**Programming Guide**

SanDisk ZS API

Version 3.0

SanDisk Confidential

Documentation ID: ZS-v3.0-PG-02

# Contents

Contents 3

Chapter 1: Introduction 4

Chapter 2: API 7

Initialization and Shutdown 7

Containers 11

Objects 17

Transactions 25

Range Query 26

Snapshot 31

Cache Control 32

Statistics 34

Miscellaneous 37

Chapter 3: ZS Administration 38

Configuration 38

License 38

Logging 38

Stats 39

Command Line Interface(CLI) 40

ZS Metadata Validation (ZSCK) 41

Chapter 4: ZS Package 47

Chapter 5: Compiling with ZS 48

Chapter 6: “Hello World” with ZS 49

Appendix: ZS Statistics 51

Appendix: ZS Storm Mode 58

# Chapter 1: Introduction

The SanDisk ZetaScale API (ZS) is a substrate for flash-optimized data storage solutions, including caches, key-value stores and databases. ZS 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 in operating costs.

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

The system level optimizations in ZS 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).

In addition to these features, the ZS 3.0 release supports very large storage systems in the petabyte range. See the Appendix: ZS Storm Mode for details.

Figure 1 shows the high level architecture of ZS. ZS 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 ZS are:

* Container Manager
  + ZS 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
  + ZS objects are subject to the policies of their owner containers
  + The ZS 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 ZS 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 ZS flash manager is responsible for shard allocation, replication and recovery
* Replication Layer (roadmap item)
  + ZS 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 ZS elasticity module implements automatic sharding for seamlessly expanding (or shrinking) the number of nodes serving particular containers.
* Messaging Subsystem and Transport Layers
  + The ZS messaging service provides high-speed, robust communications for inter-node ZS messages, remote object access, replication and sharding.
* Threading Module (roadmap item)
  + In addition to pthreads, ZS offers an optional user-mode lightweight threading module that maximizes concurrency by minimizing operating system calls.

Container Mgt

Naming, create, open, delete

ZS 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 1: ZS Architecture

The remainder of this document describes the ZS programming interface. Chapter 2 provides coding examples and detailed descriptions for all ZS functions. Chapter 3 summarizes the ZS administration. Chapter 4 describes the contents of the ZS package. Chapter 5 describes how to compile ZS into an application. Chapter 6 summarizes the key routines of Chapter 2 with a short “Hello, world” program that starts ZS, creates a container, reads, writes and enumerates some objects, and shuts ZS down. ZS 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 ZS has 3 steps (Listing 1). First, ZS configuration parameters must be initialized if values other than defaults are required. Next, ZSInit() must be called to initialize global ZS state. Third, each thread that uses ZS must initialize per-thread state using ZSInitPerThreadState().

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

Figure 2 shows a sample configuration file; the configuration file and ZSSetProperty() use the same name for each parameter. ZSInit() uses the property values to initialize ZS global state. Each application thread that will call ZS functions must then allocate a per-thread ZS context using ZSInitPerThreadState(). The opaque context pointer provided by ZSInitPerThreadState() is required for almost all ZS function calls.

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

// ZS Initialization

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

// Load some properties:

if (***ZSLoadProperties(“/opt/sandisk/config/ZS\_defaults.prop”)*** != ZS\_SUCCESS) {

…

}

// Set other properties as desired

ZSSetProperty(“ZS\_REFORMAT”, “1”); // Do not recover persistent containers

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

ZSSetProperty(“ZS\_FLASH\_FILENAME”, “/mnt/ssd/ZS\_flash%d”; // location of flash device/files

…

if (***ZSInit(&state)*** != ZS\_SUCCESS) {

error( "ZSInit() failed" );

}

. . .

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

// Put this code where a pthread is initialized

struct ZS\_state \*ZS\_state;

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

ZS\_status\_t status;

// ZS\_state created by ZSInit() above

status = ***ZSInitPerThreadState(ZS\_state, &thd\_state)***;

assert(status == ZS\_SUCCESS);

#### Listing 1: Initializing ZS

// ZS Termination

// Delete Per-thread ZS Context

// thd\_state created by ZSInitPerThreadState() above

if (***ZSReleasePerThreadState( &thd\_state)*** != ZS\_SUCCESS) {

error( "ZSReleasePerThreadState() failed" );

}

. . .

// ZS\_state allocated using ZSInit() above

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

// state handles have been released!

if (***ZSShutdown(ZS\_state)*** != ZS\_SUCCESS) {

error( "ZSShutdown() failed" );

}

#### Listing 2: Terminating ZS

**ZS\_status\_t ZSLoadProperties (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 ZSLoadProperties() or ZSSetProperty().

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE if property file could not be loaded

ZS\_INVALID\_PARAMETER if property file name is invalid

**ZS\_status\_t ZSSetProperty (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:

ZS\_SUCCESS if successful

ZS\_FAILURE if property could not be set

|  |  |  |  |
| --- | --- | --- | --- |
| **ZS Configuration Properties** | | | |
| **Property Name** | **Type** | **Default** | **Description** |
| ZS\_FLASH\_FILENAME | char \* | /tmp/sandisk0 | Path to flash device. This should be a link to the actual flash device. |
| ZS\_FLASH\_SIZE | uint32\_t | 200 | Total amount of flash in GB. |
| ZS\_CACHE\_SIZE | uint64\_t | 100000000 | Memory reserved for DRAM cache in bytes |
| ZS\_REFORMAT | uint32\_t | 0 | If 1, delete the contents of all persistent containers (instead of recovering them). |
| ZS\_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. |
| ZS\_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. |
| ZS\_ASYNC\_PUT\_THREADS | uint32\_t | 32 | Size of the thread pool that performs asynchronous writeback and flush operations. |
| ZS\_MODIFIED\_FRACTION | double | 1.0 | The maximum fraction of cache space that can hold modified data. If this is exceeded, ZS performs flushes of LRU dirty objects until it drops below the threshold. |
| ZS\_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 ZS Configuration Structure

# Sample Property File

ZS\_FLASH\_FILENAME = “/opt/sandisk/flash/ssd0”

ZS\_FLASH\_SIZE = 100

ZS\_CACHE\_SIZE = 40000000000

ZS\_LICENSE\_PATH = /opt/ZSlicense

#### Figure 2: Sample Property File for ZS Initialization

**ZS\_status\_t ZSInit (struct ZS\_state \*\**ZS\_state*)**

Initialize ZS global state. This function should be called exactly once, and must be called before any other ZS function is called (except **ZSLoadProperties()** or **ZSSetProperty()**).

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_LICENSE\_CHK\_FAILED if license is invalid or not present

ZS\_FAILURE otherwise

**ZS\_status\_t ZSInitPerThreadState (struct ZS\_state \**ZS\_state*, struct ZS\_thread\_state \*\**thd\_state*)**

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSReleasePerThreadState (struct ZS\_thread\_state \*\**thd\_state*)**

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSShutdown (struct ZS\_state \**ZS\_state*)**

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

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *ZS\_state* | Opaque handle for ZS global state previously acquired in **ZSInit()**. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

## Containers

ZS objects are manipulated within containers. Containers are created with the ZSOpenContainer() function (Listing 3). The ZS 3.0 release supports up to 65K containers. Container properties are specified in the structure ZS\_container\_props\_t, which is defined in ZS.h. Table 3 lists the various properties that can be configured, including size and durability level. For convenience, ZS\_container\_props\_t can be preloaded with default values using ZSLoadCntrPropDefaults().

ZSOpenContainer() returns a container identifier, the cguid, that is used to identify the container in subsequent ZS function calls. The cguid remains valid until the container is closed using ZSCloseContainer(). Containers may be deleted by calling ZSDeleteContainer(). 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 ZSGetContainers(). The properties of an open container can be retrieved using ZSGetContainerProps(). Some container properties can be changed dynamically using ZSSetContainerProps(). Table 3 shows which properties can be altered dynamically.

// ZS Containers

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

typedef struct {

uint64\_t size\_kb;

ZS\_durability\_level\_t durability\_level;

…

} ZS\_container\_props\_t;

...

struct ZS\_thread\_state \*thd\_state;

char \*cname;

ZS\_container\_props\_t props;

ZS\_cguid\_t cguid;

ZS\_status\_t status;

uint32\_t flags;

...

// Load default properties

if (***ZSLoadCntrPropDefaults(&props)*** != ZS\_SUCCESS) {

…

}

// Change default properties as desired.

props.size\_kb = 20000;

props.durability\_level = ZS\_DURABILITY\_SW\_CRASH\_SAFE;

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

// (read/write access is default).

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

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

// an error if the container does not exist.

status = ***ZSOpenContainer(***

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

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

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

***flags, // flags***

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

***);***

if (status != ZS\_SUCCESS) {

...

