NAME

helium – a utility to test and benchmark the performance of helium

SYNOPSIS

helium −−perf [**−−trace**] [options] *URL...*

helium −−server [**−−trace**] *URL...*

helium −−test [**−−trace**] *URL...*

helium [**−−version** | **−−help**]

DESCRIPTION

helium is a utility that allows users to measure the performance of the Helium data store and run sanity Helium's internal sanity checks.

When running **helium**, you can specify one or more *URLs* as targets for the tests. The argument must be a comma-separated list of accessible block devices or files. Be sure you have sufficient permissions to both read and write to the device.

WARNING: All block devices or files specified by this the *URL* will be overwritten. Be sure to back up your data before running this program.

Helium can run in one of three modes:

−−perf Run all built-in performance tests. Used to measure the performance of Helium on a given system. This is the only mode that accepts the additional options listed below.

−−server Run a built-in server. Used to host a server that is able to connect to Helium client(s).

−−test Run all built-in self tests. Used to perform sanity checks and make sure Helium is working properly.

OPTIONS

−o *OPERATIONS*
Set the number of operations per thread.

−T *THREADS*
Require Helium to run exactly the specified number of threads.

−t *MAX_THREADS*
Run performance starting from 1 thread to "*MAX_THREADS* threads.

−v *SIZE*
Set the size of the objects to be used during the specified transactions.

−−help Print this help information and quit.

−−nowarn
Do not issue warning prompts. Since helium overwrites data on the "*URL*, use this with caution.

−−trace
Print trace messages to stderr. This is primarly used for debugging.

−−version
Print version information and quit.

ADVANCED OPTIONS

−−config *FILE*
Read Helium environment settings from a configuration file. See **he_open**(3) for technical details on how this file is processed.

−−delete *THREADS*
Run deletes only. Assumes a previous run of −−updates was done.

−−randread *THREADS*
Run random reads only. The "*THREADS*" is the number of threads used when running a previous −−update.

−−seqread *THREADS*
Run sequential reads only. The "*THREADS*" is the number of threads used when running a previous −−update.

−−trim Send a trim command to the *URL* before starting a test. If the block device referred to by the *URL* does not support trim, this option is
ignored.

−−update
Run Helium updates and exit.

EXAMPLES

helium --perf -t 16 -v 100 -o 1000000 --trim he://.//dev/ssd

* Run on 1 to 16 threads.

* Perform 1 million operations per thread.

* For each thread, run insert, then read, rand_read, and lastly delete.

* Use an object size of 100 bytes.

* Pass trim instructions to the solid state drive.

* Run it on block device "/dev/ssd".

* After the run, the performance in millions of operations per second is printed to stdout.

helium --perf -T 4 he://.//_3000000000

* Run on exactly 4 threads.

* Run a memory only test using a 3GB ramdisk. (The '_' in front of a device name specifies that a memory device is to be used)

* After the run, the performance in millions of operations per second is printed to stdout.

SEE ALSO

he(3), **he_open**(3), **lsblk**(8)

NOTICE

NAME

The Helium Data Store Application Programming Interface

INTRODUCTION

Helium Data Store is a lightweight embedded key/value datastore specifically designed to take advantage of Solid State Devices (SSDs) and next generation many-core processing platforms. In a nutshell, Helium is designed to (a) provide a rich, simple, intuitive, and consistent interface to application programmers; and (b) provide an ultra-high performance, robust, networked, and portable storage engine to simplify application design and maintenance.

HELIUM TERMS

Helium Library

The Helium embedded library that provides a set of functions and associated types, structures, and constants.

Block Device

A storage device (e.g., SSD, loop device, partition, fixed-size file) that is accessible to Helium via low-level operating system open/read/write/close interfaces. I/O to a block device or fixed-size file is in some chunk size (typically multiples of 512 byte sectors).

Helium Volume

One or more block devices or fixed-size files bundled into a single volume (managed by Helium), providing a single large storage fabric to be used by datastores.

Helium URL

A uniform way to specify a volume that may reside on a local machine or across the network.

Helium Key/Value Item (abbreviated as item) An item is a key, a value, and the associated lengths of the key and value data blobs. Helium assumes keys and values are binary blobs of data.

Helium Datastore

A datastore is a container of key/value pairs that are isolated from all other datastores residing on a shared volume. Helium datastore operations create, delete, rename, and query status of datastores.

Helium Transaction

An isolated sequence of operations that may be discarded or atomically merged with the datastore.

Datastore Update (abbreviated as update)

Atomic operations that change the state of a datastore (or transaction), e.g., by adding an item or deleting an item.

Datastore Lookup (abbreviated as lookup)

Atomic operation that retrieve data from a datastore (or transaction) without changing the state of the datastore.

VALIDITY API

he_is_valid(3), **he_is_transaction**(3)

DATASTORE API

he_enumerate(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3)

TRANSACTION API

he_transaction(3), **he_commit**(3), **he_discard**(3)

UPDATE API

he_update(3), **he_insert**(3), **he_replace**(3), **he_delete**(3) **he_delete_lookup**(3)

LOOKUP API

he_exists(3), **he_lookup**(3), **he_next**(3), **he_prev**(3) **he_iterate**(3)

SUPPORT API

he_version(3), **he_perror**(3), **he_strerror**(3)

NOTICE

NAME

he_is_valid - check datastore handle for validity

SYNOPSIS

#include <he.h>

int he_is_valid(he_t *he***);**
int he_is_transaction(he_t *he***);**

DESCRIPTION

These functions return information about the validity of a Helium datastore.

he_is_valid() checks if the handle pointed to by *he* is a valid and open Helium datastore.

he_is_transaction() checks if the handle pointed to by *he* is a valid transaction handle.

RETURN VALUE

On success, true (non-zero integer) is returned. On error, false (zero) is returned and *errno* is set appropriately.

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_is_transaction - check transaction handle for validity

SYNOPSIS

#include <he.h>

int he_is_valid(he_t *he***);**
int he_is_transaction(he_t *he***);**

DESCRIPTION

These functions return information about the validity of a Helium datastore.

he_is_valid() checks if the handle pointed to by *he* is a valid and open Helium datastore.

he_is_transaction() checks if the handle pointed to by *he* is a valid transaction handle.

RETURN VALUE

On success, true (non-zero integer) is returned. On error, false (zero) is returned and *errno* is set appropriately.

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_enumerate - enumerate datastores residing on volume

SYNOPSIS

#include <he.h>

int he_enumerate(const char *_url_, he_cbf_t _cbf_ , void *_arg_);

DESCRIPTION

The **he_enumerate**() function is used to obtain a list of datastores that reside on a volume. This function takes as its first argument *url*, specifying the volume (see **he_open**(3) for details on valid URL formats). The second argument is a pointer to a function, namely a callback function residing within the user code. The third argument *arg*, is a generic pointer that is passed directly to the callback function and otherwise unused by Helium internals. The callback function is called once for each datastore residing on the volume.

