**Programming Guide**

SanDisk FDF API

Version 1.2

Sandisk Confidential

sd-logo

Documentation ID: FDF-v0.2-AG-01

# Contents

Contents 4

Chapter 1: Introduction 5

Chapter 2: API 8

Initialization and Shutdown 8

Containers 12

Objects 17

Cache Control 23

Statistics 25

Chapter 3: “Hello World” with FDF 29

Chapter 4: Compiling with FDF 31

Appendix: FDF Statistics 32

# Chapter 1: Introduction

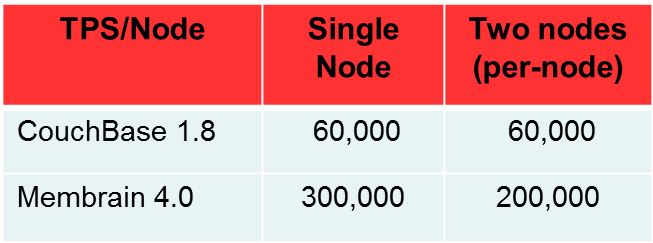
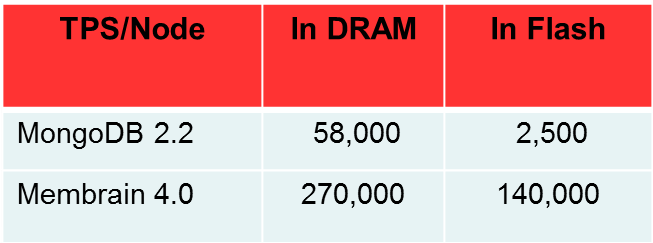
The SanDisk Flash Data Fabric (FDF) is a substrate for flash-optimized data storage solutions, including caches, key-value stores and databases. FDF provides an object API with configurable attributes, and leverages flash storage for high performance and high availability. Higher performance and availability allows multiple servers to be consolidated, with a significant reduction on operating costs.

FDF was developed because many applications realize limited benefits from flash storage without extensive system level optimization. FDF incorporates many of the system level optimizations that are required to exploit flash. Applications can be flash-optimized with much less effort by using FDF as their storage layer

The system level optimizations in FDF include:

* Intelligent DRAM caching.
* Heavily optimized access paths for high performance.
* Optimized threading to maximize concurrency and minimize response time.
* Configurable flash management algorithms to optimize different workloads.
* High Performance Replication with fully automatic failover and failback (roadmap feature).

Figure 1 shows performance data comparing a version of memcached incorporating FDF (membrain) against two other NoSQL key-value stores: MongoDB and CouchBase. The tables show the performance of each application for two cases: when all of the data can fit in memory, and when only part of the data can fit in DRAM. In all cases flash memory is used for storage. The data shows how applications that are not optimized for flash can perform very poorly when data does not fit in DRAM. Membrain, on the other, performs very well when most of the accesses must go to flash.



#### Figure 1: Performance Benefits of FDF

Figure 2 shows the high level architecture of FDF. FDF is a user-space operating environment that manages hardware resources (CPU, memory, network and storage) required for data access and provides software interfaces for container, object and cluster management. The main components of FDF are:

* Container Manager
  + FDF manages data using a “container” model.
  + Containers are collections of objects.
  + Containers support various storage policies that determine how objects are located and cached.
* Object Manager
  + FDF objects are subject to the policies of their owner containers
  + The FDF object manager implements the storage policies of the containers
  + The object manager implements the Create/Replace/Update/Delete (CRUD) operations for objects.
* Cluster Manager (roadmap item)
  + Manages cluster membership, replication, failover and recovery.
* Protocol Layer
  + The FDF protocol layer implements the request/response logic that the upper layers (container, object and cluster managers) use to access the lower layers (DRAM cache, flash manager, replication subsystem, etc.).
* DRAM Cache
  + The DRAM cache holds recently accessed objects in fast main memory.
  + Writes may be handled using a write-through or writeback policy, configurable per container.
* Flash Manager
  + The FDF flash manager is responsible for shard allocation, replication and recovery
* Replication Layer (roadmap item)
  + FDF supports synchronous replication for high availability, providing automatic failover and no data loss.
  + Recovery of a failed node is automatic with highly optimized data transfer.
* Elasticity Module (roadmap item)
  + The FDF elasticity module implements automatic sharding for seamlessly expanding (or shrinking) the number of nodes serving particular containers.
* Messaging Subsystem and Transport Layers
  + The FDF messaging service provides high-speed, robust communications for inter-node FDF messages, remote object access, replication and sharding.
* Threading Module (roadmap item)
  + In addition to pthreads, FDF offers an optional user-mode lightweight threading module that maximizes concurrency by minimizing operating system calls.

Container Mgt

Naming, create, open, delete

FDF Protocol Layer

Object Mgt

Naming, create, delete,

Cluster Mgt

Naming, configure

DRAM

Cache

Flash Manager

Replication

Elasticity Module

Messaging Subsystem

Connect, send, receive

Transport Layer

Threading

Module

#### Figure 2: FDF Architecture

The remainder of this document describes the FDF programming interface. Chapter 2 provides coding examples and detailed descriptions for all FDF functions. Chapter 3 summarizes the key routines of Chapter 2 with a short “Hello, world” program that starts FDF, creates a container, reads, writes and enumerates some objects, and shuts FDF down. Chapter 4 describes how to compile FDF into an application. FDF provides extensive statistics that can be used to tune performance and analyze performance problems. The Appendix provides detailed tables listing all of these statistics.

# Chapter 2: API

## Initialization and Shutdown

Initializing FDF has 3 steps (Listing 1). First, FDF configuration parameters must be initialized if values other than defaults are required. Next, FDFInit() must be called to initialize global FDF state. Third, each thread that uses FDF must initialize per-thread state using FDFInitPerThreadState().

Configuration parameters (properties) can be assigned using FDFSetProperty() and/or FDFLoadProperties(). Table 1 provides a list of all FDF configuration parameters. FDFSetProperty() assigns a value to a single property. FDFLoadProperties() can be used to load many properties from a file. As a final alternative, FDFInit() will automatically load properties from the file specified by the environment variable *FDF\_PROPERTY\_FILE*, if it is set. If a property is assigned a value multiple times, the final assignment takes precedence.