}

. . .

status = ***ZSCloseContainer(***

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

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

***);***

if (status != ZS\_SUCCESS) {

...

}

status = ***ZSDeleteContainer(***

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

***cguid // container id***

***);***

if (status != ZS\_SUCCESS) {

...

}

#### Listing 3: Manipulating Containers

**ZS\_status\_t ZSLoadCntrPropDefaults (ZS\_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:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

|  |  |  |  |
| --- | --- | --- | --- |
| **ZS\_container\_props\_t Fields** | | | |
| **Name** | **Type** | **Can Change Dynamically?** | **Description** |
| size\_kb | uint64\_t | yes | Size of the container in kB. Size can only be increased dynamically. |
| durability\_level | ZS\_durability\_level\_t | No | Set the durability policy for the container:  ZS\_DURABILITY\_PERIODIC: sync storage every 1024 writes. Up to 1024 objects can be lost in the event of application crash  ZS\_DURABILITY\_SW\_CRASH\_SAFE: This policy guarantees no data loss in the event of software crashes. But some data might be lost in the event of hardware failure.  ZS\_DURABILITY\_HW\_CRASH\_SAFE: This policy guarantees no data loss if the hardware crashes. |

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

**ZS\_status\_t ZSOpenContainer (struct ZS\_thread\_state \**thd\_state*, char \**cname*, ZS\_container\_props\_t \**props*, uint32\_t *flags*, ZS\_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 |= ZS\_CTNR\_RO\_MODE) opens the container in read-only mode. Setting the create flag (flag |= ZS\_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 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 ZS 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:

ZS\_SUCCESS if successful

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

ZS\_CONTAINER\_EXISTS if the container already exists

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

ZS\_FAILURE\_OPERATION\_DISALLOWED if system is shutting down

ZS\_LICENSE\_CHK\_FAILED if license is invalid or missing

ZS\_INVALID\_PARAMETER if a parmeter is invalid or missing

ZS\_FAILURE\_INVALID\_CONTAINER\_SIZE if the size is greater than total storage size

ZS\_UNLIMITED\_CONTAINER\_MUST\_BE\_NON\_EVICTING unlimited must be non-evicting

ZS\_PUT\_META\_FAILED if container metadata cannot be stored

ZS\_FAILURE\_MEMORY\_ALLOC if memory cannot be allocated

ZS\_FAILURE otherwise

**ZS\_status\_t ZSOpenContainerSpecial (struct ZS\_thread\_state \**thd\_state*, char \**cname*, ZS\_container\_props\_t \**props*, uint32\_t *flags*, ZS\_container\_meta\_t \*cmeta, ZS\_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 |= ZS\_CTNR\_RO\_MODE) opens the container in read-only mode. Setting the create flag (flag |= ZS\_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 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 ZS 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. |
| In | *cmeta* | Meta data for specifying various comparator callback functions |
| out | *cguid* | Container identifier to use in subsequent container operations. |

### Returns:

ZS\_SUCCESS if successful

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

ZS\_CONTAINER\_EXISTS if the container exists already

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

ZS\_FAILURE otherwise

**ZS\_status\_t ZSCloseContainer (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*)**

Close the specified container.

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_CONTAINER\_UNKNOWN if the container does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSDeleteContainer (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*)**

Delete the specified container.

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_CONTAINER\_UNKNOWN if the container does not exist

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

ZS\_FAILURE otherwise

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

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSGetContainerProps (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t*cguid*, ZS\_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 ZS state. |
| in | *cguid* | Container id. |
| in,out | *props* | Pointer to property structure in which to return properties. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSSetContainerProps (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*, ZS\_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 ZS state. |
| in | *cguid* | Container id. |
| in | *props* | Pointer to property structure. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

## Objects

Objects are read, written and deleted to/from an open container using the ZSReadObject(), ZSWriteObject(), and ZSDeleteObject() calls (Listing 4). The ZSReadObject() call returns object data in an ZS-allocated buffer. The application is responsible for freeing this buffer using ZSFreeBuffer().

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

ZSNextEnumeratedObject(), and ZSFinishEnumeration() (Listing 5). The keys and data for enumerated objects are returned via ZS-allocated buffers, which must be subsequently freed using ZSFreeBuffer().

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

ZS\_status\_t status;

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

ZS\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

char \*pdata;

uint64\_t \*pdatalen;

...

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

// Application must later release the buffer using

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

status = ***ZSReadObject(***

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

***cguid, // cguid***

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

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

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

***pdatalen // object size***

***);***

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

...

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

...

}

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

ZS\_status\_t status;

struct ZS\_thread\_state \*thd\_state;

ZS\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

uint64\_t datalen;

char \*pdata;

uint32\_t flags;

...

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

… OR …

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

… OR …

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

status = ***ZSWriteObject(***

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

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

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

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

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

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

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

***);***

if (status != ZS\_SUCCESS) {

...

}

#### Listing 4: Manipulating Objects

**ZS\_status\_t ZSReadObject (struct ZS\_thread\_state \**thd\_state*, ZS\_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 ZS. The application must subsequently free this buffer using **ZSFreeBuffer()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS 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 ZS allocated buffer holding object data. |
| out | *datalen* | Length of data in bytes. |

### Returns:

ZS\_SUCCESS if successful

ZS\_INVALID\_PARAMETER if the container identifier is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSReadObjectExpiry (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*, ZS\_readobject\_t \*robj)**

Read an object including its expiry time from a container. The object is returned via a buffer that is allocated by ZS and the expiry time is returned via user supplied buffer. The application must subsequently free this buffer using **ZSFreeBuffer()**.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container identifier. |
| in, out | *robj* | Pointer to a ZS\_readobject\_t structure to hold given key and object data, expiry time. The definition of ZS\_readobject\_t is given below.  typedef struct {  char \*key; /\* Application supplied buffer which holds key \*/  uint32\_t key\_len; /\* Length of the key \*/  char \*data; /\* ZS allocated buffer which holds data \*/  uint64\_t data\_len; /\* Length of the data \*/  ZS\_time\_t current; /\* Current time. Not used in this API \*/  ZS\_time\_t expiry; /\* Expiry time of the obect filled bu ZS \*/  } ZS\_readobject\_t; |

### Returns:

ZS\_SUCCESS if successful

ZS\_INVALID\_PARAMETER if the container identifier is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSFreeBuffer (char \**buf*)**

Frees a buffer that was allocated in ZS and returned via **ZSReadObject()** or **ZSNextEnumeratedObject()**.

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSWriteObject (struct ZS\_thread\_state \**thd\_state*, ZS\_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 |= ZS\_WRITE\_MUST\_NOT\_EXIST), the write succeeds only if the object does NOT already exist. If the non-existence flag is set (flag |= ZS\_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 ZS 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). |
| in | *data* | Pointer to the data to be written. |
| in | *datalen* | Length of data in bytes. |

### Returns:

ZS\_SUCCESS if successful

ZS\_INVALID\_PARAMETER if the container identifier is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_FLASH\_ENOSPC if the container is out of space

ZS\_CONTAINER\_UNKNOWN if the container does not exist

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

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

ZS\_FAILURE otherwise

**ZS\_status\_t ZSWriteObjectExpiry (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*, ZS\_writeobject\_t \*wobj, uint32\_t *flags*)**

Write an object with expiry time to a container. The object becomes invalid automatically after the specified expiry time. 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 |= ZS\_WRITE\_MUST\_NOT\_EXIST), the write succeeds only if the object does NOT already exist. If the non-existence flag is set (flag |= ZS\_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 ZS state. |
| in | *cguid* | Container identifier. |
| in | *wobj* | Pointer to ZS\_writeobject\_t to hold key, data and expiry time.  The ZS\_writeobject\_t is defined as below  typedef struct {  char \***key; /**\* User supplied buffer which holds key \*/  uint32\_t key\_len; /\* Key size in bytes + NULL termination\*/  char \*data; /\* User supplied buffer which holds the data \*/  uint64\_t data\_len; /\* Length of the data in bytes \*/  ZS\_time\_t current; /\* Current time \*/  ZS\_time\_t expiry; /\*Expiry time of the object \*/  } ZS\_w |
| in | *flags* | Write operation control flags |

### Returns:

ZS\_SUCCESS if successful

ZS\_INVALID\_PARAMETER if the container identifier is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_FLASH\_ENOSPC if the container is out of space

ZS\_CONTAINER\_UNKNOWN if the container does not exist

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

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

ZS\_FAILURE otherwise

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

Delete an object from a container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS 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:

ZS\_SUCCESS if successful

ZS\_INVALID\_PARAMETER if the container identifier is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_CONTAINER\_UNKNOWN if the container does not exist

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

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

ZS\_status\_t status;

struct ZS\_thread\_state \*thd\_state;

ZS\_cguid\_t cguid;

char \*key;

char \*data;

uint32\_t key\_size;

uint64\_t data\_size;

struct ZS\_iterator \*iterator;

...

status = ***ZSEnumerateContainerObjects(***

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

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

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

if (status != ZS\_SUCCESS) {

…

}

// NOTE: application must eventually free key and data

while (***ZSNextEnumeratedObject(***

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

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

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

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

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

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

***)*** == ZS\_SUCCESS)

{

…

}

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

…

}

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

// This should ALWAYS be called when the application

// has finished an enumeration (whether or not

// ZSNextEnumeratedObject() has exhausted all objects).