The callback function *cbf* must have the following signature.

int some_function_name(void *_arg_, const char *_name_);

The callback function *cbf* must return 0 as long as additional datastore names are to be received. Any value, other than 0, will cause the call to **he_enumerate**() to terminate, returning **HE_ERR_TERMINATED**.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int cbf(void *arg, const char *name) {
        *((int *)arg) += 1;
        printf("%s\n", name);
        return 0;
}

int main() {
        int count;

        count = 0;
        if (he_enumerate("he://.//file", cbf, &count)) {
                he_perror("he_enumerate");
                return -1;
        }
        printf("%i datastores found\n", count);
        return 0;
}
```

RETURN VALUE

On success **he_enumerate**() returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**.

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_open - create or open datastore

SYNOPSIS

#include <he.h>

he_t he_open(const char ****url*,**
const char ****name*,**
int *flags***,**
const struct he_env ****env*);**
int he_close(he_t *he***);**

DESCRIPTION

The **he_open**() and **he_close**() functions are used to obtain a handle to a datastore or close the handle when done operating on the datastore.

The **he_open**() function takes as its first argument *url*, a URL specifying a volume. The second argument *name*, names the datastore. A datastore name must not be NULL or an empty string. The third argument, *flags*, determines opening modes, as explained below. On success, **he_open**() returns a handle of type **he_t**. On error, NULL is returned and *errno* is set accordingly. The **he_t** handle is defined as a void pointer and may be copied or shared among multiple threads as needed. Alternatively, multiple handles to the same datastore may be obtained by calling **he_open**(). Each handle obtained by calling **he_open**() must be closed, when no longer needed, by calling **he_close**().

A volume must be properly initialized (i.e., formatted) before the first use. Once formatted, any number of datastores may be created on the volume. Formatting a volume is accomplished using the **he_open**() function. Prior to formatting a volume, all data residing on the underlying block devices (or fixed-size files) must be backed up. Once formatted, non-Helium content on the volume may no longer be retrievable. A volume, once created, must only be accessed through the Helium API. Volumes remain static from the moment of creation and on. In other words, it is not possible to dynamically expand or shrink a volume once it has been formatted. The minimum size of a volume must be at least a Gigabyte (2^30).

The *flags* argument controls the opening mode and may be a bitwise OR of the following constants.

HE_O_CREATE – If the datastore does not exist, create it. Otherwise, this bit has no effect. If this bit is not set, **he_open**() will fail when attempting to open a datastore that does not exist.

HE_O_TRUNCATE – If the datastore exists, truncate it on open. Truncating a datastore removes all items stored in it. This flag is only applicable if no other handle to the datastore is open, otherwise, the call fails by returning NULL and setting *errno* to **HE_ERR_DATASTORE_IN_USE**.

HE_O_VOLUME_CREATE – If *url* specifies an invalid volume, a valid volume is created. Otherwise, this flag has no effect.

HE_O_VOLUME_TRUNCATE – If the volume is a valid volume, and no other handles are currently open, the volume is truncated, removing all datastores (and associated items) from the volume. Warning, this flag must be used with extreme care.

HE_O_NOSORT – Specify this flag to keep the datastore in un-sorted order, hence reducing memory overhead at the expense of slower delete operations. Furthermore, if set, **he_next**(3) or **he_prev**(3) functionality will no longer be supported. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

HE_O_SCAN – On startup, force a scan and rebuild of the indexer. Depending on the datastore

size, this may take a while, as the entire dataset is read and a new indexer is constructed. Helium automatically scans and reconstructs the indexer if needed, hence this flag is not essential for correct operations. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

HE_O_CLEAN – On startup, force garbage collection of all datastores residing on the volume. Depending on the datastore size, this may take a while, as the entire dataset is processed. Helium automatically performs garbage collection if needed, hence this flag is not essential for correct operations. This flag is ignored if a handle to some datastore residing on the volume is currently open.

HE_O_COMPRESS – Specify this flag to compress the datastore's item values before writing to the volume. As common sense would dictate, compression *may* yield storage savings at the expense of performance. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

URL SPECIFICATION

Helium datastores reside on a properly formatted volume. A volume is a set of one or more block devices (or fixed-sized files). Helium uses a URL scheme for specifying the path to a volume, regardless of it residing on a local machine or on a remote Helium server. In Unix-like environments, block devices (or fixed-sized files) are listed under the /dev directory (e.g., /dev/sda). Note that a partition on a raw device is considered a raw device as well (e.g., /dev/sda1). A Helium URL is a null-terminated string taking one of the following forms.

he://./path-to-device1,path-to-device2,...

he://www.server.com/path-to-device1,path-to-device2,...

he://www.server.com:123/path-to-device1,path-to-device2,...

All URLs must begin with the **he://** scheme. Then, the URL must specify the machine location of the block devices (or fixed-sized files). Finally, the URL must list the pathnames of the devices. In the first form, the dot is used to specify that the volume is on the local machine and intended to be used by a single process, hence, Helium will directly read and write the volume, in exclusive mode, using low-level operating system API. Some examples of the first form are shown below.

he://.//dev/loop0

he://.//dev/sda,/dev/sdb

When more than one block device (or fixed-size files) are used to form a volume, the ordering of the devices in the URL is insignificant. For instance, the following two URLs point to the same volume.

he://.//dev/sda,/dev/sdb

he://.//dev/sdb,/dev/sda

In the second form, a volume may be served by a single server to a number of clients (on localhost or remote host). The second form specifies the address of a Helium server, but using an IP or a hostname. A Helium server may reside on a remote node. All Helium functions (e.g., **he_open**()) seamlessly operate within a node or across node boundaries while providing precise semantics and consistent behavior, as described in this document. The word **localhost** may be used if both the server and client reside on the same physical node. By default, Helium uses port number 41000. A colon (i.e., ':') may be used, following the host address, to explicitly specify a port number, as shown below.

he://www.some-server.com:12345//dev/sda

/dev/sda,/dev/sdb

/dev/sdb,/dev/sda

OPERATING ENVIRONMENT

The *env* argument of **he_open**(), if not NULL, is used to specify environment values. The *env* argument is only applicable if no other handle to the datastore is open, otherwise, it is ignored. The *env* is a pointer to the following structure.

```
struct he_env {
        uint64_t fanout;
        uint64_t write_cache;
        uint64_t read_cache;
        uint64_t auto_commit_period;
        uint64_t auto_clean_period;
        uint64_t clean_util_pct;
        uint64_t clean_dirty_pct;
        uint64_t retry_count;
        uint64_t retry_delay;
};
```

The *fanout* member determines the level of internal parallelism that Helium will accommodate. This number determines the number of operations that may simultaneously operate on the datastore, provided enough processor resources are available. This setting impacts performance, not functionality. This member is needed when a new datastore is created, otherwise, it is ignored. Once created, a datastore's fanout remains fixed. The default fanout is 33.

The *write_cache* member determines the amount of memory, per datastore, in bytes, dedicated to write buffering. The default write buffering is 256 MiB and is adequate for majority of use cases. Increase this value only if your key/value items are very large (4 KiB or larger).

The *read_cache* member determines the amount of memory, per datastore, in bytes, dedicated to read buffering. The default read buffering is 1 GiB. For ultimate performance, increase this value to be 10% above the application working set.

The *auto_commit_period* member specifies the duration of time, in seconds, between automatic commits of the datastore. This value limits the amount of data loss in the event of an unexpected shutdown or system crash. You may set this value to 2^64 - 1 to effectively disable auto commit. The default auto commit period is 10 seconds.

The *auto_clean_period* member specifies the duration of time, in seconds, between automatic check for garbage collection. Every this often, the status of datastore is checked (see the next two members) and garbage collection is triggered if needed. The default auto commit period is 2 seconds.

The *clean_util_pct* members specifies the percentage of volume utilization (100 * utilized / capacity) that triggers garbage collection. Both utilization percentage and dirty percentage must be reached before garbage collection is activated in the background. The default utilization percentage is set to 60%.

The *clean_dirty_pct* member specifies the percentage of garbage (100 * deleted_items / (valid_items + deleted_items)) that triggers garbage collection. Both utilization percentage and dirty percentage must be reached before garbage collection is activated in the background. The default dirty percentage is set to 20%.

The *retry_count* member specifies the number of times an operation that failed due to volume being full will be retried. Subsequent attempts may succeed if the background garbage collector is given some time to free up space. The default retry count is set to 500.

The *retry_delay* member, working in conjunction with the *rety_count*, specifies the delay, in micro-seconds, between retry operations. The default retry delay is set to 20000 micro-seconds.

Set any member of **he_env** structure to zero to force the default value. Out of range values will be adjusted to the minimum or maximum settings as needed.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int main() {
        he_t he;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        if (!he) {
                he_perror("he_open");
                return -1;
        }

        /* datastore operations */

        if (he_close(he)) {
                he_perror("he_close");
                return -1;
        }
        return 0;
}
```