Figure 3 shows a sample configuration file; the configuration file and FDFSetProperty() use the same name for each parameter. FDFInit() uses the property values to initialize FDF global state. Each application thread that will call FDF functions must then allocate a per-thread FDF context using FDFInitPerThreadState(). The opaque context pointer provided by FDFInitPerThreadState() is required for almost all FDF function calls.

Listing 2 shows the typical FDF shutdown sequence. First, all per-thread contexts must be released using FDFReleasePerThreadState(). Once this is done, SDFShutdown() must be called exactly once to release FDF resources.

// FDF Initialization

struct FDF\_state \*state; // Opaque handle for overall FDF state

// Load some properties:

if (***FDFLoadProperties(“/opt/sandisk/config/fdf\_defaults.prop”)*** != SDF\_SUCCESS) {

…

}

// Set other properties as desired

FDFSetProperty(“SDF\_REFORMAT”, “1”); // Do not recover persistent containers

FDFSetProperty(“SDF\_CC\_MAXCACHESIZE”, “20000000000”); // size of DRAM cache in bytes

FDFSetProperty(“AIO\_NUM\_FILES”, “1”); // number of flash files

FDFSetProperty(“SDF\_FLASH\_FILENAME”, “/mnt/ssd/fdf\_flash%d”; // location of flash device/files

…

if (***FDFInit(&state)*** != FDF\_SUCCESS) {

error( "FDFInit() failed" );

}

. . .

// Get a per-thread context for each thread that will call FDF routines

// Put this code where a pthread is initialized

struct FDF\_state \*fdf\_state;

struct FDF\_thread\_state \*thd\_state; // Opaque handle for per-thread FDF state

FDF\_status\_t status;

// fdf\_state created by FDFInit() above

status = ***FDFInitPerThreadState(fdf\_state, &thd\_state)***;

assert(status == FDF\_SUCCESS);

#### Listing 1: Initializing FDF

**FDF\_status\_t FDFLoadProperties (char \**prop\_filename*)**

Load property values.

Read property values from the specified file. Properties not specified in the file are set to built-in defaults or values set by prior calls to FDFLoadProperties() or FDFSetProperty().

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *prop\_filename* | Name of property file |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE if property file could not be loaded

**FDF\_status\_t FDFSetProperty (char \**property\_name,* char \**value\_string*)**

Assign a value to a property.

Interpret the *value\_string* based on the property type (see Table 1).

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *property\_name* | Name of property |
| in | *value\_string* | Value to assign |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE if property could not be set

|  |  |  |  |
| --- | --- | --- | --- |
| **FDF Configuration Properties** | | | |
| **Property Name** | **Type** | **Default** | **Description** |
| AIO\_NUM\_FILES | uint32\_t | 1 | Number of flash devices. |
| SDF\_FLASH\_FILENAME | char \* | /tmp/schooner%d | Location of flash devices. ‘%d’ is replaced with 0, 1, … <AIO\_NUM\_FILES – 1>. |
| SDF\_FLASH\_SIZE | uint32\_t | 200 | Total amount of flash in GB. |
| SDF\_CC\_MAXCACHESIZE | uint64\_t | 100000000 | Bytes of memory reserved for DRAM cache. |
| SDF\_CC\_NSLABS | uint32\_t | 1000 | Number of DRAM cache partitions. |
| SDF\_REFORMAT | uint32\_t | 0 | If 1, delete the contents of all persistent containers (instead of recovering them). |
| SDF\_MAX\_OBJ\_SIZE | uint32\_t | 1048576 | Maximum allowed object size. This cannot exceed 8MB. |
| SDF\_MAX\_OUTSTANDING\_BACKGROUND\_FLUSHES | uint32\_t | 8 | Maximum number of background flushes that can be done in parallel. This can be set to 0 to disable the flusher completely. |
| SDF\_BACKGROUND\_FLUSH\_SLEEP\_MSEC | uint32\_t | 1000 | Time (in milliseconds) for which the background flusher sleeps when it has found no dirty data in the cache. Minimum accepted value is 100. |
| SDF\_ASYNC\_PUT\_THREADS | uint32\_t | 32 | Size of the thread pool that performs asynchronous writeback and flush operations. |
| SDF\_MODIFIED\_FRACTION | double | 1.0 | The maximum fraction of cache space that can hold modified data. If this is exceeded, FDF performs flushes of LRU dirty objects until it drops below the threshold. |
| SDF\_MAX\_FLUSHES\_PER\_MOD\_CHECK | uint32\_t | 10 | Maximum number of flushes that can be issued at a time if the number of modified bytes in the cache exceeds the limit specified by ***cache\_modified\_fraction*** or the "mod\_thresh" admin port command. |

#### Table 1: Field Definitions for FDF Configuration Structure

# Sample Property File

AIO\_NUM\_FILES = 1

SDF\_FLASH\_FILENAME = “/opt/sandisk/flash/ssd%d”

SDF\_FLASH\_SIZE = 100

SDF\_CC\_MAXCACHESIZE = 40000000000

SDF\_MAX\_OBJ\_SIZE = 2097152

SDF\_MAX\_OUTSTANDING\_BACKGROUND\_FLUSHES = 0

#### Figure 3: Sample Property File for FDF Initialization

**FDF\_status\_t FDFInit (struct FDF\_state \*\**fdf\_state*)**

Initialize FDF global state. This function should be called exactly once, and must be called before any other FDF function is called (except **FDFLoadProperties()** or **FDFSetProperty()**).

### Parameters:

|  |  |  |
| --- | --- | --- |
| out | *fdf\_state* | Pointer in which to return opaque handle for FDF global state. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFInitPerThreadState (struct FDF\_state \**fdf\_state*, struct FDF\_thread\_state \*\**thd\_state*)**

Initialize per-thread FDF state. This function should be called once for each application thread that will make calls to FDF functions that require per-thread context.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *fdf\_state* | Opaque handle for FDF global state previously acquired in **FDFInit()**. |
| out | *thd\_state* | Pointer in which to return opaque handle for FDF per-thread state. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise Release per-thread FDF state.

