_OPERATION_NOT_ALLOWED` (client error)
- `PP_WRONG_STREAM_ERR` (client error)
**FPMonitor_GetAllStatistics**

**Syntax:**
```
FPMonitor_GetAllStatistics (const FPMonitorRef inMonitor,
char *outData, FPInt *ioDataLen)
```

**Return Value:**
void

**Input Parameters:**
const FPMonitorRef inMonitor, FPInt *ioDataLen

**Output Parameters:**
char *outData, FPInt *ioDataLen

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function retrieves all available statistical information about the Centera cluster in xml format and writes it to the given buffer. The monitor object has been opened with the `FPMonitor_Open` function.

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 will allow the application to perform this call if the server capability "monitor" is true. If this capability is false, the error `FP_OPERATION_NOT_ALLOWED` will be 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 the `FPMonitor_Open` function. The reference may also be NULL.
- `char *outData`
  `outData` is the memory buffer that will store the statistical information.
- `FPInt *ioDataLen`
  On input `ioDataLen` specifies the length, in characters, of the `outData` buffer. On output it specifies the actual length of the string in characters.

**Example:**
```
char vBuffer[10*1024];
FPInt l=sizeof(vBuffer);
FPMonitor_GetAllStatistics(vMonitor, vBuffer, &l);
```
**Error Handling:**

FPPool_GetLastError() returns ENOERR (zero) if successful or:

- **PP_PARAM_ERR** (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- **PP_ATTR_NOT_FOUND_ERR** (internal error)
- **PP_TAG_HAS_NO_DATA_ERR** (program logic error)
- **PP_WRONG_REFERENCE_ERR** (program logic error)
- **PP_FILE_NOT_STORED_ERR** (program logic error)
- **PP_NO_POOL_ERR** (network error)
- **PP_NO_SOCKET_AVAIL_ERR** (network error)
- **PP_PROTOCOL_ERR** (internal error)
- **PP_UNKNOWN_OPTION** (internal error)
- **PP_SERVER_ERR** (server error)
- **PP_CONTROLFIELD_ERR** (server error)
- **PP_NOT_RECEIVE_REPLY_ERR** (network error)
- **PP_SEGDATA_ERR** (internal error)
- **PP_BLOBBUSY_ERR** (server error)
- **PP_SERVER_NOTREADY_ERR** (server error)
- **PP_PROBEPACKET_ERR** (internal error)
- **PP_CLIPCLOSED_ERR** (program logic error)
- **PP_POOLCLOSED_ERR** (program logic error)
- **PP_OPERATION_NOT_ALLOWED** (client error)
- **PP_WRONG_STREAM_ERR** (client error)
FPMonitor_GetAllStatisticsStream

Syntax:  
FPMonitor_GetAllStatisticsStream (const FPMonitorRef inMonitor, FPStreamRef inStream)

Return Value:  void

Input Parameters:  const FPMonitorRef inMonitor, FPStreamRef inStream

Concurrency Requirement:  This function is thread safe.

Description:  This function retrieves all available statistical information about the Centera cluster in xml format and writes it to the given stream. The monitor object has been opened with the FPMonitor_Open function.

Note: The server will allow the application to perform this call if the server capability "monitor" is true. If this capability is false, the error FP_OPERATION_NOT_ALLOWED will be 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 the FPMonitor_Open function. 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_CreateToStudio();  
FPMonitor_GetAllStatisticsStream(vMonitor, vStream);  
FPStream_Close(vStream);

Error Handling:  FPPool_GetLastError() returns ENOERR (zero) if successful or:

- 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)
C API Reference

- `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)
**FPEventCallback_RegisterForAllEvents**

**Syntax:**
```c
FPEventCallback_RegisterForAllEvents (const FPMonitorRef inMonitor, FPStreamRef inStream)
```

**Return Value:**
FPEventCallbackRef

**Input Parameters:**
const FPMonitorRef inMonitor, FPStreamRef inStream

**Concurrency Requirement:**
This function is thread safe.

**Description:**
This function asynchronously registers the application to receive Centera events (alerts) in xml format. The registration will be 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 will send – with an interval determined by the server – keep-alive monitoring packets to the cluster to ensure that the Access Node is still online. The server answers with a keep-alive reply.

If the Access Node is offline, the alert-receiving thread fails over to another Access Node. If all Access Nodes 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 will allow the application to perform this call if the server capability "monitor" is true. If this capability is false, the error `FP_OPERATION_NOT_ALLOWED` will be returned.

**Parameters:**
- const FPMonitorRef inMonitor
  The reference to a monitor object as returned from the `FPMonitor_Open` function. The reference may also be NULL.
- FPStreamRef inStream
  The reference to a stream (as returned from the functions `FPStream_CreateXXX` or `FPStream_CreateGenericStream`).

**Example:**
```c
FPStreamRef vStream = CreateMyStream();
```
FPEventCallbackRef
vRef=FPEventCallback_RegisterForAllEvents(vMonitor, vStream);

Error Handling:
FPool_GetLastError() returns ENOERR (zero) if successful or:

- PP_Param_Err (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- PPVertexAttribArray_not_found (internal error)
- PP_Tag_has_no_data_Err (program logic error)
- PP_Wrong_reference_Err (program logic error)
- PP_File_not_stored_Err (program logic error)
- PP_No_Pool_Err (network error)
- PP_No_Socket_Avail_Err (network error)
- PP_Protocol_Err (internal error)
- PP_Unknown_option (internal error)
- PP_Server_Err (server error)
- PP_Controlfield_Err (server error)
- PP_Not_receive_reply_Err (network error)
- PP_Segdata_Err (internal error)
- PP_Blobbusy_Err (server error)
- PP_Server_Notready_Err (server error)
- PP_ProbePacket_Err (internal error)
- PP_ClipClosed_Err (program logic error)
- PP_PoolClosed_Err (program logic error)
- PP_Operation_Not_allowed (client error)
- PP_Wrong_Stream_Err (client error)
**FPEventCallback_Close**

**Syntax:**

```c
FPEventCallback_Close (const FPEventCallbackRef inRegister)
```

**Return Value:**

`void`

**Input Parameters:**

`const FPEventCallbackRef inRegister`

**Concurrency Requirement:**

This function must be 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. The reference may also be `NULL`.

**Example:**

`FPEventCallback_Close(vRef);`

**Error Handling:**

`FPPool_GetLastError()` returns `ENOERR` (zero) if successful or:

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

As well as Centera specific error codes the API may also report operating system error codes. If the API returns an error code which is not listed below, refer to a list of platform-specific error codes (Windows error code 10055, for example, means that no buffer space is available).

All Centera error codes are displayed as negative values. If you want to see the error name or the description of the error code instead of the error number, you need to call the Toolbox functions `FPError_GetErrorString` or `FPError_GetErrorMessage`. In this case you need to include `FPErrorUtils.h`. This file also provides three more error handling functions. Refer to Chapter 5, *Toolbox* for more information on the Toolbox error functions.