RETURN VALUE

On success **he_open**() returns a valid datastore handle. On error, NULL is returned and *errno* is set to an appropriate Helium error code. On success **he_close**() returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3),

he_insert(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_close - close datastore

SYNOPSIS

#include <he.h>

he_t he_open(const char ****url*,**
const char ****name*,**
int *flags***,**
const struct he_env ****env*);**
int he_close(he_t *he***);**

DESCRIPTION

The **he_open**() and **he_close**() functions are used to obtain a handle to a datastore or close the handle when done operating on the datastore.

The **he_open**() function takes as its first argument *url*, a URL specifying a volume. The second argument *name*, names the datastore. A datastore name must not be NULL or an empty string. The third argument, *flags*, determines opening modes, as explained below. On success, **he_open**() returns a handle of type **he_t**. On error, NULL is returned and *errno* is set accordingly. The **he_t** handle is defined as a void pointer and may be copied or shared among multiple threads as needed. Alternatively, multiple handles to the same datastore may be obtained by calling **he_open**(). Each handle obtained by calling **he_open**() must be closed, when no longer needed, by calling **he_close**().

A volume must be properly initialized (i.e., formatted) before the first use. Once formatted, any number of datastores may be created on the volume. Formatting a volume is accomplished using the **he_open**() function. Prior to formatting a volume, all data residing on the underlying block devices (or fixed-size files) must be backed up. Once formatted, non-Helium content on the volume may no longer be retrievable. A volume, once created, must only be accessed through the Helium API. Volumes remain static from the moment of creation and on. In other words, it is not possible to dynamically expand or shrink a volume once it has been formatted.

The *flags* argument controls the opening mode and may be a bitwise OR of the following constants.

HE_O_CREATE – If the datastore does not exist, create it. Otherwise, this bit has no effect. If this bit is not set, **he_open**() will fail when attempting to open a datastore that does not exist.

HE_O_TRUNCATE – If the datastore exists, truncate it on open. Truncating a datastore removes all items stored in it. This flag is only applicable if no other handle to the datastore is open, otherwise, the call fails by returning NULL and setting *errno* to **HE_ERR_DATASTORE_IN_USE**.

HE_O_VOLUME_CREATE – If *url* specifies an invalid volume, a valid volume is created. Otherwise, this flag has no effect.

HE_O_VOLUME_TRUNCATE – If the volume is a valid volume, and no other handles are currently open, the volume is truncated, removing all datastores (and associated items) from the volume. Warning, this flag must be used with extreme care.

HE_O_NOSORT – Specify this flag to keep the datastore in un-sorted order, hence reducing memory overhead at the expense of slower delete operations. Furthermore, if set, **he_next**(3) or **he_prev**(3) functionality will no longer be supported. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

HE_O_SCAN – On startup, force a scan and rebuild of the indexer. Depending on the datastore

size, this may take a while, as the entire dataset is read and a new indexer is constructed. Helium automatically scans and reconstructs the indexer if needed, hence this flag is not essential for correct operations. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

HE_O_CLEAN – On startup, force garbage collection of all datastores residing on the volume. Depending on the datastore size, this may take a while, as the entire dataset is processed. Helium automatically performs garbage collection if needed, hence this flag is not essential for correct operations. This flag is ignored if a handle to some datastore residing on the volume is currently open.

HE_O_COMPRESS – Specify this flag to compress the datastore's item values before writing to the volume. As common sense would dictate, compression *may* yield storage savings at the expense of performance. This flag is used when a new volume is created or an old volume is truncated. Otherwise, it is ignored.

URL SPECIFICATION

Helium datastores reside on a properly formatted volume. A volume is a set of one or more block devices (or fixed-sized files). Helium uses a URL scheme for specifying the path to a volume, regardless of it residing on a local machine or on a remote Helium server. In Unix-like environments, block devices (or fixed-sized files) are listed under the /dev directory (e.g., /dev/sda). Note that a partition on a raw device is considered a raw device as well (e.g., /dev/sda1). A Helium URL is a null-terminated string taking one of the following forms.

he://./path-to-device1,path-to-device2,...

he://www.server.com/path-to-device1,path-to-device2,...

he://www.server.com:123/path-to-device1,path-to-device2,...

All URLs must begin with the **he://** scheme. Then, the URL must specify the machine location of the block devices (or fixed-sized files). Finally, the URL must list the pathnames of the devices. In the first form, the dot is used to specify that the volume is on the local machine and intended to be used by a single process, hence, Helium will directly read and write the volume, in exclusive mode, using low-level operating system API. Some examples of the first form are shown below.

he://.//dev/loop0

he://.//dev/sda,/dev/sdb

When more than one block device (or fixed-size files) are used to form a volume, the ordering of the devices in the URL is insignificant. For instance, the following two URLs point to the same volume.

he://.//dev/sda,/dev/sdb

he://.//dev/sdb,/dev/sda

In the second form, a volume may be served by a single server to a number of clients (on localhost or remote host). The second form specifies the address of a Helium server, but using an IP or a hostname. A Helium server may reside on a remote node. All Helium functions (e.g., **he_open**()) seamlessly operate within a node or across node boundaries while providing precise semantics and consistent behavior, as described in this document. The word **localhost** may be used if both the server and client reside on the same physical node. By default, Helium uses port number 41000. A colon (i.e., ':') may be used, following the host address, to explicitly specify a port number, as shown below.

he://www.some-server.com:12345//dev/sda

/dev/sda,/dev/sdb

/dev/sdb,/dev/sda

OPERATING ENVIRONMENT

The *env* argument of **he_open**(), if not NULL, is used to specify environment values. The *env* argument is only applicable if no other handle to the datastore is open, otherwise, it is ignored. The *env* is a pointer to the following structure.

struct **he_env** {
    uint64_t *fanout*;
    uint64_t *write_cache*;
    uint64_t *read_cache*;
    uint64_t *auto_commit_period*;
    uint64_t *auto_clean_period*;
    uint64_t *clean_util_pct*;
    uint64_t *clean_dirty_pct*;
    uint64_t *retry_count*;
    uint64_t *retry_delay*;
};

The *fanout* member determines the level of internal parallelism that Helium will accommodate. This number determines the number of operations that may simultaneously operate on the datastore, provided enough processor resources are available. This setting impacts performance, not functionality. This member is needed when a new datastore is created, otherwise, it is ignored. Once created, a datastore's fanout remains fixed. The default fanout is 33.