// FDF Termination

// Delete Per-thread FDF Context

// thd\_state created by FDFInitPerThreadState() above

if (***FDFReleasePerThreadState( &thd\_state)*** != FDF\_SUCCESS) {

error( "FDFReleasePerThreadState() failed" );

}

. . .

// fdf\_state allocated using FDFInit() above

// This should only be called after all per-thread

// state handles have been released!

if (***FDFShutdown(fdf\_state)*** != FDF\_SUCCESS) {

error( "FDFShutdown() failed" );

}

#### Listing 2: Terminating FDF

**FDF\_status\_t FDFReleasePerThreadState (struct FDF\_thread\_state \*\**thd\_state*)**

Free per-thread FDF state. This function should be called once by each application thread before shutting down FDF with **FDFShutdown()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFShutdown (struct FDF\_state \**fdf\_state*)**

Stop FDF and release all FDF resources. FDF will flush and sync any buffered objects. This function should be called exactly once, after all per-thread contexts have been released with FDFReleasePerThreadState().

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *fdf\_state* | Opaque handle for FDF global state previously acquired in **FDFInit()**. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

## Containers

FDF objects are manipulated within containers. Containers are created with the FDFOpenContainer() function (Listing 3). Container properties are specified in the structure FDF\_container\_props\_t, which is defined in fdf.h. Table 3 lists the various properties that can be configured, including persistence, cache write policy (writeback or write-through), size, flash management algorithm (fifo or slab), and durability level. For convenience, FDF\_container\_props\_t can be preloaded with default values using FDFLoadCntrPropDefaults().

FDFOpenContainer() returns a container identifier, the cguid, that is used to identify the container in subsequent FDF function calls. The cguid remains valid until the container is closed using FDFCloseContainer(). Containers may be deleted by calling FDFDeleteContainer(). Container deletion does not take effect until there are no threads with the container in the Open state.

An application can retrieve a list of all available containers using FDFGetContainers(). The properties of an open container can be retrieved using FDFGetContainerProps(). Some container properties can be changed dynamically using FDFSetContainerProps(). Table 3 shows which properties can be altered dynamically.

// FDF Containers

/\*\* Structure for container properties (from fdf.h) \*/

typedef struct {

uint32\_t version;

FDF\_boolean\_t fifo\_mode;

FDF\_boolean\_t persistent;

FDF\_boolean\_t writethru;

uint64\_t size;

FDF\_durability\_level\_t durability\_level;

…

} FDF\_container\_props\_t;

...

struct FDF\_thread\_state \*thd\_state;

char \*cname;

FDF\_container\_props\_t props;

FDF\_cguid\_t cguid;

FDF\_status\_t status;

uint32\_t flags;

...

// Load default properties

if (***FDFLoadCntrPropDefaults(&props)*** != SDF\_SUCCESS) {

…

}

// Change default properties as desired.

props.fifo\_mode = FDF\_FALSE;

props.persistent = FDF\_TRUE;

props.writethru = FDF\_TRUE;

props.size\_kb = 20000;

props.durability\_level = FDF\_DURABILITY\_MAX;

flags = FDF\_CTNR\_RO\_MODE; // Open container for read-only access

// (read/write access is default).

flags |= FDF\_CNTR\_CREATE; // Create the container if it doesn’t exist.

// If FDF\_CNTR\_CREATE flag is not set, return

// an error if the container does not exist.

status = ***FDFOpenContainer(***

***thd\_state, // FDF per-thread context***

***cname, // container name***

***props, // container properties***

***flags, // flags***

***&cguid // place to return container id***

***);***

if (status != FDF\_SUCCESS) {

...

}

. . .

status = ***FDFCloseContainer(***

***thd\_state, // FDF per-thread context***

***cguid, // container id***

***);***

if (status != FDF\_SUCCESS) {

...

}

status = ***FDFDeleteContainer(***

***thd\_state, // FDF per-thread context***

***cguid // container id***

***);***

if (status != FDF\_SUCCESS) {

...

}

#### Listing 3: Manipulating Containers

**FDF\_status\_t FDFLoadCntrPropDefaults (FDF\_cntr\_props\_t \**props*)**

Preload a container properties structure with default values.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *props* | Pointer to a properties structure allocated by the application. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

|  |  |  |  |
| --- | --- | --- | --- |
| **FDF\_container\_props\_t Fields** | | | |
| **Name** | **Type** | **Can Change Dynamically?** | **Description** |
| size\_kb | uint32\_t | No | Size of the container in kB. |
| fifo\_mode | FDF\_boolean\_t | No | If FDF\_TRUE, manage flash as a circular buffer with FIFO replacement; if FDF\_FALSE, manage flash using slab allocation. |
| persistent | FDF\_boolean\_t | No | If FDF\_TRUE, persist the container across shutdowns and crashes; if FDF\_FALSE, reformat the container after shutdowns or crashes. |
| evicting | FDF\_boolean\_t | No | If FDF\_TRUE, container will serve as a cache; if FDF\_FALSE, objects are retained until deleted. |
| writethru | FDF\_boolean\_t | Yes | If FDF\_TRUE, use a write-thru policy in the DRAM cache for all writes to this container; if FDF\_FALSE, use a writeback DRAM cache policy. |
| durability\_level | FDF\_durability\_level\_t | No | Set the durability policy for the container:  FDF\_FULL\_DURABILITY: sync each write individually  FDF\_PERIODIC\_DURABILITY: sync storage every ‘k’ writes |

#### Table 3: Field Definitions for FDF Container Properties Structure

**FDF\_status\_t FDFOpenContainer (struct FDF\_thread\_state \**thd\_state*, char \**cname*, FDF\_container\_props\_t \**props*, uint32\_t*flags*, FDF\_cguid\_t \**cguid*)**