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Error Name</th>
<th>Description &amp; Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10001</td>
<td>FP_INVALID_NAME</td>
<td>The name that you have used is not xml compliant.</td>
</tr>
<tr>
<td>-10002</td>
<td>FP_UNKNOWN_OPTION</td>
<td>You have used an unknown option name with <code>FPPool_SetIntOption</code> and/or <code>FPPool_GetIntOption</code>.</td>
</tr>
<tr>
<td>-10003</td>
<td>FP_NOT_SEND_REQUEST_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10004</td>
<td>FP_NOT_RECEIVE_REPLY_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10005</td>
<td>FP_SERVER_ERR</td>
<td>The server reports an error. An internal error on the server occurred. Try again.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10006</td>
<td>PP_PARAM_ERR</td>
<td>You have used an incorrect or unknown parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Example:</strong> 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.</td>
</tr>
<tr>
<td>-10007</td>
<td>PP_PATH_NOT_FOUND_ERR</td>
<td>This path does not correspond to a file or directory on the client system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The path in one of your parameters does not point to an existing file or directory. Verify the path in your code.</td>
</tr>
<tr>
<td>-10008</td>
<td>PP_CONTROLFIELD_ERR</td>
<td>The server reports that the operation generated a “Controlfield missing” error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This internal error was generated because the required control field was not found. Try again. (Obsolete from v2.0.)</td>
</tr>
<tr>
<td>-10009</td>
<td>PP_SEGDATA_ERR</td>
<td>The server reports that the operation generated a “Segdatafield missing” error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.)</td>
</tr>
<tr>
<td>-10010</td>
<td>PP_DUPLICATE_FILE_ERR</td>
<td>A duplicate CA already exists on the server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you did not enable duplicate file detection, verify that you have not already stored this data and try again.</td>
</tr>
<tr>
<td>-10011</td>
<td>PP_OFFSET_FIELD_ERR</td>
<td>The server reports that the operation generated an “Offsetfield missing” error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This internal error was generated because the offset field was not found in the packet. Try again. (Obsolete from v2.0.)</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10012</td>
<td>FP_OPERATION_NOT_SUPPORTED</td>
<td>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.</td>
</tr>
<tr>
<td>-10013</td>
<td>FP_ACK_NOT_RCV_ERR</td>
<td>A write acknowledgment was not received. Verify your LAN connections and try again.</td>
</tr>
<tr>
<td>-10014</td>
<td>FP_FILE_NOT_STORED_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10015</td>
<td>FP_NUMLOC_FIELD_ERR</td>
<td>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.)</td>
</tr>
<tr>
<td>-10016</td>
<td>FP_SECTION_NOT_FOUND_ERR</td>
<td>The GetSection request could not retrieve the defined section tag. This internal error was generated because a required section is missing in the ClipFile (also referred to as CDF). Verify the content of your code and try again. (Obsolete from v2.0.)</td>
</tr>
<tr>
<td>-10017</td>
<td>FP_TAG_NOT_FOUND_ERR</td>
<td>The referenced tag could not be found in the C-Clip Descriptor File (CDF). This internal error was generated because information is missing from the description section in the ClipFile (also referred to as CDF). Verify the content of your code and try again.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10018</td>
<td>PP_ATTR_NOT_FOUND_ERR</td>
<td>Could not find an attribute with that name. If FPTag_GetXXAttribute 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.</td>
</tr>
<tr>
<td>-10019</td>
<td>PP_WRONG_REFERENCE_ERR</td>
<td>The reference that you have used is invalid. The reference was not opened, already closed, or not of the correct type.</td>
</tr>
<tr>
<td>-10020</td>
<td>PP_NO_POOL_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10021</td>
<td>PP_CLIP_NOT_FOUND_ERR</td>
<td>Could not find the referenced C-Clip in the cluster. Returned by FPClip_Open, it means the ClipFile (also referred to as CDF) could not be found on the server. Verify that the original data was correctly stored and try again.</td>
</tr>
<tr>
<td>-10022</td>
<td>PP_TAGTREE_ERR</td>
<td>An error exists in the tag tree. Verify the content of your code and try again.</td>
</tr>
<tr>
<td>-10023</td>
<td>PP_ISNOT_DIRECTORY_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10024</td>
<td>PP_UNEXPECTEDTAG_ERR</td>
<td>Either a “file” or “folder” tag was expected but not given. An unexpected tag was found when retrieving the ClipFile (also referred to as CDF). The ClipFile is probably corrupt.</td>
</tr>
<tr>
<td>-10025</td>
<td>PP_TAG_READONLY_ERR</td>
<td>The tag cannot be changed or deleted (it is probably a top tag). Verify your program logic.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10026</td>
<td>FP_OUT_OF_BOUNDS_ERR</td>
<td>The options parameter is out of bounds. One of the function parameters exceeds its preset limits. Verify each parameter in your code.</td>
</tr>
<tr>
<td>-10027</td>
<td>FP_FILESYS_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10029</td>
<td>FP_STACK_DEPTH_ERR</td>
<td>You have exceeded the 64 nested tag limit. Review the structure of your content description and try again.</td>
</tr>
<tr>
<td>-10030</td>
<td>FP_TAG_HAS_NO_DATA_ERR</td>
<td>You are trying to access blob data of a tag that does not contain blob data. This error will also be returned when FPTag_BlobPurge is called from an application compiled with SDK v1.0, v1.0 SP1, v1.0 SP2, or v1.1.</td>
</tr>
<tr>
<td>-10031</td>
<td>FP_VERSION_ERR</td>
<td>The C-Clip has been created using a more recent version of the client software than you are using. Upgrade to the latest version.</td>
</tr>
<tr>
<td>-10032</td>
<td>FP_MULTI_BLOB_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10033</td>
<td>FP_PROTOCOL_ERR</td>
<td>You have used an unknown protocol option (we support only HPP). 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.</td>
</tr>
<tr>
<td>-10034</td>
<td>FP_NO_SOCKET_AVAIL_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10035</td>
<td>FP_BLOBIDFIELD_ERR</td>
<td>A BlobID field (the Content Address) was expected but not given. Upgrade to the latest client and server versions. (Obsolete from v2.0.)</td>
</tr>
<tr>
<td>-10036</td>
<td>FP_BLOBIDMISMATCH_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10037</td>
<td>FP_PROBEPACKET_ERR</td>
<td>The probe packet does not contain valid server addresses. Upgrade to the latest client and server versions. (Obsolete from v2.0.)</td>
</tr>
<tr>
<td>-10038</td>
<td>FP_CLIPCLOSED_ERR</td>
<td>(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.</td>
</tr>
<tr>
<td>-10039</td>
<td>FP_POOLCLOSED_ERR</td>
<td>(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.</td>
</tr>
<tr>
<td>-10040</td>
<td>FP_BLOBBUSY_ERR</td>
<td>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.</td>
</tr>
<tr>
<td>-10041</td>
<td>FP_SERVER_NOTREADY_ERR</td>
<td>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 Access Nodes are running but the Storage Nodes have not been initialized yet. This error can also occur when not enough mirror groups are found on the server. Try again.</td>
</tr>
<tr>
<td>-10042</td>
<td>FP_SERVER_NO_CAPACITY_ERR</td>
<td>The server has no capacity to store data. Enlarge the server’s capacity and try again.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10101</td>
<td>FP_SOCKET_ERR</td>
<td>An error on the network socket occurred. Verify the network.</td>
</tr>
<tr>
<td>-10102</td>
<td>FP_PACKETDATA_ERR</td>
<td>The data packet contains wrong data. Verify the network, the version of the server or try again later.</td>
</tr>
<tr>
<td>-10103</td>
<td>FP_ACCESSNODE_ERR</td>
<td>No Access Node can be found. Verify the IP addresses provided with FPPool_Open.</td>
</tr>
<tr>
<td>-10151</td>
<td>FP_OPCODE_FIELD_ERR</td>
<td>The Query Opcode field is missing from the packet.</td>
</tr>
<tr>
<td>-10152</td>
<td>FP_PACKET_FIELD_MISSING_ERR</td>
<td>The packet field is missing.</td>
</tr>
<tr>
<td>-10153</td>
<td>FP_AUTHENTICATION_FAILED_ERR</td>
<td>Authentication to get access to the server failed. Check the profile name and secret.</td>
</tr>
<tr>
<td>-10154</td>
<td>FP_UNKNOWN_AUTH_SCHEME_ERR</td>
<td>An unknown authentication scheme has been used.</td>
</tr>
<tr>
<td>-10155</td>
<td>FP_UNKNOWN_AUTH_PROTOCOL_ERR</td>
<td>An unknown authentication protocol has been used.</td>
</tr>
<tr>
<td>-10156</td>
<td>FP_TRANSACTION_FAILED_ERR</td>
<td>Transaction on the server failed. FPClip_Delete, FPClip_Purge, or FPTag_BlobPurge could not delete the complete C-Clip or blob because of server problems. Try again later.</td>
</tr>
<tr>
<td>-10201</td>
<td>FP_OPERATIONQUIRES_MARK</td>
<td>The application requires marker support but the stream does not provide that.</td>
</tr>
<tr>
<td>-10202</td>
<td>FP_QUERYCLOSED_ERR</td>
<td>The FPQuery for this object is already closed. (Java only).</td>
</tr>
<tr>
<td>-10203</td>
<td>FP_WRONG_STREAM_ERR</td>
<td>The function expects an input stream and gets an output stream or vice-versa.</td>
</tr>
<tr>
<td>-10204</td>
<td>FP_OPERATION NOTALLOWED</td>
<td>The use of this operation is restricted or this operation is not allowed because the server capability is false.</td>
</tr>
<tr>
<td>-10205</td>
<td>FP_SDK_INTERNAL_ERR</td>
<td>An SDK internal programming error has been detected.</td>
</tr>
<tr>
<td>-10206</td>
<td>FP_OUTOFMEMORY_ERR</td>
<td>The system ran out of memory. Check the system’s capacity.</td>
</tr>
<tr>
<td>-10207</td>
<td>FP_OBJECTINUSE_ERR</td>
<td>Cannot close the object because it is in use. Check your code.</td>
</tr>
<tr>
<td>Value</td>
<td>Error Name</td>
<td>Description &amp; Action</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>-10208</td>
<td>PP_NOTYET_OPEN_ERR</td>
<td>The object is not yet opened. Check your code.</td>
</tr>
<tr>
<td>-10209</td>
<td>PP_STREAM_ERR</td>
<td>An error occurred in the generic stream. Check your code.</td>
</tr>
<tr>
<td>-10210</td>
<td>PP_TAG_CLOSED_ERR</td>
<td>The FPTag for this object is already closed. (Java only.)</td>
</tr>
<tr>
<td>-10211</td>
<td>PP_THREAD_ERR</td>
<td>An error occurred while creating a background thread.</td>
</tr>
</tbody>
</table>
Use this chapter in conjunction with Chapter 3, C API Reference of this manual and javadoc which is available on the SDK CD. The main sections of this chapter are:

- Classes .................................................................4-2
- Errors .................................................................4-13
Classes

- FPPool
- FPClip
- FPTag
- FPStream
- FPQuery

Constants

The static members of the Java interface FPLibraryConstants have the same name equivalents as the constants defined in the C header file FPAPI.h. For example, you can refer to the C constant FP_OPEN_ASTREE in Java as FPLibraryConstants.FP_OPEN_ASTREE.
**FPPool**

Use this class to open, close, or set options for a pool. The system creates the object using the address list (10.10.11.11:3218 and 10.10.11.12 in the code example below).