// It resets internal iteration state and frees memory

// for the iterator.

status = ***ZSFinishEnumeration(***

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

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

if (status != ZS\_SUCCESS) {

...

}

#### Listing 5: Enumerating Container Objects

**ZS\_status\_t ZSEnumerateContainerObjects (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*, struct ZS\_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 ZS state. |
| in | *cguid* | Container id. |
| out | *iterator* | Opaque handle for the iteration context for this enumeration. |

### Returns:

ZS\_SUCCESS if successful

ZS\_CONTAINER\_UNKNOWN if the container does not exist

ZS\_FAILURE otherwise

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

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_OBJECT\_UNKNOWN if the enumeration has completed

ZS\_FAILURE otherwise

**ZS\_status\_t ZSFinishEnumeration (struct ZS\_thread\_state \**thd\_state*, struct ZS\_iterator \**iterator*)**

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

## Transactions

A transaction identifies changes to objects that are to be made persistent in their entirety, or not at all. Participating objects may be added individually or in groups by such calls as ZSWriteObject() and ZSMPut(), but not by container-level calls. The transaction is concluded by a call to commit, or by a crash. Crashing reverts object changes by all uncommitted transactions.

Transaction facilities are controlled by property ZS\_TRX, and initialized at ZS start. With transactions disabled (ZS\_TRX=0), the application enjoys marginally higher performance, but container contents are at risk if the application crashes or the machine suffers a loss of power. Application is required to maintain isolation of object acesses between threads. To ensure complete recovery after a crash, transactions are limited to 100K object operations and must commit promptly.

**ZS\_status\_t ZSTransactionStart (struct ZS\_thread\_state \**thd\_state*)**

Applications can group one or more key/value operations into a transaction. ZSTransactionStart() marks the start of a transaction. Subsequent object creation, deletion and updates will be part of this transaction. Operations within a transaction can span multiple containers. The changes caused by these operations are visible to other application threads while the transaction is in progress, but are not permanent until explicitly committed by the application thread. A thread can be associated with one transaction at most.

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_TRANS\_LEVEL\_EXCEEDED if too many transactions are active

ZS\_OUT\_OF\_MEM if memory is exhausted

ZS\_FAILURE otherwise

**ZS\_status\_t ZSTransactionCommit (struct ZS\_thread\_state \**thd\_state*)**

### Commits the object changes in a transaction to permanent storage. If the application or system crashes before the commit is completed, all of the changes are reverted on ZS restart.

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_NO\_TRANS if thread is not associated with a transaction

ZS\_OUT\_OF\_MEM if memory is exhausted  
ZS\_TRANS\_ABORTED if transaction exceeded the limit on number of allowed key value operations

ZS\_FAILURE if aborted due to conflict with another transaction

\

**uint64\_t ZSTransactionID(struct ZS\_thread\_state \**thd\_state*)**

### Return the ID of the current transaction for this thread.

### Parameters:

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

### Returns:

Non-zero transaction ID on success

Zero if there is no active transaction

## Range Query

The Range query APIs allow applications to retrieve range of objects from a container in an order based on given query attributes. The range query can be based on either a primary index or a secondary index of the container. Listing 6 shows the data structures used to define the query attributes.

**ZS\_status\_t ZSGetRange(struct ZS\_thread\_state \**thd\_state,* ZS\_cguid\_t *cguid*, ZS\_indexid\_t *indexid,* struct ZS\_cursor *\*\*cursor,* ZS\_range\_meta\_t *\*meta)***

This API initializes a range query based on given query attributes. Application calls this API first to start a range query. The API returns a handle upon success. The application will use this handle to iterate one or more items at a time using the API ZSGetRangeNext.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container Id |
| in | *indexid* | Persistent opaque handle for an Index to use for Range (use PRIMARY\_INDEX for primary Index). This handle is defined in ZS.h as uint64\_t ZS\_indexid\_t; |
| in | *meta* | Query Attributes |
| out | *cursor* | Opaque cursor for this query. Application will use this as handle for iterating the objects using ZSGetRange API.  The curor is defined in ZS.h as below  struct ZS\_cursor; |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid

ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_FAILURE otherwise

typedef struct ZS\_range\_meta {

uint32\_t flags; // flags controlling type of range query (see below)

uint32\_t keybuf\_size; // size of application provided key buffers (if applicable)

uint64\_t databuf\_size; // size of application provided data buffers (if applicable)

char \*key\_start; // start key

uint32\_t keylen\_start; // length of start key

char \*key\_end; // end key

uint32\_t keylen\_end; // length of end key

uint64\_t start\_seq; // starting sequence number (if applicable)

uint64\_t end\_seq; // ending sequence number (if applicable)

} ZS\_range\_meta\_t;

typedef enum ZS\_range\_enums\_t {

BUFFER\_PROVIDED = 1<<0, // buffers for keys and data provided by application

ALLOC\_IF\_TOO\_SMALL = 1<<1, // if supplied buffers are too small, ZS will allocate

SEQNO\_LE = 1<<2, // only return objects with seqno <= end\_seq

SEQNO\_GT\_LE = 1<<3, // only return objects with start\_seq < seqno <= end\_seq

START\_GT = 1<<4, // keys must be > key\_start

START\_GE = 1<<5, // keys must be >= key\_start

START\_LT = 1<<6, // keys must be < key\_start

START\_LE = 1<<7, // keys must be <= key\_start

END\_GT = 1<<8, // keys must be > key\_end

END\_GE = 1<<9, // keys must be >= key\_end

END\_LT = 1<<10, // keys must be < key\_end

END\_LE = 1<<11, // keys must be <= key\_end

KEYS\_ONLY = 1<<12, // only return keys (data is not required)

PRIMARY\_KEY = 1<<13, // return primary keys in secondary index query

};

#### Listing 6: Specifying Query Attributes

**ZS\_status\_t ZSGetNextRange(struct ZS\_thread\_state \**thd\_state,* structZS\_cursor *\*cursor,* int *n\_in,* int *\*n\_out,*** **ZS\_range\_data\_t *\*values)***

This API iterates one or more items at a time in a container based on the query attributes. Application calls this API with the cursor returned by the **ZSGetRange** API. This API returns required number of iterated items through application supplied buffer.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cursor* | Cursor for this indexed query |
| in | *n\_in* | size of 'values' array and this parameter defines the maximum number of iterated items to be returned |
| out | *n\_out* | Actual number of items returned |
| out | *values* | Array of returned key/data values. Values is returned in the following data structure defined in ZS.h.  typedef struct ZS\_range\_data {  ZS\_status\_t status; // status  char \*key; // index key value  uint32\_t keylen; // index key length  char \*data; // data  uint64\_t datalen; // data length  uint64\_t seqno; // sequence number for last update  uint64\_t syndrome; // syndrome for key  ZS\_time\_t t\_expiry; // expiry time for the object  char \*primary\_key; // primary key value (if required)  uint32\_t primary\_keylen; // primary key length (if required)  } ZS\_range\_data\_t; |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_UNKNOWN\_CURSOR if the cursor is invalid

ZS\_QUERY\_DONE if the query has completed

ZS\_QUERY\_PAUSED if the query has paused

ZS\_FAILURE otherwise

**ZS\_status\_t ZSGetRangeFinish(struct ZS\_thread\_state \**thd\_state,* struct ZS\_cursor *\*cursor)***

This API ends a range query. It frees up resources allocated for this query in ZS. The application should call this API after enumerating required number of objects.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cursor* | Cursor for this indexed search |

### Returns:

ZS\_SUCCESS if successful   
ZS\_UNKNOWN\_CURSOR if the cursor is invalid

**ZS\_status\_t ZSRangeUpdate(struct ZS\_thread\_state \**thd\_state,* ZS\_cguid\_*t* *cguid*,char *\*range\_key*,uint32\_t *range\_key\_len,* ZS\_range\_update\_cb\_t *callback\_func,* void *\*callback\_args,* ZS\_range\_cmp\_cb\_t *range\_cmp\_callback,* void \**range\_cmp\_cb\_args,* uint32\_t *\*objs\_updated*)**