Open container 'cname' with the specified properties. If no flags are set (flags=0), the container will be opened in read-write mode, and will NOT be created if it does not already exist. Setting the read-only flag flag (flag |= FDF\_CTNR\_RO\_MODE) opens the container in read-only mode. Setting the create flag (flag |= FDF\_CTNR\_CREATE) will create the container if it does not exist. The 'props' parameter is only used if the create flag is set. If successful, this function returns a container identifier that is is used in subsequent container operations (such as read object, write object, etc.). Once a container is opened, the cguid can be used by multiple application threads. It is not necessary, nor desired, that each thread call open container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cname* | Name of container. |
| in | *props* | Properties of container (only used if it is created). |
| in | *flags* | Flags that modify behavior of the open operation. |
| out | *cguid* | Container identifier to use in subsequent container operations. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist, and FDF\_CTNR\_CREATE is not specified

FDF\_TOO\_MANY\_CONTAINERS if the maximum number of containers has been created

FDF\_FAILURE otherwise

**FDF\_status\_t FDFCloseContainer (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*)**

Close the specified container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_FAILURE otherwise

**FDF\_status\_t FDFDeleteContainer (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*)**

Delete the specified container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_OPEN\_CONTAINER if the container has not been closed

FDF\_FAILURE otherwise

**FDF\_status\_t FDFGetContainers (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t \**cguid\_array*, uint32\_t \**n\_cguids*)**

Retrieve a list of all containers. The list is returned as an FDF-allocated array. The application must subsequently free the array using 'free'.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| out | *cguid\_array* | Pointer in which to return FDF allocated array of cguid's. |
| out | *n\_cguids* | Number of containers in array. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFGetContainerProps (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, FDF\_container\_props\_t \**props*)**

Retrieve the properties for a particular container. The properties are returned via an application-allocated property structure.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container id. |
| in,out | *props* | Pointer to property structure in which to return properties. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFSetContainerProps (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, FDF\_container\_props\_t \**props*)**

Change the properties for a particular container. The properties are passed via an application-allocated property structure. Note that only a small subset of container properties can be changed dynamically. See the section on container properties for details.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container id. |
| in | *props* | Pointer to property structure. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

## Objects

Objects are read, written and deleted to/from an open container using the FDFReadObject(), FDFWriteObject(), and FDFDeleteObject() calls (Listing 4). The FDFReadObject() call returns object data in an FDF-allocated buffer. The application is responsible for freeing this buffer using FDFFreeBuffer().

All of the objects within a container can be enumerated using FDFEnumerateContainerObjects(),

FDFNextEnumeratedObject(), and FDFFinishEnumeration() (Listing 5). The keys and data for enumerated objects are returned via FDF-allocated buffers, which must be subsequently freed using FDFFreeBuffer().

//\*\*\*\*\*\*\*\*\*\* read an object \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FDF\_status\_t status;

struct FDF\_thread\_state\_t \*thd\_state;

FDF\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

char \*pdata;

uint64\_t \*pdatalen;

...

// Object data is returned in an FDF-allocated buffer.

// Application must later release the buffer using

// ***FDFFreeBuffer(char \*pdata).***

status = ***FDFReadObject(***

***thd\_state, // FDF per-thread context***

***cguid, // cguid***

***objkey, // object key***

***keylen, // object key length***

***&pdata, // pointer to data***

***pdatalen // object size***

***);***

if (status == FDF\_OBJECT\_UNKNOWN) {

...

} else if (status != FDF\_SUCCESS) {

...

}

//\*\*\*\*\*\*\*\*\*\* write an object \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FDF\_status\_t status;

struct FDF\_thread\_state \*thd\_state;

FDF\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

uint64\_t datalen;

char \*pdata;

uint32\_t flags;

...

flags = FDF\_WRITE\_MUST\_NOT\_EXIST; // use this to ‘create’

… OR …

flags = FDF\_WRITE\_MUST\_EXIST; // use this to ‘update’

… OR …

flags = 0; // use this to ‘create or update’

status = ***FDFWriteObject(***

***thd\_state, // FDF per-thread context***

***cguid, // container id***

***objkey, // object key***

***keylen, // key length***

***pdata, // data buffer***

***datalen, // data length***

***flags // write condition flags***

***);***

if (status != FDF\_SUCCESS) {

...

}

#### Listing 4: Manipulating Objects

**FDF\_status\_t FDFReadObject (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, char \**key*, uint32\_t*keylen*, char \*\**data*, uint64\_t \**datalen*)**

Read an object from a container. The object is returned via a buffer that is allocated by FDF. The application must subsequently free this buffer using **FDFFreeBuffer()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |
| in | *key* | Key value as an array of bytes. |
| in | *keylen* | Length of key in bytes (including any null termination). |
| out | *data* | Pointer to FDF allocated buffer holding object data. |
| out | *datalen* | Length of data in bytes. |

### Returns:

FDF\_SUCCESS if successful

FDF\_OBJECT\_UNKNOWN if the object does not exist

FDF\_FAILURE otherwise

**FDF\_status\_t FDFFreeBuffer (char \**buf*)**

Frees a buffer that was allocated in FDF and returned via **FDFReadObject()** or **FDFNextEnumeratedObject()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *buf* | Pointer to buffer |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFWriteObject (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, char \**key*, uint32\_t*keylen*, char \**data*, uint64\_t*datalen*, uint32\_t*flags*)**

Write an object to a container. The semantics of the write operation are controlled via 'flags'. If no flags are set (flags=0), the object is written whether or not it already exists. If the existence flag is set (flag |= FDF\_WRITE\_MUST\_NOT\_EXIST), the write succeeds only if the object does NOT already exist. If the non-existence flag is set (flag |= FDF\_WRITE\_MUST\_EXIST), the write succeeds only if the object already exists. It is an error to set both the existence and non-existence flags.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |
| in | *key* | Key value as an array of bytes. |
| in | *keylen* | Length of key in bytes (including any null termination). |
| out | *data* | Pointer to the data to be written. |
| out | *datalen* | Length of data in bytes. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_OBJECT\_UNKNOWN if the object does not exist and the existence flag is set

FDF\_OBJECT\_EXISTS if the object exists and the non-existence flag is set

FDF\_FAILURE otherwise

**FDF\_status\_t FDFDeleteObject (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, char \**key*, uint32\_t*keylen*)**