Example:

```java
String vAddrList[] = { "10.10.11.11:3218", "10.10.11.12" };
FPPool vPool = new FPPool (vAddrList);
vPool.setOption ("buffersize", 10240);
//...
vPool.Close();
```

Mapping:

<table>
<thead>
<tr>
<th>C Functions</th>
<th>Java Method of Class FPPool</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPPool_Open</td>
<td>FPPool(String [] pAddr), FPPool(String pAddr)</td>
</tr>
<tr>
<td>FPPool_Close</td>
<td>Close()</td>
</tr>
<tr>
<td>FPPool_GetPoolInfo</td>
<td>getPoolInfo()</td>
</tr>
<tr>
<td>FPPool_GetCapability</td>
<td>getCapability()</td>
</tr>
<tr>
<td>FPPool_GetIntOption</td>
<td>getOption()</td>
</tr>
<tr>
<td>FPPool_GetGlobalOption</td>
<td>getGlobalOption()</td>
</tr>
<tr>
<td>FPPool_GetClusterTime</td>
<td>getClusterTime()</td>
</tr>
<tr>
<td>FPPool_GetComponentVersion</td>
<td>getComponentVersion()</td>
</tr>
<tr>
<td>FPPool_GetLastErrorInfo</td>
<td>no Java equivalent: the FPLibrary exception object holds this information</td>
</tr>
<tr>
<td>FPPool_SetIntOption</td>
<td>setOption()</td>
</tr>
<tr>
<td>FPPool_SetGlobalOption</td>
<td>setGlobalOption()</td>
</tr>
</tbody>
</table>
Use this inner class of FPPool to retrieve information about the pool. In the example below getPoolInfo() returns an instance of the FPPool.PoolInfo class. Use the get functions to request the information that you need.

Example:

```java
FPPool vPool = new FPPool("152.62.69.169:3218");
FPPool.PoolInfo vPoolInfo = vPool.getPoolInfo();

    System.out.println("pool version "+vPoolInfo.getVersion());
    System.out.println("pool capacity "+vPoolInfo.getCapacity());
    System.out.println("pool free space "+vPoolInfo.getFreeSpace());
    System.out.println("pool cluster id "+vPoolInfo.getClusterID());
    System.out.println("pool replica addr "+vPoolInfo.getReplicaAddress());

vPool.Close();
```

Mapping:

<table>
<thead>
<tr>
<th>C Structure FPPoolInfo Member</th>
<th>Java Method of Class FPPoolInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacity</td>
<td>getCapacity()</td>
</tr>
<tr>
<td>freeSpace</td>
<td>getFreeSpace()</td>
</tr>
<tr>
<td>clusterID</td>
<td>getClusterID()</td>
</tr>
<tr>
<td>clusterName</td>
<td>getClusterName()</td>
</tr>
<tr>
<td>version</td>
<td>getVersion()</td>
</tr>
<tr>
<td>replicaAddress</td>
<td>getReplicaAddress()</td>
</tr>
</tbody>
</table>
The static public initializer `loadFPLibrary` loads the underlying library (dll or shared library) which contains the bodies of the methods defined by this class. The library is called `FPLibrary.dll` on Windows and `libFPLibrary.so` on Unix systems. It must be in a location where the virtual machine can find it.
**FPClip**

Use this class to open, close, update, read, or write a C-Clip.

In the example below, the system first retrieves a C-Clip from the cluster (using its unique address or C-Clip ID) and displays information about this C-Clip on the screen. Subsequently the system retrieves the top-level tag of the C-Clip to modify or create lower level tags. Finally, the system writes the changed C-Clip to the cluster and returns a new address (C-Clip ID).

**Example:**

```java
FPClip vClip = new FPClip(vPool,
   "F14MJOVKGI2O2e31ET1RRKB96BK",
   FPLibraryConstants.FP_OPEN_ASTREE) ;

System.out.println("num blobs: " + vClip.getNumBlobs()) ;
System.out.println("total size: " +
   vClip.getTotalSize()) ;
System.out.println("clipid: " + vClip.getClipID()) ;
System.out.println("name: " + vClip.getName()) ;
Date date = new Date(vClip.getCreationDate()) ;
System.out.println("creation: " +
   vClip.getCreationDate() + " " + date) ;

FPTag vTopTag = vClip.getTag() ; //...
String vNewClipID = vClip.Write () ;
vClip.Close() ;
```

**Mapping:**

<table>
<thead>
<tr>
<th>C Functions</th>
<th>Java Method of Class FPClip</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPClip_Create</td>
<td>FPClip(FPPool pPool, String pName)</td>
</tr>
<tr>
<td>FPClip_SetName</td>
<td>FPClip.SetName(String)</td>
</tr>
<tr>
<td>FPClip_Open</td>
<td>FPClip(FPPool pPool, String pClipID, int pOpenMode)</td>
</tr>
<tr>
<td>FPClip_Close</td>
<td>Close()</td>
</tr>
<tr>
<td>FPClip_Exists</td>
<td>Exists(FPPool pPool, String pClipID)</td>
</tr>
<tr>
<td>FPClip_Delete</td>
<td>Delete(FPPool pPool, String pClipID)</td>
</tr>
<tr>
<td>FPClip_Purge</td>
<td>Purge(FPPool pPool, String pClipID)</td>
</tr>
<tr>
<td>FPClip_GetTopTag</td>
<td>getTopTag()</td>
</tr>
</tbody>
</table>
### Classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPClip_GetPoolRef</td>
<td>getPoolRef()</td>
</tr>
<tr>
<td>FPClip_GetNumTags</td>
<td>getNumTags()</td>
</tr>
<tr>
<td>FPClip_GetNumBlobs</td>
<td>getTotalBlobs()</td>
</tr>
<tr>
<td>FPClip_GetTotalSize</td>
<td>getTotalSize()</td>
</tr>
<tr>
<td>FPClip_GetClipID</td>
<td>getClipID()</td>
</tr>
<tr>
<td>FPClip_GetName</td>
<td>getName()</td>
</tr>
<tr>
<td>FPClip_GetCreationDate</td>
<td>getCreationDate()</td>
</tr>
<tr>
<td>FPClip_SetRetentionPeriod</td>
<td>setRetentionPeriod()</td>
</tr>
<tr>
<td>FPClip_GetRetentionPeriod</td>
<td>getRetentionPeriod()</td>
</tr>
<tr>
<td>FPClip_IsModified</td>
<td>IsModified()</td>
</tr>
<tr>
<td>FPClip_FetchNext</td>
<td>FetchNext()</td>
</tr>
<tr>
<td>FPClip_SetDescriptionAttribute</td>
<td>FPClip.setDescriptionAttribute(String, String)</td>
</tr>
<tr>
<td>FPClip_GetDescriptionAttribute</td>
<td>FPClip.getDescriptionAttribute(String)</td>
</tr>
<tr>
<td>FPClip_GetNumDescriptionAttributes</td>
<td>FPClip.getNumDescriptionAttributes()</td>
</tr>
<tr>
<td>FPClip_GetDescriptionAttributeIndex</td>
<td>String[]</td>
</tr>
<tr>
<td>FPClip_RemoveDescriptionAttribute</td>
<td>FPClip.removeDescriptionAttribute(String)</td>
</tr>
<tr>
<td>FPClip_Write</td>
<td>Write()</td>
</tr>
<tr>
<td>FPClip_RawRead</td>
<td>RawRead (OutputStream)</td>
</tr>
<tr>
<td>FPClip_RawOpen</td>
<td>FPClip (FPPool pPool, String pClipID, InputStream pStream, long pOptions)</td>
</tr>
</tbody>
</table>

**Note:** The parameter `long pOptions` is reserved for future expansion.
FPTag

Use this class to open, close, update, read, or write a tag. You can also use it to read or write blobs and to purge (delete) them if necessary.