This API ends a range query. It frees up resources allocated for this query in ZS. The application should call this API after enumerating required number of objects.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *range\_key* | Start of the range to update. |
| in | *range\_key\_len* | Length of key in bytes (including any null termination). |
| in | *callback\_func* | Range update callback function. |
| in | *callback\_args* | Range update callback function arguments. |
| in | *range\_cmp\_cb\_callback* | Range comparator callback function. |
| in | *range\_cmb\_cb\_args* | Range comparator callback function arguments. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_FAILURE\_MEMORY\_ALLOC if out of memory

ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_UNKNOWN\_CURSOR if the cursor is invalid

ZS\_QUERY\_DONE if the query has completed

ZS\_QUERY\_PAUSED if the query has paused

ZS\_OUT\_OF\_STORAGE\_SPACE if the container has run out of space

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSMPut(struct ZS\_thread\_state \**thd\_state,* ZS\_cguid\_t *cguid*, uint32\_t *num\_objs,* struct ZS\_obj\_t *\*objs,* uint32\_t *flags,* uint32\_t *\*objs\_written*)**

Write multiple objects. The semantics of ZSMPut are the same as ZSWriteObject.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container Id |
| in | *num\_objs* | The number of objects to write. |
| in | *flags* | Write option flags (see ZSWriteObject). |
| out | *objs\_written* | The number of objects written. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_FAILURE\_CONTAINER\_NOT\_OPEN if the container is not open

ZS\_OUT\_OF\_STORAGE\_SPACE if the container has run out of space

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

## Snapshot

**ZS\_status\_t ZSCreateContainerSnapshot(struct ZS\_thread\_state \**thd\_state, ZS\_cguid\_t cguid, uint64\_t \*snap\_seq)***

The application can use this API to create a snapshot of a container data. The API returns an monotonically increasing integer number as unique identifier for the snapshot. The application should subsequently enumerate objects using the range APIs and copy the data to different location in order to backup the snapshot . The snapshot is crash-safe. Application can create more than one snapshots at a time.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container id |
| out | *snap\_seq* | Persistent sequence number of this snapshot |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_OUT\_OF\_STORAGE\_SPACE if the container has run out of space

ZS\_TOO\_MANY\_SNAPSHOTS if snapshot limit is reached

ZS\_FAILURE otherwise

**ZS\_status\_t ZSDeleteContainerSnapshot(struct ZS\_thread\_state \**thd\_state, ZS\_cguid\_t cguid, uint64\_t snap\_seq)***

This API deletes a snapshot of a container.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container id |
| out | *snap\_seq* | Persistent sequence number of the snapshot to be deleted |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_SNAPSHOT\_NOT\_FOUND if no snapshot for snap\_seq is found

ZS\_FAILURE otherwise

**ZS\_status\_t ZSGetContainerSnapshots(struct ZS\_thread\_state \**thd\_state, ZS\_cguid\_t cguid, uint32\_t \*n\_snapshots, uint64\_t \*snap\_seqs)***

Application can use this API to get a list of all current snapshots in a container. Array returned in snap\_seqs is allocated by ZS and must be freed by application using ZSFreeBuffer API.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *thd\_state* | Opaque handle for per-thread ZS state. |
| in | *cguid* | Container id |
| out | *n\_snapshots* | Number of snapshots returned |
| out | *snap\_seqs* | Array of snapshot sequence numbers of all current snapshots |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE\_ILLEGAL\_CONTAINER\_ID if the container id is invalid  
ZS\_FAILURE\_CONTAINER\_NOT\_FOUND if the container does not exist

ZS\_SNAPSHOT\_NOT\_FOUND if no snapshot for snap\_seq is found

ZS\_FAILURE\_STORAGE\_WRITE if the write fails

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_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 ZSFlushObject(), ZSFlushContainer(), and ZSFlushCache() calls allow an application to force modified data to be written to storage and synced (Listing 8). ZSFlushObject() does this for a single object, ZSFlushContainer() does this for an entire container, and ZSFlushCache() does this for the entire cache.

If the durability level of a container is set to something other than ZS\_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 \*\*\*\*\*\*\*\*\*\*\*

ZS\_status\_t status;

struct ZS\_thread\_state \*thd\_state;

ZS\_cguid\_t cguid;

char \*objkey;

uint32\_t keylen;

...

status = ***ZSFlushObject(***

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

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

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

***keylen // key length***

***);***

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

...

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

...

}

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

status = ***ZSFlushContainer(***

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

***cguid // container id***

***);***

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

...

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

...

}

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

status = ***ZSFlushCache(***

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

***);***

if (status == ZS\_SUCCESS) {

...

}

#### Listing 8: Controlling the Cache

**ZS\_status\_t ZSFlushObject (struct ZS\_thread\_state \**thd\_state*, ZS\_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 ZS 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:

ZS\_SUCCESS if successful

ZS\_CONTAINER\_UNKNOWN if the container does not exist

ZS\_OBJECT\_UNKNOWN if the object does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSFlushContainer (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*)**

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_CONTAINER\_UNKNOWN if the container does not exist

ZS\_FAILURE otherwise

**ZS\_status\_t ZSFlushCache (struct ZS\_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 ZS state. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

## Statistics

ZS maintains an extensive set of counters for monitoring events within the ZS subsystem. Most events are counted on a per-container basis as well as for all containers within the ZS instance. ZSGetContainerStats() and ZSGetStats() are used to retrieve these statistics (Listing 9). The same statistics structure is used for both calls. The Appendix provides detailed lists of the counters that are returned in ZS\_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 ZS\_histo\_t structure. The histograms are not supported in the ZS 3.0 release.

//\*\*\*\*\*\*\*\*\*\* get ZS stats \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\* Types for ZS stats (from ZS.h) \*/

typedef struct {

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

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

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

ZS\_histo\_t key\_size\_histo;

ZS\_histo\_t data\_size\_histo;

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

} ZS\_stats\_t;

...

ZS\_status\_t status;

struct ZS\_thread\_state \*thd\_state;

ZS\_stats\_t ZS\_stats;

...

status = ***ZSGetStats(***

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

***&ZS\_stats // returns ZS stats***

***);***

if (status != ZS\_SUCCESS) {

...

}

. . .

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

ZS\_stats\_t ctnr\_stats;

ZS\_cguid\_t cguid;

...

status = ***ZSGetContainerStats(***

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

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

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

***);***

if (status != ZS\_SUCCESS) {

...

}

#### Listing 9: Retrieving ZS Statistics

**ZS\_status\_t ZSGetStats (struct ZS\_thread\_state \**thd\_state*, ZS\_stats\_t \**stats*)**

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

### Parameters:

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

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**ZS\_status\_t ZSGetContainerStats (struct ZS\_thread\_state \**thd\_state*, ZS\_cguid\_t *cguid*, ZS\_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 ZS state. |
| in | *cguid* | Container id. |
| in,out | *stats* | Pointer to structure in which to return the statistics. |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

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

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

|  |  |  |
| --- | --- | --- |
| **ZS\_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 ZS Statistics Structure

## Miscellaneous

**ZS\_status\_t ZSGetVersion (*char \*\*str*)**

Returns the ZS version. The version string is returned via a buffer that is allocated by ZS The application should free the buffer using the API ZSFreeBuffer.

### Parameters:

|  |  |  |
| --- | --- | --- |
| out | *str* | Pointer to hold the version string |

### Returns:

ZS\_SUCCESS if successful

ZS\_FAILURE otherwise

**char \* ZSStrError(ZS\_status\_t ZS\_errno)**

Returns error string for a given ZS error code.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *ZS\_errno* | ZS error code |

### Returns:

Error string

**ZS\_status\_t ZSGetLastError(ZS\_cguide\_t cguid, \*\*pp\_err\_context, uint32\_t \*p\_err\_size)**

Returns last ZS error code for a given container cguid.

### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *cguid* | Container id. Must be BTREE type container. |
| in | *pp\_err\_context* | The last BTREE error message. |
| in | *p\_err\_size* | The size of the BTREE error message. |

### Returns:

Last ZS error status for the container.

# Chapter 3: ZS Administration

## Configuration

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

## License

ZS requires a valid license to operate without interruption. Licenses can be obtained from the SanDisk sales team and installed by the user in the server to which the license is bound. ZS expects the license to be in the file /opt/sandisk/ZS/license by default. The license file can be put in a different location using the configuration parameter ZS\_LICENSE\_PATH=<License file path>

Without a valid license, ZS will work for a grace period of 3 days. ZS will terminate if a valid license is not installed within the grace period. The grace period will be 30 days upon expiry of a valid license.

ZS warns periodically by printing warning messages in stdderr if license is close to expire or License is already expired and in grace period.