Delete an object from a container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |
| in | *key* | Key value as an array of bytes. |
| in | *keylen* | Length of key in bytes (including any null termination). |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_OBJECT\_UNKNOWN if the object does not exist

FDF\_FAILURE otherwise

//\*\*\*\*\*\*\*\*\*\* enumerate container objects \*\*\*\*\*\*\*\*\*\*\*

FDF\_status\_t status;

struct FDF\_thread\_state \*thd\_state;

FDF\_cguid\_t cguid;

char \*key;

char \*data;

uint32\_t key\_size;

uint64\_t data\_size;

struct FDF\_iterator \*iterator;

...

status = ***FDFEnumerateContainerObjects(***

***thd\_state, // FDF per-thread context***

***cguid, // container id***

***&iterator); // returns container stats***

if (status != FDF\_SUCCESS) {

…

}

// NOTE: application must eventually free key and data

while (***FDFNextEnumeratedObject(***

***thd\_state, // FDF per-thread context***

***iterator, // pointer to iterator context***

***&key, // returns key***

***&key\_size, // returns key length***

***&data, // returns data***

***&data\_size // returns data length***

***)*** == FDF\_SUCCESS)

{

…

}

if (status != FDF\_OBJECT\_UNKNOWN) {

…

}

//\*\*\*\*\*\*\*\*\*\* indicate end of enumeration \*\*\*\*\*\*\*\*\*\*\*

// This should ALWAYS be called when the application

// has finished an enumeration (whether or not

// FDFNextEnumeratedObject() has exhausted all objects).

// It resets internal iteration state and frees memory

// for the iterator.

status = ***FDFFinishEnumeration(***

***thd\_state, // FDF per-thread context***

***iterator); // container id***

if (status != FDF\_SUCCESS) {

...

}

#### Listing 5: Enumerating Container Objects

**FDF\_status\_t FDFEnumerateContainerObjects (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t *cguid*, struct FDF\_iterator \*\**iterator*)**

Start an enumeration of all objects in a container. One or more threads may initiate multiple concurrent enumerations of the same container. The state of each enumeration is identified with an opaque 'iterator' handle that is returned by this function. If the contents of a container change while an enumeration is in progress, the changes may or may not be visible to the enumeration. An object deletion will not be visible if the deleted object is enumerated before the deletion. An object creation may or may not be visible depending on the internal location of the object relative to the internal iteration order.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container id. |
| out | *iterator* | Opaque handle for the iteration context for this enumeration. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_FAILURE otherwise

**FDF\_status\_t FDFNextEnumeratedObject (struct FDF\_thread\_state \**thd\_state*, struct FDF\_iterator \**iterator*, char \*\**key*, uint32\_t \**keylen*, char \*\**data*, uint64\_t \**datalen*)**

Retrieve the next object in an enumeration that was started using **FDFEnumerateContainerObjects()**. The key and data values are returned via FDF-allocated buffers that must be ultimately freed by the application using **FDFFreeBuffer()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *iterator* | Opaque handle for the iteration context for this enumeration. |
| out | *key* | Pointer to FDF allocated buffer holding the key value. |
| out | *keylen* | Length of key in bytes (including any null termination). |
| out | *data* | Pointer to FDF allocated buffer holding object data. |
| out | *datalen* | Length of data in bytes. |

### Returns:

FDF\_SUCCESS if successful

FDF\_OBJECT\_UNKNOWN if the enumeration has completed

FDF\_FAILURE otherwise

**FDF\_status\_t FDFFinishEnumeration (struct FDF\_thread\_state \**thd\_state*, struct FDF\_iterator \**iterator*)**

Terminate an enumeration that was started with **FDFEnumerateContainerObjects()**. All resources that were created for the enumeration are freed. All calls to **FDFEnumerateContainerObjects()** must be matched with a call to **FDFFinishEnumeration()**, whether or not the enumeration was run to completion (ie: **FDFEnumerateContainerObjects()** returned FDF\_OBJECT\_UNKNOWN).

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *iterator* | Opaque handle for the iteration context for this enumeration. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

## Cache Control

If a container is configured for writeback caching, the cache may contain modified data that has not yet been written to storage. The FDFFlushObject(), FDFFlushContainer(), and FDFFlushCache() calls allow an application to force modified data to be written to storage and synced (Listing 6). FDFFlushObject() does this for a single object, FDFFlushContainer() does this for an entire container, and FDFFlushCache() does this for the entire cache.

If the durability level of a container is set to something other than FDF\_FULL\_DURABILITY, the flush operations can be used to make all prior updates to the container durable (whether or not the container is configured for writeback mode).

//\*\*\*\*\*\*\*\*\*\* flush object from cache and sync \*\*\*\*\*\*\*\*\*\*\*

FDF\_status\_t status;

struct FDF\_thread\_state \*thd\_state;

FDF\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

...

status = ***FDFFlushObject(***

***thd\_state, // FDF per-thread context***

***cguid, // container id***

***objkey, // object key***

***keylen // key length***

***);***

if (status == FDF\_OBJECT\_UNKNOWN) {

...

} else if (status != FDF\_SUCCESS) {

...

}

//\*\*\*\*\*\*\*\*\*\* flush container from cache and sync \*\*\*\*\*\*\*\*\*\*\*

status = ***FDFFlushContainer(***

***thd\_state, // FDF per-thread context***

***cguid // container id***

***);***

if (status == FDF\_CONTAINER\_UNKNOWN) {

...

} else if (status != FDF\_SUCCESS) {

...

}

//\*\*\*\*\*\*\*\*\*\* flush cache and sync \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

status = ***FDFFlushCache(***

***thd\_state, // FDF per-thread context***

***);***

if (status == FDF\_SUCCESS) {

...