The *write_cache* member determines the amount of memory, per datastore, in bytes, dedicated to write buffering. The default write buffering is 256 MiB and is adequate for majority of use cases. Increase this value only if your key/value items are very large (4 KiB or larger).

The *read_cache* member determines the amount of memory, per datastore, in bytes, dedicated to read buffering. The default read buffering is 1 GiB. For ultimate performance, increase this value to be 10% above the application working set.

The *auto_commit_period* member specifies the duration of time, in seconds, between automatic commits of the datastore. This value limits the amount of data loss in the event of an unexpected shutdown or system crash. You may set this value to 2^64 - 1 to effectively disable auto commit. The default auto commit period is 10 seconds.

The *auto_clean_period* member specifies the duration of time, in seconds, between automatic check for garbage collection. Every this often, the status of datastore is checked (see the next two members) and garbage collection is triggered if needed. The default auto commit period is 2 seconds.

The *clean_util_pct* members specifies the percentage of volume utilization (100 * utilized / capacity) that triggers garbage collection. Both utilization percentage and dirty percentage must be reached before garbage collection is activated in the background. The default utilization percentage is set to 60%.

The *clean_dirty_pct* member specifies the percentage of garbage (100 * deleted_items / (valid_items + deleted_items)) that triggers garbage collection. Both utilization percentage and dirty percentage must be reached before garbage collection is activated in the background. The default dirty percentage is set to 20%.

The *retry_count* member specifies the number of times an operation that failed due to volume being full will be retried. Subsequent attempts may succeed if the background garbage collector is given some time to free up space. The default retry count is set to 500.

The *retry_delay* member, working in conjunction with the *rety_count*, specifies the delay, in micro-seconds, between retry operations. The default retry delay is set to 20000 micro-seconds.

Set any member of **he_env** structure to zero to force the default value. Out of range values will be adjusted to the minimum or maximum settings as needed.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int main() {
        he_t he;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        if (!he) {
                he_perror("he_open");
                return -1;
        }

        /* datastore operations */

        if (he_close(he)) {
                he_perror("he_close");
                return -1;
        }
        return 0;
}
```

RETURN VALUE

On success **he_open**() returns a valid datastore handle. On error, NULL is returned and *errno* is set to an appropriate Helium error code. On success **he_close**() returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3),

he_insert(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_remove - remove datastore

SYNOPSIS

#include <he.h>

int he_remove(he_t *he***);**
int he_rename(he_t *he***, const char ****name***);**

DESCRIPTION

The **he_remove**() and **he_rename**() functions are used to remove (i.e., permanently delete) a datastore or change its name, respectively.

Both of these functions require a valid handle, previously obtained from a call to **he_open**(3). The **he_remove**() function is otherwise identical to **he_close**(3) with the additional side effect that the datastore, and associated items will be deleted from the volume. After a call to **he_remove**(), the handle must be considered closed, eliminating the need to call **he_close**(3).

The **he_rename**() function will change the name of the datastore, represented by the handle *he*, to that pointed to by *name*.

These functions are likely to fail for the following common reasons:

An attempt to remove a datastore that is currently open in multiple places (i.e., multiple open handles exist). It is assumed that the handle used to remove a datastore is the only handle to that datastore. A failure of this kind is denoted by a return value of **HE_ERR_DATASTORE_IN_USE**.

An attempt to rename a datastore to a new name that is already in use by another datastore on a shared volume. A failure of this kind is denoted by a return value of **HE_ERR_DATASTORE_EXISTS**.

RETURN VALUE

On success **he_remove**() and **he_rename**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_rename - rename datastore

SYNOPSIS

#include <he.h>

int he_remove(he_t *he***);**
int he_rename(he_t *he***, const char ****name***);**

DESCRIPTION

The **he_remove**() and **he_rename**() functions are used to remove (i.e., permanently delete) a datastore or change its name, respectively.

Both of these functions require a valid handle, previously obtained from a call to **he_open**(3). The **he_remove**() function is otherwise identical to **he_close**(3) with the additional side effect that the datastore, and associated items will be deleted from the volume. After a call to **he_remove**(), the handle must be considered closed, eliminating the need to call **he_close**(3).

The **he_rename**() function will change the name of the datastore, represented by the handle *he*, to that pointed to by *name*.

These functions are likely to fail for the following common reasons:

An attempt to remove a datastore that is currently open in multiple places (i.e., multiple open handles exist). It is assumed that the handle used to remove a datastore is the only handle to that datastore. A failure of this kind is denoted by a return value of **HE_ERR_DATASTORE_IN_USE**.

An attempt to rename a datastore to a new name that is already in use by another datastore on a shared volume. A failure of this kind is denoted by a return value of **HE_ERR_DATASTORE_EXISTS**.

RETURN VALUE

On success **he_remove**() and **he_rename**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_stats - return datastore statistics

SYNOPSIS

#include <he.h>

int he_stats(he_t *he***, struct he_stats ****stats***);**

DESCRIPTION

The **he_stats**() function is used to obtain datastore/transaction, volume, and performance statistics. The first argument to **he_stats**() must be a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument must be a pointer to a structure of type **he_stats**(). The **he_stats**() structure has a number of members as described below.

const char ****name*;**

The name of the datastore. Do not modify this buffer or free it.

uint64_t *valid_items***;**

A count of the number of items that are valid.

uint64_t *deleted_items***;**

A count of the number of items that have been deleted and continue to take up space on the volume. The number of deleted items may change as the garbage collector runs in the background.

uint64_t *utilized***;**

Total bytes currently used up by all datastores, in bytes, including key bytes, value bytes, and metadata bytes of valid and deleted items. The utilized portion of the total capacity may change (i.e., shrink) as the garbage collector runs in the background. Volume storage capacity is shared among all datastores residing on it.

uint64_t *capacity***;**

Total capacity, in bytes, of the volume to be shared among datastores residing on it.

uint64_t *buffered_writes***;**

Total number of bytes written to the volume. Since data is first copied to DRAM buffers prior to being written to the device, this number may be larger than the total number of bytes written to the device.

uint64_t *buffered_reads***;**

Total number of bytes read from the volume. Since, if available, data is served from DRAM rather than being read from the device, this number may be larger than the total number of bytes read from the device.

uint64_t *device_writes***;**

Total number of bytes written to the volume.

uint64_t *device_reads***;**

Total number of bytes read from the volume.

uint64_t *auto_commits***;**

The number of automatically performed flushes of buffered data to persistent storage since the volume was first opened for datastore transactions.

uint64_t *auto_cleans***;**

The number of automatically performed garbage collection rounds since the volume was first opened for datastore transactions.

RETURN VALUE

On success **he_stats**() returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**.

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_transaction - obtain a transaction handle

SYNOPSIS

#include <he.h>

he_t he_transaction(he_t *he***);**
int he_commit(he_t *he***);**
int he_discard(he_t *he***);**

DESCRIPTION

Helium provides simple and powerful transaction capabilities. A transaction is a sequence of operations that take place in isolation from other transactions and may be discarded or committed (added to the datastore) atomically. An opaque handle, similar to the one returned by **he_open**(3), is used to perform update and lookup operations in a transaction. This handle is obtained by calling **he_transaction**().

The **he_transaction**() function takes as its only argument a handle *he*, to a datastore obtained from a call to **he_open**(3). On return, a new handle is returned, ready to be used by **he_update**(3) or **he_delete**(3).

In the case of **he_update**(3) on a transaction, an update operation will be queued to be applied against the datastore at the time of commit. In the case of **he_delete**(3) on a transaction, a delete operation will be queue to be applied against the datastore at the time of commit.

The **he_discard**() function must be called to terminate a transaction and atomically expunge all data associated with the transaction. On return, the handle passed to **he_discard**() is closed and no longer valid.

The **he_commit**() function must be called to terminate a transaction and atomically apply the tranaction content to the datastore. On return, the handle passed to **he_commit**() is closed and no longer valid. Committed transactions become part of the main datastore and accessible via the datastore handle, returned by **he_open**(3).

Helium's transaction commit rules are to either apply all updates/deletes, if these update/deletes can be successfully applied, or to fail completely and leave the datastore unchanged. An update of an item that exists in the datastore will result in a value replacement. A delete of an item that does not exist in the datastore will result in a failed transaction. A failed transaction is equivalent to a discard operation.

Finally, a transaction handle may be used by any of **he_stats**, **he_exists**, **he_lookup**, **he_next** or **he_prev**. These functions perform operations against the transaction content rather than the datastore content when used with a transaction handle.

To clarify, consider these examples:

```
/* example: successful commit, item2 and item3 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item1);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: failed commit, datastore content unchanged */
```

```
datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item4);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: successful commit, only item1 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_update(transaction, item2);
he_delete(transaction, item2); /* nullifies the update of item2 */
he_commit(transaction); /* transaction handle is closed */
```

The **he_commit**() and **he_discard**() may be called on the main datastore in order to commit data to the volume or discard (truncate) the datastore. However, when passing the datastore handle to these functions, the handle remains open.

Uncommitted transaction are discarded when the datastore handle is closed. All pending transactions are closed when the datastore handle is closed.

RETURN VALUE

On success **he_transaction**() returns a handle. On error, NULL is returned and *errno* is set accordingly. On success **he_commit**() and **he_discard**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_commit - commit and close transaction

SYNOPSIS

#include <he.h>

he_t he_transaction(he_t *he***);**
int he_commit(he_t *he***);**
int he_discard(he_t *he***);**

DESCRIPTION

Helium provides simple and powerful transaction capabilities. A transaction is a sequence of operations that take place in isolation from other transactions and may be discarded or committed (added to the datastore) atomically. An opaque handle, similar to the one returned by **he_open**(3), is used to perform update and lookup operations in a transaction. This handle is obtained by calling **he_transaction**().

The **he_transaction**() function takes as its only argument a handle *he*, to a datastore obtained from a call to **he_open**(3). On return, a new handle is returned, ready to be used by **he_update**(3) or **he_delete**(3).

In the case of **he_update**(3) on a transaction, an update operation will be queued to be applied against the datastore at the time of commit. In the case of **he_delete**(3) on a transaction, a delete operation will be queue to be applied against the datastore at the time of commit.

The **he_discard**() function must be called to terminate a transaction and atomically expunge all data associated with the transaction. On return, the handle passed to **he_discard**() is closed and no longer valid.

The **he_commit**() function must be called to terminate a transaction and atomically apply the tranaction content to the datastore. On return, the handle passed to **he_commit**() is closed and no longer valid. Committed transactions become part of the main datastore and accessible via the datastore handle, returned by **he_open**(3).

Helium's transaction commit rules are to either apply all updates/deletes, if these update/deletes can be successfully applied, or to fail completely and leave the datastore unchanged. An update of an item that exists in the datastore will result in a value replacement. A delete of an item that does not exist in the datastore will result in a failed transaction. A failed transaction is equivalent to a discard operation.

Finally, a transaction handle may be used by any of **he_stats**, **he_exists**, **he_lookup**, **he_next** or **he_prev**. These functions perform operations against the transaction content rather than the datastore content when used with a transaction handle.

To clarify, consider these examples:

```
/* example: successful commit, item2 and item3 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item1);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: failed commit, datastore content unchanged */
```

```
datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item4);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: successful commit, only item1 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_update(transaction, item2);
he_delete(transaction, item2); /* nullifies the update of item2 */
he_commit(transaction); /* transaction handle is closed */
```

The **he_commit**() and **he_discard**() may be called on the main datastore in order to commit data to the volume or discard (truncate) the datastore. However, when passing the datastore handle to these functions, the handle remains open.

Uncommitted transaction are discarded when the datastore handle is closed. All pending transactions are closed when the datastore handle is closed.

RETURN VALUE

On success **he_transaction**() returns a handle. On error, NULL is returned and *errno* is set accordingly. On success **he_commit**() and **he_discard**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_discard - discard and close transaction

SYNOPSIS

#include <he.h>

he_t he_transaction(he_t *he***);**
int he_commit(he_t *he***);**
int he_discard(he_t *he***);**

DESCRIPTION

Helium provides simple and powerful transaction capabilities. A transaction is a sequence of operations that take place in isolation from other transactions and may be discarded or committed (added to the datastore) atomically. An opaque handle, similar to the one returned by **he_open**(3), is used to perform update and lookup operations in a transaction. This handle is obtained by calling **he_transaction**().

The **he_transaction**() function takes as its only argument a handle *he*, to a datastore obtained from a call to **he_open**(3). On return, a new handle is returned, ready to be used by **he_update**(3) or **he_delete**(3).

In the case of **he_update**(3) on a transaction, an update operation will be queued to be applied against the datastore at the time of commit. In the case of **he_delete**(3) on a transaction, a delete operation will be queue to be applied against the datastore at the time of commit.

The **he_discard**() function must be called to terminate a transaction and atomically expunge all data associated with the transaction. On return, the handle passed to **he_discard**() is closed and no longer valid.

The **he_commit**() function must be called to terminate a transaction and atomically apply the tranaction content to the datastore. On return, the handle passed to **he_commit**() is closed and no longer valid. Committed transactions become part of the main datastore and accessible via the datastore handle, returned by **he_open**(3).

Helium's transaction commit rules are to either apply all updates/deletes, if these update/deletes can be successfully applied, or to fail completely and leave the datastore unchanged. An update of an item that exists in the datastore will result in a value replacement. A delete of an item that does not exist in the datastore will result in a failed transaction. A failed transaction is equivalent to a discard operation.

Finally, a transaction handle may be used by any of **he_stats**, **he_exists**, **he_lookup**, **he_next** or **he_prev**. These functions perform operations against the transaction content rather than the datastore content when used with a transaction handle.

To clarify, consider these examples:

```
/* example: successful commit, item2 and item3 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item1);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: failed commit, datastore content unchanged */
```

```
datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_delete(transaction, item4);
he_update(transaction, item3);
he_commit(transaction); /* transaction handle is closed */