Examples:

1. Tag manipulation

```java
String vTagName = pTag.getName () ;
String vName    = pTag.getStringAttribute ("NAME") ;

if (vTagName.equalsIgnoreCase (FP_TAG_FOLDER))
{
   File vNewDir = new File (pParentDir, vName) ;
   vNewDir.mkdir () ;

   FPTag vChild = pTag.getFirstChild () ;
   while (vChild != null)
   {
      RetrieveTag (vChild, vNewDir) ;
      FPTag vNext = vChild.getSibling () ;
      vChild.Close() ;
      vChild = vNext ;
   }
}
```

2. Blob manipulation

```java
if (vTag.getTag().equalsIgnoreCase("file"))
{
   // we got a File object to put the file data into,
   // create the file on the local disk:
   pFile.createNewFile () ;

   // create an outputstream object from it
   OutputStream vStream = new FileOutputStream (pFile) ;

   // read the blob data into the stream
   pTag.BlobRead (vStream) ;

   // flush & close the output stream
   vStream.close() ;
}
```
## Java API Reference

### Mapping:

<table>
<thead>
<tr>
<th>C Functions</th>
<th>Java Method of Class FPTag</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPTag_Create</td>
<td>FPTag(FPTag pParent, String pName)</td>
</tr>
<tr>
<td>FPTag_Copy</td>
<td>FPTag (FPTag pTag, FPTag pParentTag, int pOptions)</td>
</tr>
<tr>
<td>FPTag_Close</td>
<td>Close()</td>
</tr>
<tr>
<td>FPTag_GetPoolRef</td>
<td>getPoolRef()</td>
</tr>
<tr>
<td>FPTag_GetClipRef</td>
<td>getClipRef()</td>
</tr>
<tr>
<td>FPTag_GetSibling</td>
<td>getSibling()</td>
</tr>
<tr>
<td>FPTag_GetPrevSibling</td>
<td>getPrevSibling()</td>
</tr>
<tr>
<td>FPTag_GetFirstChild</td>
<td>getFirstChild()</td>
</tr>
<tr>
<td>FPTag_GetParent</td>
<td>getParent()</td>
</tr>
<tr>
<td>FPTag_Delete</td>
<td>Delete()</td>
</tr>
<tr>
<td>FPTag_GetTagName</td>
<td>getTagName()</td>
</tr>
<tr>
<td>FPTag_GetBlobSize</td>
<td>getBlobSize()</td>
</tr>
<tr>
<td>FPTag_SetStringAttribute</td>
<td>setAttribute(String pName, String pValue)</td>
</tr>
<tr>
<td>FPTag_SetLongAttribute</td>
<td>setAttribute(String pName, long pValue)</td>
</tr>
<tr>
<td>FPTag_SetBoolAttribute</td>
<td>setAttribute(String pName, boolean pValue)</td>
</tr>
<tr>
<td>FPTag_GetStringAttribute</td>
<td>getStringAttribute(String pName)</td>
</tr>
<tr>
<td>FPTag_GetLongAttribute</td>
<td>getLongAttribute(String pName)</td>
</tr>
<tr>
<td>FPTag_GetBoolAttribute</td>
<td>getBooleanAttribute(String pName)</td>
</tr>
<tr>
<td>FPTag_RemoveAttribute</td>
<td>RemoveAttribute(String pName)</td>
</tr>
<tr>
<td>FPTag_GetNumAttributes</td>
<td>getNumAttributes()</td>
</tr>
<tr>
<td>FPTag_GetIndexAttribute</td>
<td>getAttributes()</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FPTag_BlobWrite</td>
<td>BlobWrite(InputStream pStream)</td>
</tr>
<tr>
<td></td>
<td>BlobWrite(InputStream pStream, long pOptions)</td>
</tr>
<tr>
<td>FPTag_BlobRead</td>
<td>BlobRead(OutputStream pStream)</td>
</tr>
<tr>
<td></td>
<td>BlobRead(OutputStream pStream, long pOptions)</td>
</tr>
<tr>
<td>FPTag_BlobReadPartial</td>
<td>BlobReadPartial(OutputStream pStream, long pOffset, long pLength, long pOptions)</td>
</tr>
<tr>
<td>FPTag_BlobPurge</td>
<td>BlobPurge()</td>
</tr>
<tr>
<td>FPTag_BlobExists</td>
<td>BlobExists()</td>
</tr>
</tbody>
</table>

**Note:** To retry FPTag.BlobWrite, FPTag.BlobRead and FPTag.BlobReadPartial, marking should be supported. Use the FPFileInputStream class to support marking.
FPStream

No direct equivalents of the C based FPStream functions exist in Java. Use a subclass of java.io.InputStream or java.io.OutputStream instead (belonging to the standard java.io package).

To use FP_OPTION_CLIENT_CALCID, the InputStream must support the mark() method, that is, markSupported() must return true. The java.io.FileInputStream does not support the mark() method, use FPPFileInputStream instead. Refer to Editing Attributes in the Centera Programmer’s Guide, P/N 069001127, for an example on how to use FP_OPTION_CLIENT_CALCID.
Use this class to query C-Clips that are stored on the pool within a certain time frame.

**Example:**

```java
String vAddr = "127.0.0.1:3218";
FPPool vPool = new FPPool(vAddr);
long vFromTimestamp = 0L;
long vToTimestamp = -1L; // Use current timestamp as 'to'
// timestamp
FPQuery vQuery = new FPQuery(vPool, vFromTimestamp, vToTimestamp);
while (true)
{
    FPQuery.QueryResult vResult =
        vQuery.FetchResult(120000L); // Give 2 minutes timeout
    if (vResult.isQueryProgress())
    {
        long vTimestamp = vResult.getTimeStamp();
    }
    else if (vResult.isQueryEnd())
        break;
    else if (vResult.isQueryAborted())
        break;
    else
    {
        long vTimestamp = vResult.getTimeStamp();
        int vCode = vResult.getStatus();
        String vId = vResult.getClipID();
    }
}
vQuery.Close();
```

**Mapping:**

<table>
<thead>
<tr>
<th>C Function</th>
<th>Java Method of Class FPQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPQuery_Open</td>
<td>FPQuery (FPPool pPool, long pStartTime, long pStopTime, String pReserved)</td>
</tr>
<tr>
<td>FPQuery_GetPoolRef</td>
<td>getPoolRef()</td>
</tr>
<tr>
<td>FPQuery_FetchResult</td>
<td>FetchResult()</td>
</tr>
<tr>
<td>FPQuery_Close</td>
<td>Close()</td>
</tr>
</tbody>
</table>
Errors

The static members of the Java interface FPLibraryErrors have the same name equivalents as the errors defined in the C header file FPErrors.h. For example, you can refer to the C error code FP_INVALID_NAME in Java as FPLibraryErrors.FP_INVALID_NAME. Refer to Error Codes on page 3-175 for a complete listing of all Centera error codes.

Additionally you can also call three Toolbox error functions on any FPLibraryException that is thrown in Java. Refer to Chapter 5, Toolbox for more information on these functions.
The Toolbox API is an easy-to-use library that allows you to store and retrieve a C-Clip using a single function instead of multiple API function calls. The C Toolbox also contains two functions that return a text representation and an error description of a given error number. The Java Toolbox enables you to backup C-Clips that have been stored on the server and to restore this backup on the server again.

This chapter describes all available Toolbox functions. All functions must be considered synchronous function calls. This means that the application developer needs to implement threading and asynchronous behavior.

The main sections in this chapter are:

- Error Handling ................................................................. 5-2
- C Toolbox Functions .......................................................... 5-4
- Java Toolbox Methods ....................................................... 5-9
Error Handling

The Toolbox functions retrieve the error status of the last function call and return the error number. The function returns zero if no error has occurred. Refer to Error Codes on page 3-175 for a complete listing of all error values and their description.

If you include FPErrUtil.h you can also call the following error handling functions on every C API call:

- **Boolean FPErr_IsClientError (const short pError)**
pError is the input parameter. This method returns true if pError probably has been caused by a program logic error in the client application. A parameter is wrong or preset boundaries have been reached.

- **Boolean FPErr_IsNetworkError (const short pError)**
pError is the input parameter. This method returns true if pError probably has been caused by a network problem. The Centera server could not be reached, no additional socket could be opened or no acknowledgement has been received from the server.

- **Boolean FPErr_IsServerError (const short pError)**
pError is the input parameter. This method returns true if pError probably has been caused by a server error. Check the server logs for troubleshooting.

In addition you can call the following three methods on any FPLibraryException thrown in Java.

- **boolean isClientError()**
  This method returns true if the exception probably has been caused by a program logic error in the client application. A parameter is wrong or preset boundaries have been reached.

- **boolean isNetworkError()**
  This method returns true if the exception probably has been caused by a network problem. The Centera server could not be reached, no additional socket could be opened or no acknowledgement has been received from the server.

- **boolean isServerError()**
  This method returns true if the exception probably has been caused by a server error. Check the server logs for troubleshooting.
• boolean isOSError()
  This method returns true if the exception denotes a
  platform-specific error and is not a client, server, or network
  error.
C Toolbox Functions

This section describes the available Toolbox functions for C. With these functions you can easily:

- store a C-Clip
- retrieve a C-Clip
- display an error number as text
- return an error description
ClipFile_Store

Syntax: `ClipFile_Store (const char *pAddr, const char *pPath, FPClipID pClipID)`

Return Value: FPInt

Input Parameters: const char *pAddr, const char *pPath,

Output Parameter: FPClipID pClipID

Description: The function stores all the files and/or folders addressed by pPath to the pool. The function returns zero if successful or an error value (there is no need to get the error from FPPool_GetLastError()). The resulting C-Clip identifier is stored in pClipID.

Parameters:

- const char *pAddr
  pAddr is a comma-separated string containing one or more addresses of the available Access Nodes.

- const char *pPath
  The path to the file or folder that has to be stored as a C-Clip.

- FPClipID pClipID
  The reference to the C-Clip as returned by the function.