}

#### Listing 6: Controlling the Cache

**FDF\_status\_t FDFFlushObject (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, char \**key*, uint32\_t*keylen*)**

Force any buffered changes to an object to storage and sync storage.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |
| in | *key* | Key value as an array of bytes. |
| in | *keylen* | Length of key in bytes (including any null termination). |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_OBJECT\_UNKNOWN if the object does not exist

FDF\_FAILURE otherwise

**FDF\_status\_t FDFFlushContainer (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*)**

Force any buffered changes to a container to storage and sync storage.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container identifier. |

### Returns:

FDF\_SUCCESS if successful

FDF\_CONTAINER\_UNKNOWN if the container does not exist

FDF\_FAILURE otherwise

**FDF\_status\_t FDFFlushCache (struct FDF\_thread\_state \**thd\_state*)**

Force any buffered changes to any container to storage and sync storage.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

## Statistics

FDF maintains an extensive set of counters for monitoring events within the FDF subsystem. Most events are counted on a per-container basis as well as for all containers within the FDF instance. FDFGetContainerStats() and FDFGetStats() are used to retrieve these statistics (Listing 7). The same statistics structure is used for both calls. The Appendix provides detailed lists of the counters that are returned in FDF\_stats\_t (Table 4). This structure includes histograms of key and data sizes, and access times for each application access type. Table 5 lists the fields within the FDF\_histo\_t structure.

//\*\*\*\*\*\*\*\*\*\* get FDF stats \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\* Types for FDF stats (from fdf.h) \*/

typedef struct {

uint64\_t n\_accesses[FDF\_N\_ACCESS\_TYPES];

uint64\_t cache\_stats[FDF\_N\_CACHE\_STATS];

uint64\_t flash\_stats[FDF\_N\_FLASH\_STATS];

FDF\_histo\_t key\_size\_histo;

FDF\_histo\_t data\_size\_histo;

FDF\_histo\_t access\_time\_histo[FDF\_N\_ACCESS\_TYPES];

} FDF\_stats\_t;

...

FDF\_status\_t status;

struct FDF\_thread\_state \*thd\_state;

FDF\_stats\_t FDF\_stats;

...

status = ***FDFGetStats(***

***thd\_state, // FDF per-thread context***

***&FDF\_stats // returns FDF stats***

***);***

if (status != FDF\_SUCCESS) {

...

}

. . .

//\*\*\*\*\*\*\*\*\*\* get FDF per-container stats \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FDF\_stats\_t ctnr\_stats;

FDF\_cguid\_t cguid;

...

status = ***FDFGetContainerStats(***

***thd\_state, // FDF per-thread context***

***cguid, // container id***

***&ctnr\_stats // returns container stats***

***);***

if (status != FDF\_SUCCESS) {

...

}

#### Listing 7: Retrieving FDF Statistics

**FDF\_status\_t FDFGetStats (struct FDF\_thread\_state \**thd\_state*, FDF\_stats\_t \**stats*)**

Retrieve global access statistics for this FDF instance. The statistics are returned via an application-allocated statistics structure.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in,out | *stats* | Pointer to structure in which to return the statistics. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

**FDF\_status\_t FDFGetContainerStats (struct FDF\_thread\_state \**thd\_state*, FDF\_cguid\_t*cguid*, FDF\_stats\_t \**stats*)**

Retrieve access statistics for a particular container. The statistics are returned via an application-allocated statistics structure.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread FDF state. |
| in | *cguid* | Container id. |
| in,out | *stats* | Pointer to structure in which to return the statistics. |

### Returns:

FDF\_SUCCESS if successful

FDF\_FAILURE otherwise

|  |  |  |
| --- | --- | --- |
| **FDF\_stats\_t Fields** | | |
| **Name** | **Type** | **Description** |
| n\_accesses[FDF\_N\_ACCESS\_TYPES] | uint64\_t | Counts of FDF access types (see Table 5) |
| flash\_stats[FDF\_N\_FLASH\_STATS] | uint64\_t | Counts of various flash activities (see Table 6) |
| cache\_stats[FDF\_N\_CACHE\_STATS] | uint64\_t | Counts of various cache activities (see Tables 7-12) |
| key\_size\_histo | FDF\_histo\_t | Base-2 logarithmic histogram of key sizes in bytes |
| data\_size\_histo | FDF\_histo\_t | Base-2 logarithmic histogram of data sizes in bytes |
| access\_time\_histo[FDF\_N\_ACCESS\_TYPES] | FDF\_histo\_t | Base-2 logarithmic histogram of access latencies in microseconds |

#### Table 4: Field Definitions for FDF Statistics Structure

|  |  |  |
| --- | --- | --- |
| **FDF\_histo\_t Fields** | | |
| **Name** | **Type** | **Description** |
| n | uint64\_t | Number of events |
| min | uint64\_t | Minimum value |
| max | uint64\_t | Maximum value |
| avg | double | Average value |
| geo | double | Geometric mean |
| std | double | Standard deviation |
| counts[64] | uint64\_t | Counts of the Base-2 logarithm of the event values.  For example, counts[23] provides the number of events with values in the range [2^23, 2^24). |

#### Table 5: Field Definitions for FDF Statistics Structure

# Chapter 3: “Hello World” with FDF

Listing 8 shows a simple example of a complete FDF session:

* Initializing FDF
* Creating a container
* Writing several objects
* Reading an object
* Enumerating all objects
* Deleting an object
* Closing and deleting a container
* Terminating FDF

Note that error checking is omitted for clarity!

#include <fdf.h>

main( )

{

FDF\_container\_props\_t props;

struct FDF\_state \*fdf\_state;

struct FDF\_thread\_state \*thd\_state;

struct FDF\_iterator \*iterator;

FDF\_status\_t status;

FDF\_cguid\_t cguid;

char \*key,

\*data;

uint32\_t keylen;

uint64\_t datalen;

FDFLoadProperties( "/schooner/backup/sample.prop");

FDFInit( &fdf\_state);

FDFInitPerThreadState( fdf\_state, &thd\_state);

FDFLoadCntrPropDefaults( &props);

status = FDFOpenContainer( thd\_state, "cntr1", &props, FDF\_CTNR\_CREATE, &cguid);

FDFWriteObject( thd\_state, cguid, "key1", 5, "key1\_data", 10, 0);

FDFWriteObject( thd\_state, cguid, "key2", 5, "key2\_data", 10, 0);

FDFWriteObject( thd\_state, cguid, "key3", 5, "key3\_data", 10, 0);