/* example: successful commit, only item1 in datastore */

datastore = he_open( ... );
he_insert(datastore, item1);
he_insert(datastore, item2);
transaction = he_transaction(datastore);
he_update(transaction, item2);
he_delete(transaction, item2); /* nullifies the update of item2 */
he_commit(transaction); /* transaction handle is closed */
```

The **he_commit**() and **he_discard**() may be called on the main datastore in order to commit data to the volume or discard (truncate) the datastore. However, when passing the datastore handle to these functions, the handle remains open.

Uncommitted transaction are discarded when the datastore handle is closed. All pending transactions are closed when the datastore handle is closed.

RETURN VALUE

On success **he_transaction**() returns a handle. On error, NULL is returned and *errno* is set accordingly. On success **he_commit**() and **he_discard**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_update - update an item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_update(he_t *he***, const struct he_item ****item***);**
int he_insert(he_t *he***, const struct he_item ****item***);**
int he_replace(he_t *he***, const struct he_item ****item***);**
int he_delete(he_t *he***, const struct he_item ****item***);**
int he_delete_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions take as a first argument, *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument *item*, must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete** function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions. Helium's maximum key size is 64 KiB. Applications should use the **HE_MAX_KEY_LEN** define as necessary.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value. Helium's maximum value size is 16 MiB. Applications should use the **HE_MAX_VAL_LEN** define as necessary.

The **he_update**() function will insert *item* into the datastore/transaction if that item does not exist or update it with the new value if it does exist.

The **he_insert**() function will insert *item* into the datastore/transaction if that item does not exist or fail if it does exist.

The **he_replace**() function will fail if *item* does not exist in the datastore/transaction or update it with a new value if it does exist.

The **he_delete**() function will delete *item* from the datastore/transaction.

The **he_delete_lookup**() function will perform an atomic lookup followed by a delete operation on the key. See **he_lookup** for a detailed description of *item*, *off*, *len* and function usage.

Updated or deleted items of a transaction are applied to the datastore on a commit or discarded. See *he_transaction*, *he_commit*, and *he_discard* for details.

RETURN VALUE

On success **he_update**, **he_insert**, **he_replace**, and **he_delete** return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_insert - insert a new item into datastore/transaction

SYNOPSIS

#include <he.h>

int he_update(he_t *he***, const struct he_item ****item***);**
int he_insert(he_t *he***, const struct he_item ****item***);**
int he_replace(he_t *he***, const struct he_item ****item***);**
int he_delete(he_t *he***, const struct he_item ****item***);**
int he_delete_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions take as a first argument, *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument *item*, must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete**() function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions. Helium's maximum key size is 64 KiB. Applications should use the **HE_MAX_KEY_LEN** define as necessary.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value. Helium's maximum value size is 16 MiB. Applications should use the **HE_MAX_VAL_LEN** define as necessary.

The **he_update**() function will insert *item* into the datastore/transaction if that item does not exist or update it with the new value if it does exist.

The **he_insert**() function will insert *item* into the datastore/transaction if that item does not exist or fail if it does exist.

The **he_replace**() function will fail if *item* does not exist in the datastore/transaction or update it with a new value if it does exist.

The **he_delete**() function will delete *item* from the datastore/transaction.

The **he_delete_lookup**() function will perform an atomic lookup followed by a delete operation on the key. See **he_lookup** for a detailed description of *item*, *off*, *len* and function usage.

Updated or deleted items of a transaction are applied to the datastore on a commit or discarded. See *he_transaction*, *he_commit*, and *he_discard* for details.

RETURN VALUE

On success **he_update**, **he_insert**, **he_replace**, and **he_delete** return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_replace - replace an existing item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_update(he_t *he***, const struct he_item ****item***);**
int he_insert(he_t *he***, const struct he_item ****item***);**
int he_replace(he_t *he***, const struct he_item ****item***);**
int he_delete(he_t *he***, const struct he_item ****item***);**
int he_delete_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions take as a first argument, *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument *item*, must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete** function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions. Helium's maximum key size is 64 KiB. Applications should use the **HE_MAX_KEY_LEN** define as necessary.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value. Helium's maximum value size is 16 MiB. Applications should use the **HE_MAX_VAL_LEN** define as necessary.

The **he_update**() function will insert *item* into the datastore/transaction if that item does not exist or update it with the new value if it does exist.

The **he_insert**() function will insert *item* into the datastore/transaction if that item does not exist or fail if it does exist.

The **he_replace**() function will fail if *item* does not exist in the datastore/transaction or update it with a new value if it does exist.

The **he_delete**() function will delete *item* from the datastore/transaction.

The **he_delete_lookup**() function will perform an atomic lookup followed by a delete operation on the key. See **he_lookup** for a detailed description of *item*, *off*, *len* and function usage.

Updated or deleted items of a transaction are applied to the datastore on a commit or discarded. See *he_transaction*, *he_commit*, and *he_discard* for details.

RETURN VALUE

On success **he_update**, **he_insert**, **he_replace**, and **he_delete** return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_delete - delete an item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_update(he_t *he***, const struct he_item ****item***);**
int he_insert(he_t *he***, const struct he_item ****item***);**
int he_replace(he_t *he***, const struct he_item ****item***);**
int he_delete(he_t *he***, const struct he_item ****item***);**
int he_delete_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions take as a first argument, *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument *item*, must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete** function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions. Helium's maximum key size is 64 KiB. Applications should use the **HE_MAX_KEY_LEN** define as necessary.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value. Helium's maximum value size is 16 MiB. Applications should use the **HE_MAX_VAL_LEN** define as necessary.

The **he_update**() function will insert *item* into the datastore/transaction if that item does not exist or update it with the new value if it does exist.

The **he_insert**() function will insert *item* into the datastore/transaction if that item does not exist or fail if it does exist.

The **he_replace**() function will fail if *item* does not exist in the datastore/transaction or update it with a new value if it does exist.

The **he_delete**() function will delete *item* from the datastore/transaction.

The **he_delete_lookup**() function will perform an atomic lookup followed by a delete operation on the key. See **he_lookup** for a detailed description of *item*, *off*, *len* and function usage.

Updated or deleted items of a transaction are applied to the datastore on a commit or discarded. See *he_transaction*, *he_commit*, and *he_discard* for details.

RETURN VALUE

On success **he_update**, **he_insert**, **he_replace**, and **he_delete** return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_delete_lookup - update an item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_update(he_t *he***, const struct he_item ****item***);**
int he_insert(he_t *he***, const struct he_item ****item***);**
int he_replace(he_t *he***, const struct he_item ****item***);**
int he_delete(he_t *he***, const struct he_item ****item***);**
int he_delete_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions take as a first argument, *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument *item*, must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete** function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions. Helium's maximum key size is 64 KiB. Applications should use the **HE_MAX_KEY_LEN** define as necessary.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value. Helium's maximum value size is 16 MiB. Applications should use the **HE_MAX_VAL_LEN** define as necessary.

The **he_update**() function will insert *item* into the datastore/transaction if that item does not exist or update it with the new value if it does exist.

The **he_insert**() function will insert *item* into the datastore/transaction if that item does not exist or fail if it does exist.

The **he_replace**() function will fail if *item* does not exist in the datastore/transaction or update it with a new value if it does exist.

The **he_delete**() function will delete *item* from the datastore/transaction.

The **he_delete_lookup**() function will perform an atomic lookup followed by a delete operation on the key. See **he_lookup** for a detailed description of *item*, *off*, *len* and function usage.

Updated or deleted items of a transaction are applied to the datastore on a commit or discarded. See *he_transaction*, *he_commit*, and *he_discard* for details.

RETURN VALUE

On success **he_update**, **he_insert**, **he_replace**, and **he_delete** return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_exists, check if an item exists in datastore/transaction

SYNOPSIS

#include <he.h>

int he_exists(he_t *he***, struct he_item ****item***);**

DESCRIPTION

The **he_exists** function is used to efficiently check if a valid item exists in the datastore. The first argument to **he_exists** must be a valid handle previously obtained from a call to **he_open** or **he_transaction**. The second argument must be a pointer to an **he_item** structure. The **he_item** structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer holding a value, as needed. A value is not necessary for the **he_delete** function, thus, will be ignored. If updating, inserting, or replacing an item with an empty value, the value member may be NULL.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value.

For use with **he_exists**, only the item's *key* and *key_len* members are needed, while the remaining members are ignored. This function returns 0 on success, indicating that the item exists. The most likely error returned by this function is **HE_ERR_ITEM_NOT_FOUND**.

RETURN VALUE

On success **he_exists** returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**.

THREADS & REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the

transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_lookup - lookup an item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_next(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_prev(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions find and return an item in the datastore. Each of these functions takes as its first argument *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument must be a pointer to an **he_item** structure. The he_item structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer to hold the retrieved value. The buffer must be large enough to accommodate the requested bytes, i.e., *len*.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value.

The **he_lookup**() function requires the caller to populate the *key*, *val*, *key_len*, and *flags* members of *item*. The *val* member must point to a buffer that is large enough to receive the requested bytes. On a successful lookup, the **he_lookup**() function will populate the user buffer and write the actual size of the item's value, in bytes, to *val_len*.

The *off* and *len* arguments determine the requested value bytes. The *off* argument specifies the starting byte address, within the value, that is being requested. The *len* argument specifies total number of bytes being requested. The user buffer that is to receive the bytes (i.e., item's *val* member) must be at least as big as *len*. However, depending on the actual size of the item's value, in bytes, the number of bytes written to this buffer may be less than *len*.

The **he_next**() and **he_prev**() functions operate in exactly the same manner as **he_lookup**(). The only differences are that **he_next**() returns the item that is after *item* (i.e., succeed its key). Likewise, **he_prev**() returns the item that is before *item* (i.e., precedes its key).

The item ordering is determined by the datastore's collation scheme, which is specified in the call to **he_open**(3) and defaults to a unsigned byte compare of keys. The functions **he_next**() and **he_prev**() overwrite the item's *key* and *key_len* members to reflect the actual item that was retrieved. The calling application must provide a suitable buffer to receive the next/previous key.

Setting of the item's *key_len* member to 0 is a special case to request that **he_next**() is to retrieve the smallest item in the datastore. Likewise, setting of the item's *key_len* member to 0 is a special case to request that **he_prev** is to retrieve the largest item in the datastore.

The **he_lookup**() function will return **HE_ERR_ITEM_NOT_FOUND** if the item does not exist. Similarly, **he_next**() and **he_prev**() will return **HE_ERR_ITEM_NOT_FOUND** if a successor or predecessor item are not found, in other words, when iteration has reached the end points of the datastore.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int main() {
        struct he_item item;
        char key[HE_MAX_KEY_LEN], val[1024];
        he_t he;
        int err;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        if (!he) {
                he_perror("he_open");
                return -1;
        }

        memset(&item, 0, (sizeof (item)));
        item.key = key;
        item.val = val;
        while (!(err = he_next(he, &item, 0, sizeof (val)))) {
                printf("%s => %s\n", (char *)item.key, (char *)item.val);
        }
        if (HE_ERR_ITEM_NOT_FOUND != err) {
                he_perror("he_next");
                return -1;
        }
        if (he_close(he)) {
                he_perror("he_close");
                return -1;
        }
        return 0;
}
```