## Logging

ZS prints logs messages above a specified log level to stderr. The log level is set to “info” by default. The log level can be set using configuration parameter ZS\_LOG\_LEVEL=<log level> in the configuration file or ZSSetProperty(). The log level can also be changed dynamically through CLI.

|  |  |
| --- | --- |
| **Log Level** | **Description** |
| Error | A critical error has occurred. Contact customer support. Example: IO errors, Data inconsistency |
| Warning | Potential problem. ZS will keep running, but user should contact customer support. Example: Low on flash space, Object write errors |
| Info | Informational messages. Example: Configuration used by ZS, Admin port |
| Diagnostic | Diagnosing user errors |
| Debug/trace/trace\_low/devel | Developer level debug messages |

#### Table 6: log levels

## Stats

Applications can retrieve ZS statistics using ZSGetContainerStats() and ZSGetStats(). Applications can also optionally enable periodic dumping of statistics to a specified file. This is disabled by default.

It can be enabled using the configuration parameter ZS\_STATS\_FILE=<filepath>. The dump interval can be configured using ZS\_STATS\_DUMP\_INTERVAL=<interval in secs>. The dump interval can also be dynamically changed through the CLI. Listing 10 shows typical statistics output.

Per Container Statistics

Container Properties:

name = e12e3940

cguid = 419

Size = 0 kbytes

persistence = enabled

eviction = disabled

writethrough = enabled

fifo = disabled

async\_writes = disabled

durability = Periodic sync

num\_objs = 7645

used\_space = 3914240

Application requests:

num\_set\_objs\_with\_expiry = 7645

completed\_enumerations = 1

active\_enumerations = 1

objects\_enumerated = 402

cached\_objects\_enumerated = 402

Overwrite and write-through statistics:

num\_new\_entries = 7645

num\_writethrus\_to\_flash = 7645

Cache to Flash Manager requests:

cache\_misses = 0

cache\_hits = 0

num\_set\_objs\_and\_put = 7645

Flash Manager responses to cache:

num\_set\_objs\_completed = 7645

Flash Manager requests/responses:

num\_set\_objs = 7645

Flash layer return codes:

num\_success = 7645

Overall ZS Statistics

Flash statistics:

num\_items\_flash = 2167157

num\_items\_created = 2188567

num\_overwrites = 21410

num\_put\_ops = 2188567

flash\_space\_allocated = 1409286144

flash\_space\_consumed = 36186112

Flash layout:

flash\_class\_map 27 15 0 0 0 0 0 0 0 0 0 0 0 0 0 0

flash\_slab\_map 1705845 461327 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Application requests:

num\_put\_objs = 1233

num\_set\_objs\_with\_expiry = 72018

num\_get\_objs = 419

num\_del\_objs = 1221

num\_flush\_objs = 2479

num\_sync\_to\_flash = 2479

num\_flush\_container = 408

Overwrite and write-through statistics:

Cache to Flash Manager requests:

cache\_misses = 0

cache\_hits = 0

num\_delete\_objs = 1221

num\_get\_objs\_to\_read = 419

num\_put\_objs = 1233

num\_set\_objs\_and\_put = 72018

Flash Manager responses to cache:

num\_delete\_objs\_completed = 1221

num\_get\_objs\_to\_read\_failed = 419

num\_put\_objs\_completed = 1233

num\_set\_objs\_completed = 72018

Flash Manager requests/responses:

num\_get\_objs = 419

num\_get\_objs\_failed = 419

num\_delete\_objs = 1221

num\_delete\_completed = 1221

num\_set\_objs = 73251

Flash layer return codes:

num\_success = 74472

num\_errors\_objects\_not\_found = 419

Cache statistics:

num\_hash\_buckets\_in\_cache = 100000

num\_cache\_partitions = 10000

num\_objects\_in\_cache = 66252

max\_cache\_capacity = 100000000

current\_data\_size\_in\_cache = 14434903

current\_key\_and\_data\_size\_in\_cache = 19129759

num\_modified\_objs\_in\_cache = 4601

num\_bytes\_of\_modified\_objs\_in\_cache = 889931

num\_mod\_objs\_flushed = 245891

num\_mod\_objs\_flushed\_by\_bgflush = 838438

background\_flush\_progress = 93

num\_background\_flushes = 363

max\_parallel\_flushes = 8

max\_parallel\_bg\_flushes = 8

time\_to\_wait\_after\_bgflush\_for\_nodirty\_data = 1000

max\_percent\_limit\_on\_modifiable\_cache = 100

num\_cache\_ops\_in\_progress = 18

#### Listing 10: Stats dump

## Command Line Interface(CLI)

ZS provides a simple command line interface (CLI) through a TCP port. The ZS CLI uses port 51350 by default. This port can be changed through the configuration parameter ZS\_ADMIN\_PORT=<port number>. The CLI functionality can be disabled by setting configuration property ZS\_ADMIN\_ENABLED=0. The CLI supports the following commands.

|  |  |
| --- | --- |
| **Command** | **Description** |
| container list | Lists all of the container names |
| container stats <container name> [v] | Prints stats of the given container. The option v (verbose) prints extended stats. |
| container stats\_dump <container name|all> [v] | Prints stats of a given container or all containers to the stats file configured through ZS\_STATS\_FILE. The option v(verbose) prints extended stats. |
| container autodump <enable/disable/interval/printcfg> [interval in secs] | This command enables or disables periodic stats dump, and configures the dump interval. |
| log\_level <set/get> [fatal/error/warning/info/diagnostic/debug/trace/trace\_low/devel] | Sets the log level |
| help | Prints help for all supported commands |
| quit | Quits the telnet session |

#### Table 7: CLI commands

Listing 11 shows sample ZS CLI usage.

[root@xen200v03]~# telnet localhost 51350

Trying 127.0.0.1...

Connected to localhost.localdomain (127.0.0.1).

Escape character is '^]'.

help

Supported commands:

container stats <container name> [v]

container stats\_dump <container name|all> [v]

container autodump <enable/disable/interval/printcfg> [interval in secs]

container list

log\_level <set/get> [fatal/error/warning/info/diagnostic/debug/trace/trace\_low/devel]

help

quit

container list

container-4819c940

container stats container-4819c940

Timestamp:Tue May 7 12:06:45 2013

Per Container Statistics

Container Properties:

name = container-4819c940

cguid = 4

Size = 1048576 kbytes

persistence = enabled

eviction = disabled

writethrough = enabled

fifo = disabled

async\_writes = disabled

durability = Periodic sync

num\_objs = 1

used\_space = 512

Application requests:

num\_set\_objs\_with\_expiry = 1

num\_get\_objs\_and\_check\_expiry = 1

completed\_enumerations = 1

objects\_enumerated = 1

cached\_objects\_enumerated = 1

Overwrite and write-through statistics:

num\_new\_entries = 1

num\_writethrus\_to\_flash = 1

Cache to Flash Manager requests:

cache\_misses = 0

cache\_hits = 1

num\_set\_objs\_and\_put = 1

Flash Manager responses to cache:

num\_set\_objs\_completed = 1

Flash Manager requests/responses:

num\_set\_objs = 1

Flash layer return codes:

num\_success = 1

#### Listing 11: CLI usage

## ZS Metadata Validation (ZSCK)

ZS provides a command, zsck, which is used to validate ZS persistent storage structures, much like a file system checker. The zsck command performs validity checks against all ZS storage metadata and can also perform container BTREE level checks if specified. Any errors encountered will be reported in the specified log file. Overall check status will be reported on the zsck display (stderr).

The zsck command has the following syntax:

zsck –-btree –-fixbtree –-logfile=<file> --help

|  |  |
| --- | --- |
| **Option** | **Description** |
| btree | Perform per container BTREE validation. |
| fixbtree | If a BTREE error is encountered, attempt to correct it. Note that is may result in loss of data, but the BTREE will be left in a consistent state. |
| logfile | Output validation check messages to the specified file. The default is “/tmp/zsck.log”. |
| help | Print the command syntax. |

#### Table 8: zsck comman options

Listing 12 shows sample zsck usage.

[root@xen200v03]zsck --btree

mcd\_check\_shard\_descriptor Reading at offset:33554432

mcd\_check\_shard\_descriptor Reading at offset:1107296256

mcd\_check\_shard\_descriptor Reading at offset:2181038080

cmc potbm: n/a (vdc only)

vmc potbm: n/a (vdc only)

vdc potbm: succeeded

mcd\_check\_pot succeeded for cmc.

mcd\_check\_pot succeeded for vmc.

cmc log check: succeeded

vmc log check: succeeded

vdc log check: succeeded

cmc flog\_check: succeeded

vmc flog\_check: succeeded

vdc flog\_check: succeeded

meta check succeeded

Feb 02 11:23:42 2015 43539d20 info fdf\_wrapper.c:3396 msg\_cb INFO: Running Data consistency checker on 3 normal containers.