Example:

```c
char *pAddr = "10.10.11.12:3218, 10.10.11.11:3218, storage.emc.com:3218"; 
// store contents of directory 'test' in pool
ClipFile_Store (pAddr, "c:\test", vNewID) ;

// and retrieve it again to directory 'test1'
ClipFile_Retrieve (pAddr, "c:\test1", vNewID) ;
```

Error Handling: This function returns 0 (zero) if successful or:

- PP_PARAM_ERR (program logic error)
- PP_PATH_NOT_FOUND_ERR (program logic error)
- any other FP Library error
ClipFile_Retrieve

Syntax:  
ClipFile_Retrieve (const char *pAddr, const char *pPath,  
const FPClipID pClipID)

Return Value:  
FPInt

Input Parameters:  
const char *pAddr, const char *pPath, const FPClipID pClipID

Description:  
The function retrieves the complete content of a C-Clip from the pool  
and stores the content in the folder pPath. The function returns zero  
if successful or an error value (there is no need to get the error from  
FPPool_GetLastError()).

Note: ClipFile_Retrieve is used to retrieve only C-Clips that were  
created using ClipFile_Store. It is incompatible with any other C-Clip  
structures.

Parameters:
- const char *pAddr
  pAddr is a comma-separated string containing one or more  
  addresses of the available Access Nodes of the pool.
- const char *pPath
  The path of the folder where the contents of the C-Clip should be  
  stored.
- const FPClipID pClipID
  The reference to the C-Clip to be retrieved from the pool.

Example:
{
    char   *pAddr = "10.10.11.12:3218, 10.10.11.11:3218,  
                    storage.emc.com:3218)" ;
    // store contents of directory 'test' in pool  
    ClipFile_Store (pAddr, "c:\\test", vNewID) ;

    // and retrieve it again to directory 'test1'  
    ClipFile_Retrieve (pAddr, "c:\\test1", vNewID) ;
}

Error Handling:  
This function returns 0 (zero) if successful or:
- FP_PARAM_ERR (program logic error)
- FP_IS_NOT_DIRECTORY_ERR (program logic error)
- FP_UNEXPECTED_TAG_ERR (internal error)
- any other FPLibrary error
FPError_GetErrorString

Syntax: FPErr or _GetErrorString (const short pError)

Return Value: char *

Input Parameter: const short pError

Description: This function returns a text representation of the given error number.

Note: The application should not change the returned string.

Parameter: const short pError

A Centera error number. Refer to Error Codes on page 3-175 for a complete list of all Centera error codes.

Example: char *vErrorString = FPError_GetErrorString (-10020)

Error Handling: This function returns no errors.
### FPError_GetErrorMessage

**Syntax:**
`FPError_GetErrorMessage (const short pError)

**Return Value:**
char *

**Input Parameter:**
const short pError

**Description:**
This function returns a description of the given error number.

**Note:** The application should not change the returned string.

**Parameter:**
const short pError

A Centera error number. Refer to Error Codes on page 3-175 for a complete list of all Centera error codes.

**Example:**
`char *vErrorMessage = FPError_GetErrorMessage (-10020)`

**Error Handling:**
This function returns no errors.
Java Toolbox Methods

This section describes the Java class FPClipFile and its methods.

FPClipFile

This class stores and retrieves files or folders and returns a string (vClipID).

Example:

```java
FPClipFile vclipFile = new FPClipFile (vAddrList) ;
String vclipID = vclipFile.Store (new File ("c:\\test")) ;

vclipFile.Retrieve (new File ("c:\\test1"), vclipID) ;
vclipFile.Close() ;
```

Mapping:

<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPClipFile</td>
<td>FPClipFile (String pAddresses)</td>
</tr>
<tr>
<td></td>
<td>Opens the connection to the pool and creates an object.</td>
</tr>
<tr>
<td>Store</td>
<td>String Store (File pFileDirItem)</td>
</tr>
<tr>
<td></td>
<td>Stores the contents of the directory or file in the pool and returns the C-Clip ID.</td>
</tr>
<tr>
<td>Retrieve</td>
<td>void Retrieve (File pParentDir, String pClipID)</td>
</tr>
<tr>
<td></td>
<td>Retrieves the C-Clip from the pool to the designated directory.</td>
</tr>
<tr>
<td>none</td>
<td>void FPClipFile.Backup (Date pStartDate, Date pEndDate, JarOutputStream pOutStream)</td>
</tr>
<tr>
<td></td>
<td>Makes a backup of all C-Clips that have been created between pStartDate and pEndDate on the Centera server. The contents of the C-Clips and their associated blobs are stored in a Jar file. The application is responsible for initializing the JarOutputStream.</td>
</tr>
<tr>
<td>none</td>
<td>void Restore (JarFile pIn)</td>
</tr>
<tr>
<td></td>
<td>Restores the data that has been backed up with Backup to the Centera server.</td>
</tr>
<tr>
<td>Close</td>
<td>Close()</td>
</tr>
<tr>
<td></td>
<td>Breaks the connection.</td>
</tr>
</tbody>
</table>
This appendix lists the syntax and sample of xml files that are retrieved by the MoPI functions.

- Discovery Information .......................................................... A-2
- Discovery Information .......................................................... A-2
- Alert Information ................................................................. A-12
Monitoring Information

Discovery Information

This section lists the syntax and a sample of the xml file that can be retrieved by FPMonitor_GetDiscovery.

Syntax Discovery

```xml
<!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? ) >
<!ATTLIST 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 >

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="" location="" name="" protectionschemes="R61,M2" serial="" serviceid=""
    serviceinfo="" siteid="" softwareversion="" availablerawcapacity="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"
               status="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="eth2" 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"
               status="online" systemid="276e989a-1dd2-11b2-9a8e-ae2a1b42afc5"
               totalrawcapacity="0" usedrawcapacity="0">
         <nics>
           <nic ip="10.255.1.1" mac="00:02:b3:5f:c5:7a" name="eth0" status="1"/>
           <nic ip="10.255.1.1" 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="eth2" 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>
     </nodes>
</cabinet>
<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="c001n05"
rail="1" roles="storage" softwareversion="2.1.0.287-1715" status="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="1" 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" status="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" status="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>
</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="root">
                    <capabilities>
                        <capability enabled="true" name="purge"/>
                        <capability enabled="true" name="write"/>
                    </capabilities>
                </securityprofile>
            </applicationcontext>
        </applicationcontexts>
    </pool>
</pools>
<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="43737db-1dd2-11b2-aac6-c6edfe63736" 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-af79ff89fd28" 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 `PPEventCallback_RegisterForAllEvents`.

### Syntax Alert

```xml
<!ELEMENT alert (failure)>
<!ATTLIST alert type (degradation|improvement) #REQUIRED>