RETURN VALUE

On success **he_lookup**(), **he_next**(), and **he_prev**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_next - lookup the next item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_next(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_prev(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions find and return an item in the datastore. Each of these functions takes as its first argument *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument must be a pointer to an **he_item** structure. The he_item structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer to hold the retrieved value. The buffer must be large enough to accommodate the requested bytes, i.e., *len*.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value.

The **he_lookup**() function requires the caller to populate the *key*, *val*, *key_len*, and *flags* members of *item*. The *val* member must point to a buffer that is large enough to receive the requested bytes. On a successful lookup, the **he_lookup**() function will populate the user buffer and write the actual size of the item's value, in bytes, to *val_len*.

The *off* and *len* arguments determine the requested value bytes. The *off* argument specifies the starting byte address, within the value, that is being requested. The *len* argument specifies total number of bytes being requested. The user buffer that is to receive the bytes (i.e., item's *val* member) must be at least as big as *len*. However, depending on the actual size of the item's value, in bytes, the number of bytes written to this buffer may be less than *len*.

The **he_next**() and **he_prev**() functions operate in exactly the same manner as **he_lookup**(). The only differences are that **he_next**() returns the item that is after *item* (i.e., succeed its key). Likewise, **he_prev**() returns the item that is before *item* (i.e., precedes its key).

The item ordering is determined by the datastore's collation scheme, which is specified in the call to **he_open**(3) and defaults to a unsigned byte compare of keys. The functions **he_next**() and **he_prev**() overwrite the item's *key* and *key_len* members to reflect the actual item that was retrieved. The calling application must provide a suitable buffer to receive the next/previous key.

Setting of the item's *key_len* member to 0 is a special case to request that **he_next**() is to retrieve the smallest item in the datastore. Likewise, setting of the item's *key_len* member to 0 is a special case to request that **he_prev** is to retrieve the largest item in the datastore.

The **he_lookup**() function will return **HE_ERR_ITEM_NOT_FOUND** if the item does not exist. Similarly, **he_next**() and **he_prev**() will return **HE_ERR_ITEM_NOT_FOUND** if a successor or predecessor item are not found, in other words, when iteration has reached the end points of the datastore.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int main() {
        struct he_item item;
        char key[HE_MAX_KEY_LEN], val[1024];
        he_t he;
        int err;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        if (!he) {
                he_perror("he_open");
                return -1;
        }