Feb 02 11:23:42 2015 43539d20 info fdf\_wrapper.c:3396 msg\_cb INFO: Running Data consistency checker on 0 deleted containers.

Feb 02 11:23:42 2015 43539d20 info fdf\_wrapper.c:3396 msg\_cb INFO: Running Data consistency checker on 3 total containers.

Feb 02 11:23:42 2015 ae701940 info fdf\_wrapper.c:3396 msg\_cb INFO: Data consistency check passed for cguid \_\_SanDisk\_pstats\_container.

Feb 02 11:23:43 2015 ae701940 info fdf\_wrapper.c:3396 msg\_cb INFO: Data consistency check passed for cguid container-1.

Feb 02 11:23:43 2015 ae701940 info fdf\_wrapper.c:3396 msg\_cb INFO: Data consistency check passed for cguid \_\_SanDisk\_seqno\_container.

btree check succeeded

#### Listing 12: zsck command output

Listing 13 shows sample zsck log file output.

Feb 02 11:23:34 2015 ZSCHECK\_LABEL 0 ZSCHECK\_SUCCESS flash label valid

Feb 02 11:23:34 2015 ZSCHECK\_SUPERBLOCK 0 ZSCHECK\_SUCCESS superblock magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SUPERBLOCK 0 ZSCHECK\_SUCCESS superblock checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 1099511627777 ZSCHECK\_SUCCESS shard properties magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 1099511627777 ZSCHECK\_SUCCESS shard properties checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 2199023255553 ZSCHECK\_SUCCESS shard properties magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 2199023255553 ZSCHECK\_SUCCESS shard properties checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 3298534883329 ZSCHECK\_SUCCESS shard properties magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_PROPERTIES 3298534883329 ZSCHECK\_SUCCESS shard properties checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS shard descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS shard descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS shard descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS shard descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS shard descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS shard descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS segment list 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS segment list 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS segment list 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS segment list 1 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS segment list 2 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SHARD\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS segment list 3 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 0 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 1 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 1 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 2 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 2 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 3 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 3 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 4 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 4 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 5 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 5 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 6 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 6 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 7 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 7 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 8 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 8 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 9 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 9 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 10 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 10 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 11 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 11 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 12 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 12 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 13 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 13 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 14 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 14 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 15 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS class descriptor 15 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 0 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 1 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 1 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 2 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 2 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 3 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 3 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 4 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 4 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 5 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 5 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 6 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 6 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 7 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 7 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 8 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 8 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 9 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 9 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 10 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 10 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 11 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 11 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 12 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 12 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 13 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 13 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 14 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 14 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 15 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS class descriptor 15 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 0 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 0 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 1 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 1 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 2 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 2 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 3 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 3 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 4 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 4 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 5 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 5 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 6 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 6 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 7 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 7 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 8 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 8 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 9 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 9 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 10 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 10 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 11 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 11 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 12 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 12 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 13 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 13 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 14 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 14 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 15 magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CLASS\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS class descriptor 15 checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS ckpt descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 1099511627777 ZSCHECK\_SUCCESS ckpt descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS ckpt descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 2199023255553 ZSCHECK\_SUCCESS ckpt descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS ckpt descriptor magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_CKPT\_DESCRIPTOR 3298534883329 ZSCHECK\_SUCCESS ckpt descriptor checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_POT\_BITMAP 3298534883329 ZSCHECK\_SUCCESS pot bitmap checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SLAB\_BITMAP 1099511627777 ZSCHECK\_SUCCESS slab bitmap checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SLAB\_BITMAP 2199023255553 ZSCHECK\_SUCCESS slab bitmap checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_SLAB\_BITMAP 3298534883329 ZSCHECK\_SUCCESS slab bitmap checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 10 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 20 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 30 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 40 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 50 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 60 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_INFO Checking object table - 70 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 1099511627777 ZSCHECK\_SUCCESS pot checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 10 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 20 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 30 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 40 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 50 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 60 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_INFO Checking object table - 70 percent complete

Feb 02 11:23:34 2015 ZSCHECK\_OBJECT\_TABLE 2199023255553 ZSCHECK\_SUCCESS pot checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 1099511627777 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 1099511627777 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 1099511627777 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 1099511627777 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 2199023255553 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 2199023255553 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 2199023255553 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 2199023255553 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 3298534883329 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 3298534883329 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 3298534883329 ZSCHECK\_SUCCESS storm log magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_STORM\_LOG 3298534883329 ZSCHECK\_SUCCESS storm log checksum valid

Feb 02 11:23:34 2015 ZSCHECK\_FLOG\_RECORD 1099511627777 ZSCHECK\_SUCCESS flog magic number valid

Feb 02 11:23:34 2015 ZSCHECK\_FLOG\_RECORD 1099511627777 ZSCHECK\_SUCCESS flog checksum valid

Feb 02 11:23:43 2015 ZSCHECK\_BTREE\_NODE 5 ZSCHECK\_SUCCESS Btree check successful for container container-1

Feb 02 11:23:43 2015 ZSCHECK\_CSHARD\_SPACE\_MAP 3298534883329 ZSCHECK\_SUCCESS Space map comparison succeeded for shard 3298534883329

#### Listing 13: zsck log file output

# Chapter 4: ZS Package

The ZS library is delivered as a Software Development Kit (SDK) in a tar file (Example: ZS\_sdk-3.0-1768.249.tar.gz). The SDK contains the following:

* Sample configuration file
* Sample application program
* Header files to be included in the application
* ZS libraries

The listing 14 shows the contents of the ZS SDK package.

./config

ZS\_sample.prop

./include:

common/ZStypes.h

ZS.h

./lib:

libZSdll.a

libZS.so

libZS.so.3.0

./samples:

Makefile

sample\_program.c

#### Listing 14: ZS SDK Contents

The section “Compiling with ZS” describes how to compile user applications with ZS API.

# Chapter 5: Compiling with ZS

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

/usr/lib64/ZS/libZS.so

/use/lib/ZS/libZS.so

/lib64/libZS.so

/lib/libZS.so

/usr/local/lib64/libZS.so

/usr/local/lib/libZS.so

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

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

See the Release Notes for details concerning supported platforms.

# Chapter 6: “Hello World” with ZS

Listing 15 shows a simple example of a complete ZS session:

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

Note that error checking is omitted for clarity!

#include <ZS.h>

main( )

{

ZS\_container\_props\_t props;

struct ZS\_state \*ZS\_state;

struct ZS\_thread\_state \*thd\_state;

struct ZS\_iterator \*iterator;

ZS\_status\_t status;

ZS\_cguid\_t cguid;

char \*key,

\*data;

uint32\_t keylen;

uint64\_t datalen;

ZSLoadProperties( "conf/ZS\_sample.prop");

ZSInit( &ZS\_state);

ZSInitPerThreadState( ZS\_state, &thd\_state);

ZSLoadCntrPropDefaults( &props);

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

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

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

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

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

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

ZSFreeBuffer( data);

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

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

ZS\_SUCCESS) {

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

datalen);

ZSFreeBuffer( key);

ZSFreeBuffer( data);

}

ZSFinishEnumeration( thd\_state, iterator);

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

ZSCloseContainer( thd\_state, cguid);

ZSDeleteContainer( thd\_state, cguid);

ZSReleasePerThreadState( &thd\_state);

ZSShutdown( ZS\_state);

return (0);

}

#### Listing 15: “Hello World” with ZS

In order to run the program, you must first create a file link to your flash device or flash raid device.

For example:

* Assume the flash device is /dev/sdc.
* Assume the ZS\_FLASH\_FILENAME property is set to “/opt/sandisk/flash0”.
* Create a softlink:

ln –s /dev/sdc /opt/sandisk/flash0

* Set the permissions on the flash device and file path such that the program has access to the device.

The program can now be executed.

Note that for testing purposes, you may also set ZS\_FLASH\_FILENAME to a normal disk file path. This would of course be much slower than accessing a flash device.