FDFReadObject( thd\_state, cguid, "key2", 5, &data, &datalen);

printf( "sdf\_get: data=%s, datalen=%ld\n", data, datalen);

FDFFreeBuffer( data);

status = FDFEnumerateContainerObjects( thd\_state, cguid, &iterator);

while (FDFNextEnumeratedObject( thd\_state, iterator, &key, &keylen, &data, &datalen) ==

FDF\_SUCCESS) {

printf( "sdf\_enum: key=%s, keylen=%d, data=%s, datalen=%ld\n", key, keylen, data,

datalen);

FDFFreeBuffer( key);

FDFFreeBuffer( data);

}

FDFFinishEnumeration( thd\_state, iterator);

FDFDeleteObject( thd\_state, cguid, "key2", 5);

FDFCloseContainer( thd\_state, cguid);

FDFDeleteContainer( thd\_state, cguid);

FDFReleasePerThreadState( &thd\_state);

FDFShutdown( fdf\_state);

return (0);

}

#### Listing 8: “Hello World” with FDF

# Chapter 4: Compiling with FDF

The FDF development kit provides an fdf.h file that must be included in application files that use FDF, and a library file libfdfdll.a that must be linked with the application. At runtime the full FDF library is dynamically loaded from one of these locations in the order shown:

/usr/lib64/fdf/libfdf.so

/use/lib/fdf/libfdf.so

/lib64/libfdf.so

/lib/libfdf.so

/usr/local/lib64/libfdf.so

/usr/local/lib/libfdf.so

If the library is not in one of the above locations, LD\_LIBRARY\_PATH is searched to find libfdf.so.

Alternatively, if you set the environment variable FDF\_LIB, the application will only look for the library there.

See the Release Notes for details concerning supported platforms.

# Appendix: FDF Statistics

Tables 5 through 12 list the counts that are available in the FDF\_stats\_t structure that can be retrieved per container (FDFGetContainerStats()) or for all containers (FDFGetStats()):

* Table 5 lists the enum values for indexing the n\_accesses[] array.
* Table 6 lists the enum values for indexing the flash\_stats[] array.
* Tables 7 – 12 list the enum values for indexing the cache\_stats[] array:
  + Table 7 lists cache overwrite, writeback and write-through counts
  + Table 8 lists counts of requests from the DRAM cache to the flash manager
  + Table 9 lists counts of responses from the flash manager to the DRAM cache
  + Table 10 lists counts of requests and responses between the flash manager and flash storage
  + Table 11 lists counts of the different return codes encountered at the lowest level flash interface
  + Table 12 lists miscellaneous cache statistics, including counts of various flushing events, and cache occupancy in bytes and objects

|  |  |
| --- | --- |
| **FDF Requests from Application (AP---)** | |
| **Name** | **Description** |
| APCOE | create object with expiry |
| APCOP | create object and put data (no expiry) |
| APPAE | put object with expiry |
| APPTA | put object (no expiry) |
| APSOE | set object with expiry |
| APSOB | set object (no expiry) |
| APGRX | get object and check expiry |
| APGRD | get object with no expiry check |
| APDBE | delete object with expiry |
| APDOB | delete object (no expiry) |
| APFLS | flush object |
| APFLI | flush and invalidate object |
| APINV | invalidate object |
| APSYC | sync container to flash |
| APICD | invalidate container delayed (memcached flush-all) |
| APGIT | get invalidation time from last APICD |
| APFCO | flush container |
| APFCI | flush and invalidate container |
| APICO | invalidate container |
| APRIV | remote object invalidate (via replication) |
| APRUP | remote object update (via replication) |

#### Table 5: FDF Statistics: FDF Requests

|  |  |
| --- | --- |
| **Flash Statistics** | |
| **Name** | **Description** |
| num\_objects | Number of objects in flash |
| num\_created\_objects | Total number of created objects |
| num\_evictions | Number of flash evictions |
| get\_hash\_collisions | Number of hash collisions for get operations |
| set\_hash\_collisions | Number of hash collisions for set operations |
| num\_overwrites | Number of overwrites |
| num\_ops | Number of all flash operations |
| num\_read\_ops | Number of read operations |
| num\_get\_ops | Number of get operations |
| num\_put\_ops | Number of put operations |
| num\_del\_ops | Number of delete operations |
| get\_exist\_checks | Number of existence checks for gets |
| num\_full\_buckets | Number of full hash buckets |
| pending\_ios | Number of pending IO’s |
| space\_allocated | Flash space allocated in bytes |
| space\_consumed | Flash space consumed in bytes |

#### Table 6: FDF Flash Statistics

|  |  |
| --- | --- |
| **FDF Overwrite/Write-Through Statistics** | |
| **Name** | **Description** |
| overwrites\_s | number of overwrites of objects in the S state |
| overwrites\_m | number of overwrites of objects in the M state |
| inplaceowr\_s | number of in-place overwrites of objects in the S state |
| inplaceowr\_m | number of in-place overwrites of objects in the M state |
| new\_entries | number of newly created cache objects (as opposed to overwrites) |
| writethrus | number of write-throughs to flash |
| writebacks | number of write-backs (of dirty objects) to flash |
| flushes | number of flush operations to flash (includes background flushes) |

#### Table 7: FDF Statistics: Overwrites and Write-throughs

|  |  |
| --- | --- |
| **Action Entity (Cache) To Home Entity (Flash Manager)**  **Requests (AH---)** | |
| **Name** | **Description** |
| AHCOB | create object |
| AHCOP | create object and put data |
| AHCWD | castout with data |
| AHDOB | delete object |
| AHFLD | flush object |
| AHGTR | get object with intent to read |
| AHGTW | get object with intent to write |
| AHPTA | put object |
| AHSOB | set object |
| AHSOP | set object put |

#### Table 8: FDF Statistics: Requests from Cache to Flash Manager

|  |  |
| --- | --- |
| **Home Entity (Flash Manager) To Action Entity (Cache)**  **Responses (HA---)** | |
| **Name** | **Description** |
| HACRC | create object completed |
| HACRF | create object failed |
| HACSC | castout completed |
| HACSF | castout failed |
| HADEC | delete object completed |
| HADEF | delete object failed |
| HAFLC | flush object completed |
| HAFLF | flush object failed |
| HAGRC | get object to read completed |
| HAGRF | get object to read failed |
| HAGWC | get object to write completed |
| HAGWF | get object to write failed |
| HAPAC | put object completed |
| HAPAF | put object failed |
| HASTC | set object completed |
| HASTF | set object failed |