<!ELEMENT failure (node, device)>
<!ATTLIST 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
<?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>
```
This glossary contains terms that are used in this manual and are related to disk storage subsystems.

**A**

**Access Node**
The nodes in a cluster that communicate with the outside world and with the cluster’s Storage Nodes. They must have public IP addresses.

**Application Program(ming) Interface (API)**
A set of function calls that enables communication between applications or between an application and an operating system.

**Automatic (AC) Transfer Switch (ATS)**
An AC power transfer switch. Its basic function is to deliver output power from one of two customer facility AC sources. It guarantees that the cluster will continue to function if a power failure occurs on one of the power sources by automatically switching to the secondary source.

**B**

**Blob**
The Distinct Bit Sequence (DBS) of user data. The DBS represents the actual content of a file and is independent of the filename and physical location.

**Note:** Do not confuse this term with the term “Binary Large Object” that exists in the database sector.
Glossary

C

C-Clip A package containing the user’s data and associated metadata. When a user presents a file to the Centera system, the system calculates a unique Content Address (CA) for the data and then stores the file. The system also creates a separate XML file containing the CA of the user’s file and application-specific metadata. Both the XML file and the user’s data are stored in the C-Clip.

C-Clip Descriptor File (CDF) The additional XML file that the system creates when making a C-Clip. This file includes the Content Addresses for all referenced blobs and associated metadata.

C-Clip ID The Content Address that the system returns to the client. It is also referred to as a C-Clip handle or C-Clip reference.

Cluster One or more cabinets on which the nodes are clustered. Clustered nodes are automatically aware of nodes that attach to and detach from the cluster.

Cluster Time The synchronized time of all the nodes within a cluster.

Command Line Interface (CLI) A set of predefined commands that you can enter via a command line. The Centera CLI allows a user to manage a cluster and monitor its performance.

Content Address (CA) An identifier that uniquely addresses the content of a file and not its location. Unlike location-based addresses, Content Addresses are inherently stable and, once calculated, they never change and always refer to the same content.

Content Address Resolution The process of discovering the IP address of a node containing a blob with a given Content Address.

Content Address Verification The process of checking data integrity by comparing the CA calculations that are made on the application server (optional) and both Storage Nodes.

Content Addressed Storage (CAS) The generic term for a Centera cluster and its software. In the same way that a Symmetrix is considered a SAN device, a cluster is considered a CAS device.

Content Protection Mirrored (CPM) The content protection scheme where each stored object is copied to another node on a Centera cluster to ensure data redundancy.
**Content Protection Parity (CPP)**
The content protection scheme where each object is fragmented into several segments that are stored on separate nodes with a parity segment to ensure data redundancy.

**Cube**
A collection of 8, 16, 24, or 32 nodes and two cube switches, forming the basic building block for a cluster.

**Distinct Bit Sequence (DBS)**
The actual content of a file independent of the filename and physical location. Every file consists of a unique sequence of bits and bytes. The DBS of a user's file is referred to as a blob in the Centera system.

**Dynamic Host Configuration Protocol (DHCP)**
An internet protocol used to assign IP addresses to individual workstations and peripherals in a LAN.

**End-to-end checking**
The process of verifying data integrity from the application end down to the second Storage Node. 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.

**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.

**Input parameter**
The required or optional information that has to be supplied to a function.
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load balancing</strong></td>
<td>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 Access Node; second, the Access Node selects the least loaded Storage Node to read or write data.</td>
</tr>
<tr>
<td><strong>Local Area Network (LAN)</strong></td>
<td>A set of linked computers and peripherals in a restricted area such as a building or company.</td>
</tr>
<tr>
<td><strong>Message Digest 5 (MD5)</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Mirror team</strong></td>
<td>A logical organization of a number of nodes that always mirror each other.</td>
</tr>
<tr>
<td><strong>MultiCast Protocol (MCP)</strong></td>
<td>A network protocol used for communication between a single sender and multiple receivers.</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>Logically, a network entity that is uniquely identified through a system ID, IP address, and port. Physically, a node is a computer system that is part of the Centera cluster.</td>
</tr>
<tr>
<td><strong>Output parameter</strong></td>
<td>The information that a function returns to the application that called the function.</td>
</tr>
<tr>
<td><strong>Pool</strong></td>
<td>A set of separate clusters that are linked together to constitute one Content Addressed Storage device.</td>
</tr>
<tr>
<td><strong>Pool Transport Protocol (PTP)</strong></td>
<td>A further evolution of the UniCast Protocol (UCP) used for communication over the Internet between the application server and an Access Node.</td>
</tr>
</tbody>
</table>
Glossary

Probing  A process where the application server requests information from the cluster to determine if it should start a PTP session.

Redundancy  A process where data objects are duplicated or encoded such that the data can be recovered given any single failure. Refer to Content Protection Mirrored (CPM), Content Protection Parity (CPP), and, Replication for specific redundancy schemes used in Centera.

Regeneration  The process of creating a data copy if a mirror copy or fragmented segment of that data is no longer available.

Relaying  A way of streaming data directly from a Storage Node over an Access Node 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.

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 assignment. This node can become a Storage or Access Node.

Storage Node  The nodes in a cluster that store data.

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.
Glossary

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.
A
Access Application Program Interface see API 1-7
Access Node 1-4, 1-7, g-1
Access Node error 3-181
acknowledgement error 3-177
AIX
   supported compilers 2-4
   system requirements 2-4
alert information A-12
   sample A-12
   syntax A-12
API 1-7, g-1
API Toolbox 5-1
Application Program(ming) Interface see API g-1
asynchronous behavior 5-1
ATS 1-4, g-1
attribute error 3-178
authentication error 3-181
authentication protocol error 3-181
authentication scheme error 3-181
Automatic (AC) Transfer Switch see ATS 1-4, g-1

B
blob 1-10, g-1
   error 3-177, 3-180
BlobExists(), Java method 4-10
BlobID field error 3-180
BlobID mismatch 3-180
BlobPurge(), Java method 4-10
BlobRead(), Java method 4-10
BlobReadPartial(), Java method 4-10
BlobWrite(), Java method 4-10
buffer size pool setting 3-13

C
C Toolbox Functions 5-4
CA 1-9, g-2
   duplicate 3-176
cabinet 1-3
capabilities server 3-22
CAS 1-9, g-2
C-Clip 1-9, g-2
   error 3-178
   handle g-2
   ID g-2
   identifier 5-5
   reference 5-5, 5-6, g-2
   retrieve content 5-6
   store content 5-6
C-Clip Descriptor File see CDF g-2
C-Clips and multithreading 3-27
CDF g-2
Centera features 1-8
Centera interfaces 1-7