# Appendix: ZS Statistics

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

* 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

|  |  |
| --- | --- |
| **ZS 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) |
| ENUM\_TOTAL | Total number enumerations completed |
| ENUM\_ACTIVE | Number of active enumerations |
| ENUM\_OBJECTS | Total number of objects enumerated |
| ENUM\_CACHED\_OBJECTS | Number of enumerated objects read from cache |
| NUM\_CONT\_DELETES\_PEND | Number of asynchronous container deletes pending |
| NUM\_CONT\_DELETES\_PROG | Number of asynchronous container deletes under progress |

#### Table 5: ZS Statistics: ZS Requests

|  |  |
| --- | --- |
| **Flash Statistics** | |
| **Name** | **Description** |
| ZS\_FLASH\_STATS\_NUM\_OBJS | Number of objects in flash |
| ZS\_FLASH\_STATS\_NUM\_CREATED\_OBJS | Total number of created objects |
| ZS\_FLASH\_STATS\_NUM\_EVICTIONS | Number of flash evictions |
| ZS\_FLASH\_STATS\_NUM\_HASH\_EVICTIONS | Number of hash evictions |
| ZS\_FLASH\_STATS\_NUM\_INVAL\_EVICTIONS | Number of invalid evictions |
| ZS\_FLASH\_STATS\_NUM\_SOFT\_OVERFLOWS | Number of soft overflows in hast table |
| ZS\_FLASH\_STATS\_NUM\_HARD\_OVERFLOWS | Number of hard overflows in hash table |
| ZS\_FLASH\_STATS\_GET\_HASH\_COLLISION | Number of hash collisions for get operations |
| ZS\_FLASH\_STATS\_SET\_HASH\_COLLISION | Number of hash collisions for set operations |
| ZS\_FLASH\_STATS\_NUM\_OVERWRITES | Number of overwrites |
| ZS\_FLASH\_STATS\_NUM\_OPS | Number of all flash operations |
| ZS\_FLASH\_STATS\_NUM\_READ\_OPS | Number of read operations |
| ZS\_FLASH\_STATS\_NUM\_GET\_OPS | Number of get operations |
| ZS\_FLASH\_STATS\_NUM\_PUT\_OPS | Number of put operations |
| ZS\_FLASH\_STATS\_NUM\_DEL\_OPS | Number of delete operations |
| ZS\_FLASH\_STATS\_GET\_EXIST\_CHECKS | Number of existence checks for gets |
| ZS\_FLASH\_STATS\_NUM\_FULL\_BUCKETS | Number of full hash buckets |
| ZS\_FLASH\_STATS\_PENDING\_IOS | Number of pending IO’s |
| ZS\_FLASH\_STATS\_SPACE\_ALLOCATED | Flash space allocated in bytes |
| ZS\_FLASH\_STATS\_SPACE\_CONSUMED | Flash space consumed in bytes |

#### Table 6: ZS Flash Statistics

|  |  |
| --- | --- |
| **ZS Overwrite/Write-Through Statistics** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_OVERWRITES\_S | number of overwrites of objects in the S state |
| ZS\_CACHE\_STAT\_OVERWRITES\_M | number of overwrites of objects in the M state |
| ZS\_CACHE\_STAT\_INPLACEOWR\_S | number of in-place overwrites of objects in the S state |
| ZS\_CACHE\_STAT\_INPLACEOWR\_M | number of in-place overwrites of objects in the M state |
| ZS\_CACHE\_STAT\_NEW\_ENTRIES | number of newly created cache objects (as opposed to overwrites) |
| ZS\_CACHE\_STAT\_WRITETHRUS | number of write-throughs to flash |
| ZS\_CACHE\_STAT\_WRITEBACKS | number of write-backs (of dirty objects) to flash |
| ZS\_CACHE\_STAT\_FLUSHES | number of flush operations to flash (includes background flushes) |
| ZS\_CACHE\_STAT\_ASYNC\_DRAINS | Number of asynchronous drain operations |
| ZS\_CACHE\_STAT\_ASYNC\_PUTS | Number of asynchronous put operations from cache to flash |
| ZS\_CACHE\_STAT\_ASYNC\_PUT\_FAILS | Number of asynchronous put operations from cache to flash failed |
| ZS\_CACHE\_STAT\_ASYNC\_FLUSHES | Number of asynchronous flush operations from cache to flash |
| ZS\_CACHE\_STAT\_ASYNC\_FLUSH\_FAILS | Number of asynchronous flush operations from cache to flash failed |
| ZS\_CACHE\_STAT\_ASYNC\_WRBKS | Number of writeback operations |
| ZS\_CACHE\_STAT\_ASYNC\_WRBK\_FAILS | Number of writeback operations failed |
| ZS\_CACHE\_STAT\_CACHE\_MISSES | Number of times objects not found in cache |
| ZS\_CACHE\_STAT\_CACHE\_HITS | Number of times objects found in cache |

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

|  |  |
| --- | --- |
| **Action Entity (Cache) To Home Entity (Flash Manager)**  **Requests (AH---)** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_AHCOB | create object |
| ZS\_CACHE\_STAT\_AHCOP | create object and put data |
| ZS\_CACHE\_STAT\_AHCWD | castout with data |
| ZS\_CACHE\_STAT\_AHDOB | delete object |
| ZS\_CACHE\_STAT\_AHFLD | flush object |
| ZS\_CACHE\_STAT\_AHGTR | get object with intent to read |
| ZS\_CACHE\_STAT\_AHGTW | get object with intent to write |
| ZS\_CACHE\_STAT\_AHPTA | put object |
| ZS\_CACHE\_STAT\_AHSOB | set object |
| ZS\_CACHE\_STAT\_AHSOP | set object put |

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

|  |  |
| --- | --- |
| **Home Entity (Flash Manager) To Action Entity (Cache)**  **Responses (HA---)** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_HACRC | create object completed |
| ZS\_CACHE\_STAT\_HACRF | create object failed |
| ZS\_CACHE\_STAT\_HACSC | castout completed |
| ZS\_CACHE\_STAT\_HACSF | castout failed |
| ZS\_CACHE\_STAT\_HADEC | delete object completed |
| ZS\_CACHE\_STAT\_HADEF | delete object failed |
| ZS\_CACHE\_STAT\_HAFLC | flush object completed |
| ZS\_CACHE\_STAT\_HAFLF | flush object failed |
| ZS\_CACHE\_STAT\_HAGRC | get object to read completed |
| ZS\_CACHE\_STAT\_HAGRF | get object to read failed |
| ZS\_CACHE\_STAT\_HAGWC | get object to write completed |
| ZS\_CACHE\_STAT\_HAGWF | get object to write failed |
| ZS\_CACHE\_STAT\_HAPAC | put object completed |
| ZS\_CACHE\_STAT\_HAPAF | put object failed |
| ZS\_CACHE\_STAT\_HASTC | set object completed |
| ZS\_CACHE\_STAT\_HASTF | set object failed |