#### Table 9: FDF Statistics: Responses from Flash Manager to Cache

|  |  |
| --- | --- |
| **Flash Requests and Responses to/from Flash Manager**  **(HF--- and FH---)** | |
| **Name** | **Description** |
| HFXST | check existence of object |
| FHXST | existence check succeeded |
| FHNXS | existence check failed |
| HFGFF | get object |
| FHDAT | object data |
| FHGTF | get object failed |
| HFPTF | put object |
| FHPTC | put object completed |
| FHPTF | put object failed |
| HFDFF | delete object |
| FHDEC | delete object completed |
| FHDEF | delete object failed |
| HFCIF | create object |
| FHCRC | create object completed |
| FHCRF | create object failed |
| HFCZF | create zeroed object |
| FHCRC | create zeroed object completed |
| FHCRF | create zeroed object failed |
| HFSET | set object |
| FHSTC | set object completed |
| FHSTF | set object failed |
| HFCSH | create shard |
| FHCSC | create shard completed |
| FHCSF | create shard failed |
| HFSSH | sync shard |
| FHSSC | sync shard completed |
| FHSSF | sync shard failed |
| HFDSH | delete shard |
| FHDSC | delete shard completed |
| FHDSF | delete shard failed |
| HFGLS | get last sequence number |
| FHGLC | get last sequence number completed |
| FHGLF | get last sequence number failed |
| HFGIC | get iteration cursors |
| FHGIC | get iteration cursors completed |
| FHGIF | get iteration cursors failed |
| HFGBC | get by cursor |
| FHGCC | get by cursor completed |
| FHGCF | get by cursor failed |
| HFGSN | get sequence number |
| HFGCS | get container stat |
| FHGSC | get container stat completed |
| FHGSF | get container stat failed |
| HFSRR | start replicating |
| FHSRC | start replicating completed |
| FHSRF | start replicating failed |
| HFSPR | stop replicating |
| FHSPC | stop replicating completed |
| FHSPF | stop replicating failed |
| HFFLA | flush object |
| FHFLC | flush object completed |
| FHFLF | flush object failed |
| HFRVG | release vip group |
| FHRVC | release vip group completed |
| FHRVF | release vip group failed |
| HFNOP | noop |
| FHNPC | noop completed |
| FHNPF | noop failed |
| HFOSH | open shard |
| FHOSC | open shard completed |
| FHOSF | open shard failed |
| HFFLS | flush object |
| FHFCC | flush object completed |
| FHFCF | flush object failed |
| HFFIV | flush invalidate object |
| FHFIC | flush invalidate object completed |
| FHFIF | flush invalidate object failed |
| HFINV | invalidate object |
| FHINC | invalidate object completed |
| FHINF | invalidate object failed |
| HFFLC | flush container |
| FHLCC | flush container completed |
| FHLCF | flush container failed |
| HFFLI | flush invalidate container |
| FHLIC | flush invalidate container completed |
| FHLIF | flush invalidate container failed |
| HFINC | invalidate container |
| FHCIC | invalidate container completed |
| FHCIF | invalidate container failed |

#### Table 10: FDF Statistics: Requests and Responses Between Flash Manager and Flash

|  |  |
| --- | --- |
| **Counts of Low-level Flash Return Codes** | |
| **Name** | **Description** |
| EOK | success |
| EPERM | not permitted |
| ENOENT | not found |
| EDATASIZE | user-supplied data buffer is too small |
| ESTOPPED | container is stopped |
| EBADCTNR | container does not exist |
| EDELFAIL | deletion for a local failure failed |
| EAGAIN | try again (transient error) |
| ENOMEM | out of memory |
| EACCES | permission denied |
| EINCONS | inconsistency during replication |
| EBUSY | device busy |
| EEXIST | object exists |
| EINVAL | invalid argument |
| EMFILE | too many objects |
| ENOSPC | out of flash space |
| ENOBUFS | out of system resource |
| ESTALE | stale data |
| EDQUOT | quota exceeded |
| RMT\_EDELFAIL | deletion for a remote failure failed |
| RMT\_EBADCTNR | container does not exist on a remote node |

#### Table 11: FDF Statistics: Flash Access Return Codes

|  |  |
| --- | --- |
| **Miscellaneous Per-Cache Statistics** | |
| **Name** | **Description** |
| hashBuckets | number of hash buckets in cache directory |
| nSlabs | number of cache partitions |
| numElements | number of objects in the cache |
| maxSz | maximum capacity of cache, in bytes |
| currSz | number of bytes that contain data in the cache (does not include keys) |
| currSzWkeys | number of bytes containing data and keys |
| nMod | number of modified objects in the cache (flash does NOT have the latest data) |
| modSzWkeys | number of bytes of modified objects in the cache (including keys) |
| nModFlushes | number of modified objects that have been flushed to flash via cache or container flushes (not including background flushes) |
| nModBGFlushes | number of modified objects that have been flushed to flash by the background flush process |
| nPending | number of pending remote cache requests (due to replication) |
| nModRecEnums | number of modified cache objects copied during the recovery process |
| bkFlshProg | progress of the current background flush cycle (percent) |
| nBkFlsh | number of times the background flusher has cycled through the cache |
| nFlshTok | total number of flushes that can occur in parallel (includes explicit and background flushes) |
| nBkFlshTok | total number of background flushes that can occur in parallel (must be <= nFlshTok) |
| FlsMs | time to wait after a background flush cycle in which no dirty data is found |
| modPct | percentage limit on the number of bytes in the cache that are modified |
| nAppBufs | number of application buffers that are in use |
| nTrans | number of cache operations that are in progress |
| nFGBufs | number of flash data buffers currenly being processed |
| nResp | number of response messages currently being processed |

#### Table 12: FDF Statistics: Miscellaneous