Centera Viewer 1-7
CENTERA_CUSTOM_METADATA 3-28, 3-61
CenteraPing 1-7, 2-19
CenteraVerify 1-7, 2-6
   INI file 2-14
   log file 2-12
   using 2-8
CenteraVerify for Unix 2-16
child tag 3-83
CLI 1-7, g-2
   definition g-2
ClipFile_Retrieve() 5-6
ClipFile_Store() 5-5
Close()
Java Toolbox function 5-9
Close(), JAVA class 5-9
Close(), Java method 4-3
closed C-Clip error 3-180
closed pool error 3-180
close 1-3
capacity 3-17
definition g-2
free space 3-17
identifier 3-17
name 3-17
replication address 3-18
software version 3-18
cluster time g-2
CodeWarrior 2-2
Command Line Interface see CLI g-2
compliancy 1-19
connection error 3-178
Content Address Resolution g-2
Content Address see CA g-2
Content Address Verification g-2
Content Addressed Storage see CAS 1-9, g-2
costent protection 1-11
Content Protection Mirrored see CPM g-2
Content Protection Parity see CPP g-3
Controlfield missing error 3-176
Coordinated Universal Time (UTC) 3-54
copying data g-5
CPM 1-11
CPP 1-11
cube g-3
cube switch 1-4
customer metadata 3-28, 3-61

D
data
error 3-179
handle incoming/outgoing g-5
data integrity check g-2, g-3
data packet error 3-181
data retention 1-12
data shredding 1-18
DBS g-3
defauxlcollisionavoidance pool setting 3-13
Delete(), Java method 4-6, 4-9

Department of Defense 1-22
rule 5015 1-22
DHCP g-3
directory error 3-178
disaster recovery 1-19
discovery information A-2
sample A-5
syntax A-2
disk failure 1-11
disk regeneration 1-14
Distinct Bit Sequence see DBS g-3
duplicate CA 3-176
Dynamic Host Configuration Protocol see DHCP g-3

E
decd-to-end checking g-3
error
Access Node 3-181
acknowledgement 3-177
attribute 3-178
authentication 3-181
authentication protocol 3-181
authentication scheme 3-181
blob 3-177
BlobID field 3-180
capacity server 3-180
C-Clip 3-178
connection 3-178
Controlfield missing 3-176
data 3-179
data packet 3-181
descriptions 3-175
directory 3-178
file system 3-179
generic stream 3-182
network socket 3-181
number 3-175, 3-176, 3-177, 3-178, 3-179, 3-180, 3-181, 3-182
Numlockfield missing 3-177
object in use 3-181
object not open 3-182
Offsetfield missing 3-176
packet field 3-181
parameter 3-176
path 3-176
probe packet 3-180
protocol 3-179
SDK internal 3-181
section 3-177
Segdatafield missing 3-176
send request 3-175
server 3-175
socket 3-179
stack depth 3-179
status 5-2
system memory 3-181
tag 3-177, 3-179
tag tree 3-178
thread 3-182
wrong reference 3-178
Exists(), Java method 4-6
Extensible Markup Language see XML  g-3

F
failover g-3
Fast Blob Naming 1-24
FetchNext(), Java method 4-7
file system error 3-179
files
  store 5-9
firewall 2-8
fixed content 1-11
folders
  store 5-9
FP_ACCESSNODE_ERR 3-181
FP_ACK_NOT_RCV_ERR 3-177
FP_ATTR_NOT_FOUND_ERR 3-178
FP_AUTHENTICATION_FAILED_ERR 3-181
FP_BLOBBUSY_ERR 3-180
FP_BLOBIDFIELD_ERR 3-180
FP_BLOBIDMISMATCH_ERR 3-180
FP_CLIP_NOT_FOUND_ERR 3-178
FP_CLOPCLOSED_ERR 3-180
FP_CONTROLFIELD_ERR 3-176
FP_DEFAULT_RETENTION_PERIOD 3-39
FP_DUPLICATE_FILE_ERR 3-176
FP_FILE_NOT_STORED_ERR 3-177
FP_FILESYS_ERR 3-180
FP_INFINITE_RETENTION_PERIOD 3-39
FP_INVALID_NAME 3-175
FP_ISNOT_DIRECTORY_ERR 3-178
FP_MULTI_BLOB_ERR 3-179
FP_NO_POOL_ERR 3-178
FP_NO_RETENTION_PERIOD 3-39
FP_NO_SOCKET_AVAIL_ERR 3-179
FP_NOT_RECEIVE_REPLY_ERR 3-175
FP_NOT_SEND_REQUEST_ERR 3-175
FP_NOTYET_OPEN_ERR 3-182
FP_NUMLOC_FIELD_ERR 3-177
FP_OBJECTINUSE_ERR 3-181
FP_OFFSET_FIELD_ERR 3-176
FP_OPCODE_FIELD_ERR 3-181
FP_OPEN_ASTREE 3-32
FP_OPEN_FLAT 3-32
FP_OPERATION_NOT_ALLOWED 3-181
FP_OPERATION_NOT_SUPPORTED 3-177
FP_OPERATIONQUIRES_MARK 3-181
FP_OPTION_BUFFERSIZE 3-13
FP_OPTION_CALCID_NOCHECK 3-106, 3-109,
  3-112
FP_OPTION_CLIENT_CALCID 3-105
FP_OPTION_CLIENT_CALCID_STREAMING
  3-106
FP_OPTION_CLUSTERNONAVAILTIME 3-16
FP_OPTION_COPY_BLOBDATA 3-71
FP_OPTION_COPY_CHILDREN 3-72
FP_OPTION_DEFAULT_COLLISION_AVOIDANCE 3-13
FP_OPTION_DEFAULTOPTIONS 3-109, 3-112
FP_OPTION_DISABLE_COLLISION_AVOIDANCE 3-107
FP_OPTION_ENABLE_COLLISION_AVOIDANCE 3-106
FP_OPTION_ENABLE_DUPLICATE_DETECTION 3-106
FP_OPTION_ENABLE_MULTICLUSTER_FAILOVER 3-13
FP_OPTION_MAXCONNECTIONS 3-15
FP_OPTION_NO_COPY_OPTIONS 3-71
FP_OPTION_PREFETCH_SIZE 3-14
FP_OPTION_RETRYCOUNT 3-15
FP_OPTION_RETRYSLEEP 3-15
FP_OPTION_SERVER_CALCID_STREAMING
  3-106
FP_OPTION_TIMEOUT 3-13
FP_OUT_OF_BOUNDS_ERR 3-179
FP_OUT_OF_MEMORY_ERR 3-181
FP_PACKET_FIELD_MISSING_ERR 3-181
Index

FP_PACKETDATA_ERR 3-181
FP_PARAM_ERR 3-176
FP_PATH_NOT_FOUND_ERR 3-176
FP_POOLCLOSED_ERR 3-180
FP_PROBEPACKET_ERR 3-180
FP_PROTOCOL_ERR 3-179
FP_QUERY_RESULT_CODE_ABORT 3-159
FP_QUERY_RESULT_CODE_COMPLETE 3-158
FP_QUERY_RESULT_CODE_END 3-158
FP_QUERY_RESULT_CODE_ERROR 3-159
FP_QUERY_RESULT_CODE_INCOMPLETE 3-158
FP_QUERY_RESULT_CODE_OK 3-158
FP_QUERY_RESULT_CODE_PROGRESS 3-159
FP_QUERYCLOSED_ERR 3-181
FP_SDK_INTERNAL_ERR 3-181
FP_SECTION_NOT_FOUND_ERR 3-177
FP_SEGDATA_ERR 3-176
FP_SERVER_ERR 3-175
FP_SERVER_NO_CAPACITY_ERR 3-180
FP_SERVER_NOTREADY_ERR 3-180
FP_SOCKET_ERR 3-181
FP_STACK_DEPTH_ERR 3-179
FP_STREAM_ERR 3-182
FP_TAG_CLOSED_ERR 3-182
FP_TAG_HAS_NO_DATA_ERR 3-179
FP_TAG_NOT_FOUND_ERR 3-177
FP_TAG_READONLY_ERR 3-178
FP_TAGTREE_ERR 3-178
FP_THREAD_ERR 3-182
FP_TRANSACTION_FAILED_ERR 3-181
FP_UNEXPECTEDTAG_ERR 3-178
FP_UNKNOWN_AUTH_SCHEME_ERR 3-181
FP_UNKNOWN_OPTION 3-175
FP_VERSION_ERR 3-179
FP_WRONG_REFERENCE_ERR 3-178
FP_WRONG_STREAM_ERR 3-181
FPClip(), Java class 4-6
FPClip(), Java method 4-6
FPClip.setDescriptionAttribute(), Java method 4-7
FPClip.setDescriptionAttribute(), Java method 4-7
FPClip.removeDescriptionAttribute(), Java method 4-7
FPClip.removeDescriptionAttributeW() 3-40
FPClip.SetName(), Java method 4-6
FPClip.Close() 3-44
FPClip_Create() 3-28
FPClip_CreateW() 3-28
FPClip_Delete() 3-40
FPClip.Exists() 3-45
FPClip_Close() 3-44
FPClip_Create() 3-28
FPClip_CreateW() 3-28
FPClip_Delete() 3-40
FPClip.Exists() 3-45
FPClip.FetchNext() 3-59
FPClip.GetClipID() 3-53
FPClip.GetCreationDate() 3-54
FPClip.GetDescriptionAttribute() 3-63
FPClip.GetDescriptionAttributeIndex() 3-66
FPClip.GetDescriptionAttributeIndexW() 3-66
FPClip.GetDescriptionAttributeW() 3-63
FPClip.GetName() 3-52
FPClip.GetNameW() 3-52
FPClip.GetNumBlob() 3-50
FPClip.GetNumDescriptionAttributes() 3-65
FPClip.GetNumTags() 3-49
FPClip.GetPoolRef() 3-48
FPClip.GetRetentionPeriod() 3-55
FPClip.GetTopTag() 3-58
FPClip.GetTotalSize() 3-51
FPClip.IsModified() 3-56
FPClip_Open() 3-31
FPClip_Purge() 3-42
FPClip_RawOpen() 3-36
FPClip_RawRead() 3-35
FPClip_RawOpen() 3-36
FPClip_RawRead() 3-35
FPClip_RemoveDescriptionAttribute() 3-40, 3-68
FPClip.RemoveDescriptionAttributeW() 3-68
FPClip.SetDescriptionAttributeW() 3-61
FPClipSetName() 3-30
FPClipSetNameW() 3-30
FPClipSetRetentionPeriod() 3-38
FPClip_Write() 3-32, 3-33
FPClipFile 5-9
FPClipFile(), Java Toolbox function 5-9
FPClipID 3-3
FPError_GetErrorMessage() 5-8
FPError_GetErrorString() 5-7
FPError_IsClientError 5-2
FPError_IsNetworkError() 5-2
FPError_IsServerError 5-2
FPErrorInfo 3-3
FPErrorUtils.h 3-175, 5-2
FPEventCallback_Close() 3-174
FPEventCallback_RegisterForAllEvents() 3-172, A-12
FPInt 3-3
FPLibraryConstants, Java interface 4-2
FPLibraryConstants.FP_OPEN_ASTREE 4-2
FPLibraryErrors, Java interface 4-13
FPLibraryErrors.FP_INVALID_NAME 4-13
FPLong 3-3
FPMonitor_Close() 3-163
FPMonitor_GetAllStatistics() 3-168
FPMonitor_GetAllStatisticsStream() 3-170
FPMonitor_GetDiscovery() 3-164
FPMonitor_Open() 3-162
FPMonitor_OpenW() 3-162
FPool(), Java class 4-3
FPool(), Java method 4-3
FPool.PoolInfo(), Java class 4-4
FPool.PoolInfo() 4-12
FPool.Close() 3-12
FPool.GetCapability() 3-21
FPool.GetCapabilityW() 3-21
FPool.GetClusterTime() 3-24
FPool.GetComponentVersion() 3-25
FPool.GetComponentVersionW() 3-25
FPool.GetGlobalOption() 3-20
FPool.GetGlobalOptionW() 3-20
FPool.GetIntOption() 3-19
FPool.GetIntOptionW() 3-19
FPool.GetLastError() 3-10
FPool.GetLastErrorInfo() 3-11
FPool.GetPoolInfo() 3-17
FPool.Open() 3-6
FPool.OpenW() 3-6
FPool.SetGlobalOption() 3-15
FPool.SetGlobalOptionW() 3-15
FPool.SetIntOption() 3-13
FPool.SetIntOptionW() 3-13
FPoolInfo 3-3
FPoolInfo() 4-12