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

|  |  |
| --- | --- |
| **Flash Requests and Responses to/from Flash Manager**  **(HF--- and FH---)** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_HFXST | check existence of object |
| ZS\_CACHE\_STAT\_FHXST | existence check succeeded |
| ZS\_CACHE\_STAT\_FHNXS | existence check failed |
| ZS\_CACHE\_STAT\_HFGFF | get object |
| ZS\_CACHE\_STAT\_FHDAT | object data |
| ZS\_CACHE\_STAT\_FHGTF | get object failed |
| ZS\_CACHE\_STAT\_HFPTF | put object |
| ZS\_CACHE\_STAT\_FHPTC | put object completed |
| ZS\_CACHE\_STAT\_FHPTF | put object failed |
| ZS\_CACHE\_STAT\_HZSF | delete object |
| ZS\_CACHE\_STAT\_FHDEC | delete object completed |
| ZS\_CACHE\_STAT\_FHDEF | delete object failed |
| ZS\_CACHE\_STAT\_HFCIF | create object |
| ZS\_CACHE\_STAT\_FHCRC | create object completed |
| ZS\_CACHE\_STAT\_FHCRF | create object failed |
| ZS\_CACHE\_STAT\_HFCZF | create zeroed object |
| ZS\_CACHE\_STAT\_FHCRC | create zeroed object completed |
| ZS\_CACHE\_STAT\_FHCRF | create zeroed object failed |
| ZS\_CACHE\_STAT\_HFSET | set object |
| ZS\_CACHE\_STAT\_FHSTC | set object completed |
| ZS\_CACHE\_STAT\_FHSTF | set object failed |
| ZS\_CACHE\_STAT\_HFCSH | create shard |
| ZS\_CACHE\_STAT\_FHCSC | create shard completed |
| ZS\_CACHE\_STAT\_FHCSF | create shard failed |
| ZS\_CACHE\_STAT\_HFSSH | sync shard |
| ZS\_CACHE\_STAT\_FHSSC | sync shard completed |
| ZS\_CACHE\_STAT\_FHSSF | sync shard failed |
| ZS\_CACHE\_STAT\_HFDSH | delete shard |
| ZS\_CACHE\_STAT\_FHDSC | delete shard completed |
| ZS\_CACHE\_STAT\_FHDSF | delete shard failed |
| ZS\_CACHE\_STAT\_HFGLS | get last sequence number |
| ZS\_CACHE\_STAT\_FHGLC | get last sequence number completed |
| ZS\_CACHE\_STAT\_FHGLF | get last sequence number failed |
| ZS\_CACHE\_STAT\_HFGIC | get iteration cursors |
| ZS\_CACHE\_STAT\_FHGIC | get iteration cursors completed |
| ZS\_CACHE\_STAT\_FHGIF | get iteration cursors failed |
| ZS\_CACHE\_STAT\_HFGBC | get by cursor |
| ZS\_CACHE\_STAT\_FHGCC | get by cursor completed |
| ZS\_CACHE\_STAT\_FHGCF | get by cursor failed |
| ZS\_CACHE\_STAT\_HFGSN | get sequence number |
| ZS\_CACHE\_STAT\_HFGCS | get container stat |
| ZS\_CACHE\_STAT\_FHGSC | get container stat completed |
| ZS\_CACHE\_STAT\_FHGSF | get container stat failed |
| ZS\_CACHE\_STAT\_HFSRR | start replicating |
| ZS\_CACHE\_STAT\_FHSRC | start replicating completed |
| ZS\_CACHE\_STAT\_FHSRF | start replicating failed |
| ZS\_CACHE\_STAT\_HFSPR | stop replicating |
| ZS\_CACHE\_STAT\_FHSPC | stop replicating completed |
| ZS\_CACHE\_STAT\_FHSPF | stop replicating failed |
| ZS\_CACHE\_STAT\_HFFLA | flush object |
| ZS\_CACHE\_STAT\_FHFLC | flush object completed |
| ZS\_CACHE\_STAT\_FHFLF | flush object failed |
| ZS\_CACHE\_STAT\_HFRVG | release vip group |
| ZS\_CACHE\_STAT\_FHRVC | release vip group completed |
| ZS\_CACHE\_STAT\_FHRVF | release vip group failed |
| ZS\_CACHE\_STAT\_HFNOP | noop |
| ZS\_CACHE\_STAT\_FHNPC | noop completed |
| ZS\_CACHE\_STAT\_FHNPF | noop failed |
| ZS\_CACHE\_STAT\_HFOSH | open shard |
| ZS\_CACHE\_STAT\_FHOSC | open shard completed |
| ZS\_CACHE\_STAT\_FHOSF | open shard failed |
| ZS\_CACHE\_STAT\_HFFLS | flush object |
| ZS\_CACHE\_STAT\_FHFCC | flush object completed |
| ZS\_CACHE\_STAT\_FHFCF | flush object failed |
| ZS\_CACHE\_STAT\_HFFIV | flush invalidate object |
| ZS\_CACHE\_STAT\_FHFIC | flush invalidate object completed |
| ZS\_CACHE\_STAT\_FHFIF | flush invalidate object failed |
| ZS\_CACHE\_STAT\_HFINV | invalidate object |
| ZS\_CACHE\_STAT\_FHINC | invalidate object completed |
| ZS\_CACHE\_STAT\_FHINF | invalidate object failed |
| ZS\_CACHE\_STAT\_HFFLC | flush container |
| ZS\_CACHE\_STAT\_FHLCC | flush container completed |
| ZS\_CACHE\_STAT\_FHLCF | flush container failed |
| ZS\_CACHE\_STAT\_HFFLI | flush invalidate container |
| ZS\_CACHE\_STAT\_FHLIC | flush invalidate container completed |
| ZS\_CACHE\_STAT\_FHLIF | flush invalidate container failed |
| ZS\_CACHE\_STAT\_HFINC | invalidate container |
| ZS\_CACHE\_STAT\_FHCIC | invalidate container completed |
| ZS\_CACHE\_STAT\_FHCIF | invalidate container failed |

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

|  |  |
| --- | --- |
| **Counts of Low-level Flash Return Codes** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_EOK | success |
| ZS\_CACHE\_STAT\_EPERM | not permitted |
| ZS\_CACHE\_STAT\_ENOENT | not found |
| ZS\_CACHE\_STAT\_EDATASIZE | user-supplied data buffer is too small |
| ZS\_CACHE\_STAT\_ESTOPPED | container is stopped |
| ZS\_CACHE\_STAT\_EBADCTNR | container does not exist |
| ZS\_CACHE\_STAT\_EDELFAIL | deletion for a local failure failed |
| ZS\_CACHE\_STAT\_EAGAIN | try again (transient error) |
| ZS\_CACHE\_STAT\_ENOMEM | out of memory |
| ZS\_CACHE\_STAT\_EACCES | permission denied |
| ZS\_CACHE\_STAT\_EINCONS | inconsistency during replication |
| ZS\_CACHE\_STAT\_EBUSY | device busy |
| ZS\_CACHE\_STAT\_EEXIST | object exists |
| ZS\_CACHE\_STAT\_EINVAL | invalid argument |
| ZS\_CACHE\_STAT\_EMFILE | too many objects |
| ZS\_CACHE\_STAT\_ENOSPC | out of flash space |
| ZS\_CACHE\_STAT\_ENOBUFS | out of system resource |
| ZS\_CACHE\_STAT\_ESTALE | stale data |
| ZS\_CACHE\_STAT\_EDQUOT | quota exceeded |
| ZS\_CACHE\_STAT\_RMT\_EDELFAIL | deletion for a remote failure failed |
| ZS\_CACHE\_STAT\_RMT\_EBADCTNR | container does not exist on a remote node |

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

|  |  |
| --- | --- |
| **Miscellaneous Per-Cache Statistics** | |
| **Name** | **Description** |
| ZS\_CACHE\_STAT\_HASH\_BUCKETS | number of hash buckets in cache directory |
| ZS\_CACHE\_STAT\_NUM\_SLABS | number of cache partitions |
| ZS\_CACHE\_STAT\_NUM\_ELEMENTS | number of objects in the cache |
| ZS\_CACHE\_STAT\_MAX\_SIZE | maximum capacity of cache, in bytes |
| ZS\_CACHE\_STAT\_CURR\_SIZE | number of bytes that contain data in the cache (does not include keys) |
| ZS\_CACHE\_STAT\_CURR\_SIZE\_WKEYS | number of bytes containing data and keys |
| ZS\_CACHE\_STAT\_NUM\_MODIFIED\_OBJS | number of modified objects in the cache (flash does NOT have the latest data) |
| ZS\_CACHE\_STAT\_NUM\_MODIFIED\_OBJS\_WKEYS | number of bytes of modified objects in the cache (including keys) |
| ZS\_CACHE\_STAT\_NUM\_MODIFIED\_OBJS\_FLUSHED | number of modified objects that have been flushed to flash via cache or container flushes (not including background flushes) |
| ZS\_CACHE\_STAT\_NUM\_MODIFIED\_OBJS\_BGFLUSHED | number of modified objects that have been flushed to flash by the background flush process |
| ZS\_CACHE\_STAT\_NUM\_PENDING\_REQS | number of pending remote cache requests (due to replication) |
| ZS\_CACHE\_STAT\_NUM\_MODIFIED\_OBJC\_REC | number of modified cache objects copied during the recovery process |
| ZS\_CACHE\_STAT\_BGFLUSH\_PROGRESS | progress of the current background flush cycle (percent) |
| ZS\_CACHE\_STAT\_NUM\_BGFLUSH | number of times the background flusher has cycled through the cache |
| ZS\_CACHE\_STAT\_NUM\_FLUSH\_PARALLEL | total number of flushes that can occur in parallel (includes explicit and background flushes) |
| ZS\_CACHE\_STAT\_NUM\_BGFLUSH\_PARALLEL | total number of background flushes that can occur in parallel (must be <= nFlshTok) |
| ZS\_CACHE\_STAT\_BGFLUSH\_WAIT | time to wait after a background flush cycle in which no dirty data is found |
| ZS\_CACHE\_STAT\_MODIFIED\_PCT | percentage limit on the number of bytes in the cache that are modified |
| ZS\_CACHE\_STAT\_NUM\_APP\_BUFFERS | number of application buffers that are in use |
| ZS\_CACHE\_STAT\_NUM\_CACHE\_OPS\_PROG | number of cache operations that are in progress |
| ZS\_CACHE\_STAT\_NUM\_FDBUFFER\_PROCESSED | number of flash data buffers currenly being processed |
| ZS\_CACHE\_STAT\_NUM\_RESP\_PROCESSED | number of response messages currently being processed |

#### Table 12: ZS Statistics: Miscellaneous

# Appendix: ZS Storm Mode

In order to support very large storage subsystems in the petabyte range, ZS supports the “Storm” mode which performs the following:

* Limits ZS\_BLOCK\_SIZE to 8192 bytes.
  + Any attempt to change the block size will disable Storm mode.
* Efficient storage mapping metadata.
* Higher levels of durability (hardware crash safe).
* Enhanced statistics.
* ZS persistent metadata validation (zsck).

By default, ZS 3.0 will be brought up in Storm mode. To disable Storm mode, set the ZS\_STORM\_MODE property to 0. Also, changing ZS\_BLOCK\_SIZE to any value other than 8192 will disable Storm mode.