FPoolInfo(), Java method 4-4
FPQuery 3-154
FPQuery.Close() 3-160
FPQuery.FetchResult() 3-158
FPQuery.GetPoolRef() 3-157
FPQuery.Open() 3-154
FPQuery.OpenW() 3-154
FPSHORT 3-3
FSStream_Close() 3-152
FSStream_Complete() 3-148
FSStream_CreateBufferForInput() 3-139
FSStream_CreateBufferForOutput() 3-140
FSStream_CreateFileForInput() 3-134
FSStream_CreateFileForOutput() 3-136
FSStream_CreateGenericStream() 3-122
FSStream_CreateTemporaryFile() 3-144
FSStream_CreateToNull() 3-143
FSStream_CreateToStdio() 3-142
FSStream_CreateWithBuffer() 3-141
FSStream_CreateWithFile() 3-138
FSStream_GetInfo() 3-151
FSStream_GetMarker() 3-146
FSStream_PrepareBuffer() 3-134, 3-147
FSStream_Read() 3-146
FSStream_ResetMark() 3-150
FSStream_SetMark() 3-149
FSStream_SetMarker() 3-146
FSStream_Write() 3-146
FSStreamInfo 3-3
FPTag(), Java class 4-8
FPTag(), Java method 4-9
FPTagBlobExists() 3-116
FPTagBlobPurge() 3-114
FPTagBlobRead() 3-108
FPTagBlobReadPartial() 3-111
FPTagBlobWrite() 3-104
FPTag.Close() 3-78
FPTag.Copy() 3-71
FPTagCreate() 3-70
FPTagCreateW() 3-70
FPTagDelete() 3-79
FPTagGetBlobSize() 3-77
FPTagGetBoolAttribute() 3-97
FPTagGetClipRef() 3-74
FPTagGetFirstChild() 3-83
FPTagGetGlobalAttribute() 3-100
FPTagGetGlobalAttributes() 3-96
FPTagGetGlobalAttributeW() 3-96
FPTagGetGlobalAttributesW() 3-99
FPTagGetGlobalAttributeW() 3-99
FPTagGetGlobalAttributes() 3-99
FPTagGetGlobalAttributeW() 3-99
FPTagGetGlobalAttributesW() 3-99
FPTagGetGlobalAttributeW() 3-99
FPTagGetStringAttribute() 3-94
FPTagGetStringAttributeW() 3-94
FPTagGetTagName() 3-75
FPTagRemoveAttribute() 3-98
Index

FPTag_RemoveAttributeW() 3-98
FPTag_SetBoolAttribute() 3-92
FPTag_SetBoolAttributeW() 3-92
FPTag_SetLongAttribute() 3-90
FPTag_SetLongAttributeW() 3-90
FPTag_SetStringAttribute() 3-88
FPTag_SetStringAttributeW() 3-88
Front Panel 1-7
function library 5-1
function outcome  g-5

G
garbage collection 1-17
generic stream
create 3-122
operation 3-118
generic stream error 3-182
getAttributes(), Java method 4-9
getBlobSize(), Java method 4-9
getBooleanAttribute(), Java method 4-9
getCapability(), Java method 4-3
getCapacity(), Java method 4-4
getClipID(), Java method 4-7
getClipRef(), Java method 4-9
getClusterID(), Java method 4-4
getClusterName(), Java method 4-4
getClusterTime(), Java method 4-3
getOption(), Java method 4-7
getParent(), Java method 4-9
getPoolInfo(), Java method 4-3, 4-4
getPoolRef(), Java method 4-7, 4-9
getPrevSibling(), Java method 4-9
getPrevSibling(), Java method 4-9
getReplicaAddress(), Java method 4-4
getRetentionPeriod(), Java method 4-7
getSibling(), Java method 4-9
getStringAttribute(), Java method 4-9
getTagName(), Java method 4-9
getTopTag(), Java method 4-6
getFileSize(), Java method 4-7
getVersion(), Java method 4-4
Greenwich Mean Time (GMT) 3-54

H
health status 1-14
HP-UX
system requirements 2-3

I
identifier
C-Clip 5-5
incoming data, handle g-5
input parameter g-3
input stream 3-122
input/output channels 3-118
installing CenteraVerify 2-7
integrity-checking 1-16
invalid name 3-175
IRIX
supported compilers 2-5
system requirements 2-4
isClientError() 5-2
isModified(), Java method 4-7
isNetworkError() 5-2
isOSError() 5-3
isServerError() 5-2

J
Java API reference 4-1
Java classes 4-2
Java FPLibrary Interface 5-9
Java method
BlobExists() 4-10
BlobPurge() 4-10
BlobRead() 4-10
BlobReadPartial() 4-10
BlobWrite() 4-10
Close() 4-3
Delete() 4-6, 4-9
Exists() 4-6
FetchNext() 4-7
FPClip()  4-6
FPClip.getDescriptionAttribute()  4-7
FPClip.getNumDescriptionAttributes()  4-7
FPClip.removeDescriptionAttribute()  4-7
FPClip.setDescriptionAttribute()  4-7
FPClip.SetName()  4-6
FPPool()  4-3
FPPoolInfo()  4-4, 4-12
FPTag()  4-9
getAttributes()  4-9
getBooleanAttribute()  4-9
getCapacity()  4-4
getCreationDate()  4-7
getFirstChild()  4-9
getComponentVersion()  4-3
getFreeSpace()  4-4
getGlobalOption()  4-3
getLongAttribute()  4-9
getName()  4-7
getNumAttributes()  4-9
getNumBlobs()  4-7
getNumTags()  4-7
getOption()  4-3
getParent()  4-9
getPrevSibling()  4-9
getReplicaAddress()  4-4
getRetentionPeriod()  4-7
getSibling()  4-9
getStringAttribute()  4-9
getTagName()  4-9
getTopTag()  4-6
getTotalSize()  4-7
getVersion()  4-4
IsModified()  4-7
RawRead()  4-7
RemoveAttribute()  4-9
setAttribute()  4-9
setRetentionPeriod()  4-7
Write()  4-7

Java Toolbox Methods  5-9
java.io.InputStream  4-11
java.io.OutputStream  4-11

L
LAN  g-4
Linux
    system requirements  2-3
load balancing  g-4
loadFPLibrary()  4-5
Local Area Network see LAN  g-4
local support procedures  1-21

M
MCP  g-4
MD5  g-4
Message Digest 5 see MD5  g-4
mirror team  g-4
mismatch BlobID  3-180
missing
    Controlfield  3-176
    Numlockfield  3-177
    Offsetfield  3-176
    Segdatafield  3-176
Monitor API see MoPI  1-24
monitoring functions  3-161
MoPI  1-24
MultiCast Protocol see MCP  g-4
multiclusterfailover pool setting  3-13
multithreading and C-Clips  3-27

N
name
    error  3-175
navigate through tags  3-81
network socket error  3-181
node  g-4
node failure  1-11
Numlockfield missing error  3-177

O
Offsetfield missing error  3-176
operation, not supported  3-177
option name, unknown  3-175
out of bounds, options parameter 3-179
outgoing data, handle g-5
output parameter g-4
output stream 3-122

P
packet field error 3-181
parameter
error 3-176
unknown 3-176
parent tag 3-86
path error 3-176
pClipID 5-5
PDU 1-4
pool g-4
pool functions 3-3, 3-5
pool setting
buffer size 3-13
defaultcollisionavoidance 3-13
multiclusterfailover 3-13
prefetchsize 3-14
timeout 3-13
Pool Transport Protocol see PTP g-4
port number 2-8
Power Distribution Unit see PDU 1-4
power rail failure 1-11
pPoolAddress 3-7
prefetchsize pool setting 3-14
probe packet error 3-180
probing g-5
protection schemes 1-11
protocol
error 3-179
unknown 3-179
PTP g-4
purge a blob 3-114
Purge, static method 4-6

Q
query functions 3-153

R
RawRead(), Java method 4-7
read-only tag 3-178
redundancy g-5
reference to C-Clip 5-5, 5-6
regeneration 1-14, g-5
relaying g-5
remote serviceability procedures 1-21
RemoveAttribute(), Java method 4-9
replica cluster 3-7
replication 1-19, g-5
replication address 3-18
retention period g-5
retrieve C-Clip content 5-6
Retrieve(), JAVA class 5-9
return value g-5
root switch 1-4

S
SDK internal error 3-181
SEC 1-22
section error 3-177
Securities and Exchange Commission see SEC 1-22
Segdatafield missing error 3-176
segmentation g-5
self-healing 1-11, 1-14, 1-16
send request error 3-175
server capabilities 3-22
server capacity error 3-180
server error 3-175
server not ready error 3-180
set attribute functions 3-87
setAttribute(), Java method 4-9
setGlobalOption() 4-3
setOption() 4-3
setRetentionPeriod(), Java method 4-7
sibling tag 3-84, 3-85
size internal buffers 3-13
socket error 3-179
Solaris
supported compilers 2-3
system requirements 2-2
spare node g-5
splitting files g-5
stack depth error 3-179
stackable stream support 3-118
statistics information A-10
sample A-10
syntax A-10
Storage Node 1-4, 1-7, g-5
store files 5-9
folders 5-9
store C-Clip content 5-6
store files 5-5
Store(), Java class 5-9
Store(), Java Toolbox function 5-9
store, files and folders 5-9
stream 3-118, g-5
support of stackable streams 3-118
synchronous function calls 5-1
system memory error 3-181
system requirements
  AIX 2-4
  HP-UX 2-3
  IRIX 2-4
  Linux 2-3
  Solaris 2-2
  Windows 2-2

T
  tag
    read-only 3-178
  unexpected 3-178
tag error 3-177, 3-179
tag tree error 3-178
thread error 3-182
threading 5-1
Time to First Byte see TTFB g-6
timeout setting pool 3-13
Toolbox API 5-1
TTFB g-6

U
  UCP g-6
  UDP g-6
unexpected tag 3-178
UniCast Protocol see UCP g-6
Unix version Centera Verify 2-16
unknown
  option name 3-175
  parameter 3-176
  protocol 3-179
User Datagram Protocol see UDP g-6
using
  CenteraVerify 2-8

V
  version error 3-179
Visual Studio 2-2

W
  WAN g-6
Wide Area Network see WAN g-6
Windows
  supported compilers 2-2
  system requirements 2-2
WORM g-6
Write Once Read Many see WORM g-6
Write(), Java method 4-7
wrong reference error 3-178

X
  XML g-3