        memset(&item, 0, (sizeof (item)));
        item.key = key;
        item.val = val;
        while (!(err = he_next(he, &item, 0, sizeof (val)))) {
                printf("%s => %s\n", (char *)item.key, (char *)item.val);
        }
        if (HE_ERR_ITEM_NOT_FOUND != err) {
                he_perror("he_next");
                return -1;
        }
        if (he_close(he)) {
                he_perror("he_close");
                return -1;
        }
        return 0;
}
```

RETURN VALUE

On success **he_lookup**(), **he_next**(), and **he_prev**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_prev - lookup the previous item in datastore/transaction

SYNOPSIS

#include <he.h>

int he_lookup(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_next(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**
int he_prev(he_t *he***,**
struct he_item ****item*,**
uint64_t *off* **,**
uint64_t *len***);**

DESCRIPTION

These functions find and return an item in the datastore. Each of these functions takes as its first argument *he*, a valid handle previously obtained from a call to **he_open**(3) or **he_transaction**(3). The second argument must be a pointer to an **he_item** structure. The he_item structure members are described below.

void ****key*;**

This member must point to a user buffer holding the key. It is required by all update and lookup functions.

void ****val*;**

This member must point to a user buffer to hold the retrieved value. The buffer must be large enough to accommodate the requested bytes, i.e., *len*.

size_t *key_len***;**

This member must be set to the size of the key, in bytes. It is required by all update and lookup functions.

size_t *val_len***;**

This member must be set to the size of the value, in bytes. An item may have a 0-byte NULL value.

The **he_lookup**() function requires the caller to populate the *key*, *val*, *key_len*, and *flags* members of *item*. The *val* member must point to a buffer that is large enough to receive the requested bytes. On a successful lookup, the **he_lookup**() function will populate the user buffer and write the actual size of the item's value, in bytes, to *val_len*.

The *off* and *len* arguments determine the requested value bytes. The *off* argument specifies the starting byte address, within the value, that is being requested. The *len* argument specifies total number of bytes being requested. The user buffer that is to receive the bytes (i.e., item's *val* member) must be at least as big as *len*. However, depending on the actual size of the item's value, in bytes, the number of bytes written to this buffer may be less than *len*.

The **he_next**() and **he_prev**() functions operate in exactly the same manner as **he_lookup**(). The only differences are that **he_next**() returns the item that is after *item* (i.e., succeed its key). Likewise, **he_prev**() returns the item that is before *item* (i.e., precedes its key).

The item ordering is determined by the datastore's collation scheme, which is specified in the call to **he_open**(3) and defaults to a unsigned byte compare of keys. The functions **he_next**() and **he_prev**() overwrite the item's *key* and *key_len* members to reflect the actual item that was retrieved. The calling application must provide a suitable buffer to receive the next/previous key.

Setting of the item's *key_len* member to 0 is a special case to request that **he_next**() is to retrieve the smallest item in the datastore. Likewise, setting of the item's *key_len* member to 0 is a special case to request that **he_prev** is to retrieve the largest item in the datastore.

The **he_lookup**() function will return **HE_ERR_ITEM_NOT_FOUND** if the item does not exist. Similarly, **he_next**() and **he_prev**() will return **HE_ERR_ITEM_NOT_FOUND** if a successor or predecessor item are not found, in other words, when iteration has reached the end points of the datastore.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int main() {
        struct he_item item;
        char key[HE_MAX_KEY_LEN], val[1024];
        he_t he;
        int err;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        if (!he) {
                he_perror("he_open");
                return -1;
        }

        memset(&item, 0, (sizeof (item)));
        item.key = key;
        item.val = val;
        while (!(err = he_next(he, &item, 0, sizeof (val)))) {
                printf("%s => %s\n", (char *)item.key, (char *)item.val);
        }
        if (HE_ERR_ITEM_NOT_FOUND != err) {
                he_perror("he_next");
                return -1;
        }
        if (he_close(he)) {
                he_perror("he_close");
                return -1;
        }
        return 0;
}
```

RETURN VALUE

On success **he_lookup**(), **he_next**(), and **he_prev**() return 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_iterate - iterate over items in datastore

SYNOPSIS

#include <he.h>

int he_iterate(he_t *he***, he_iterate_cbf_t** *cbf* **, void *** *arg***);**

DESCRIPTION

The **he_iterate**() function takes as its first argument *he*, a valid datastore handle and, atomically creates a snapshot of the datastore at the moment it is called. Thereafter, the *cbf* callback function is called for each valid item in the datastore that belongs to the snapshot. The *item* argument of the callback function will have valid *key* and *val* buffers as well as valid *key_len* and *val_len* sizes. During the iterate process, the datastore will remain available for lookup operations by other threads, however. The *arg* argument of **he_iterate**() is passed directly to the callback function *cbf*.

The callback function *cbf* must have the following signature.

int some_function_name(void * *arg***, const he_item *** *item***);**

The callback function *cbf* must return 0 as long additional items are to be received. Any value, other than 0, will cause the call to **he_iterate**() to terminated, returning **HE_ERR_TERMINATED**.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <he.h>

int dump_to_disk(void *arg, const struct he_item *item) {
        FILE *ofs = (FILE*)arg;

        fprintf(ofs, "k:%s %i\n", (char*)item.key, (int)item.key_len);
        fprintf(ofs, "v:%s %i\n", (char*)item.val, (int)item.val_len);
        return 0;
}

int main() {
        he_t he;
        FILE *ofs;

        he = he_open("he://.//file", "DATASTORE", 0, NULL);
        ofs = fopen("iterate.file", "w");
        if (he_iterate(he, dump_to_disk, ofs)) {
                he_perror("he_iterate");
        }
        he_close(he);
        return 0;
}
```

RETURN VALUE

On success **he_iterate**() returns 0. On error, a Helium error code is returned and *errno* is set accordingly. For a list of Helium error codes see **he_strerror**(3).

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_version - return Helium library version

SYNOPSIS

#include <he.h>

const char *he_version(int **major*, int **minor*, int **patch*);

DESCRIPTION

Helium version consists of three positive integers: the major number, the minor number, and the patch number. As a string, Helium version is denoted as X.Y.Z, where X is the major number, Y is the minor number, and Z is the patch number. You must call the **he_version**() function to obtain the version of the library that is linked with your application.

Any one of the arguments of **he_version**() may be NULL, in which case the function will ignore that argument. Otherwise, the memory location pointed to by the major, minor, or patch arguments will hold the Helium version on return. In addition to the **he_version**() function, your application may use the **HE_VERSION_MAJOR**, **HE_VERSION_MINOR**, or **HE_VERSION_PATCH** constants. A recommended approach is to check the return values of **he_version**() against these constants to ensure that the header file and the static/dynamic libraries are consistent.

RETURN VALUE

A read-only NULL terminated string representation of the version number in the form of major.minor.patch is returned.

THREADS AND REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_perror - print a string describing recent Helium error

SYNOPSIS

#include <he.h>

void he_perror(const char *_s_);
const char *he_strerror(int _err_);

DESCRIPTION

Nearly all Helium functions return a negative integer error code if an error is encountered and 0 otherwise. In addition, the *errno* variable is set to the same error number.

The **he_perror**() function prints a short string message to the screen corresponding to the most recent error number (i.e., the value currently in the *errno* variable). The only argument to **he_perror**() is a, possibly NULL, string that gets printed ahead of the error message. This function, in turn, calls the standard C's **perror**(3) when *errno* is not a Helium error number.

The **he_strerror**(3) function converts an error code to a short string message. The only argument to **he_strerror**(3) is an integer error number. The function returns a pointer to a string that may be used by the caller in a formatted output. The caller must not write to the buffer returned by **he_strerror**(3). This function, in turn, calls the standard C's **strerror**(3) when *err* is not a Helium error number.

HELIUM ERROR CODES

HE_ERR_SOFTWARE – An internal software error has been encountered. This is an unlikely error and is returned by Helium functions if a more appropriate error code is not found.

HE_ERR_ARGUMENTS – A Helium function was called with an invalid argument.

HE_ERR_MEMORY – Helium was unable to allocate needed system memory. Consider closing handles, adding more DRAM, etc.

HE_ERR_CHECKSUM – A checksum computation failed. This typically happens if some meta information (e.g., compressed index) was not successfully store or loaded during shutdown or startup. In these cases, the meta information is reconstructed by scanning the volume and harvesting datastore items.

HE_ERR_TERMINATED – An operation failed as a result of a user termination. For example, during a backup process, if the user callback function returns a value other than 0, the backup operation will fail with this error code.

HE_ERR_UNSUPPORTED – An attempt to perform an operation (e.g. backup) has failed because another the operation (e.g., next/prev on onsorted datastore) is unsupported.

HE_ERR_DEVICE_STAT – Helium determined that one or more devices do not exist.

HE_ERR_DEVICE_OPEN – Helium failed to open a block device (or fixed-size file) for reasons other than permissions (reported as **HE_ERR_DEVICE_ACCESS**) or exclusive lock (reported as **HE_ERR_DEVICE_LOCK**).

HE_ERR_DEVICE_ACCESS – Helium failed to open a block device (or fixed-size file) due to permissions. Run as root/administrator or change the device's permission accordingly.

HE_ERR_DEVICE_LOCK – Helium failed to open a block device (or fixed-size file) because it could not obtain an exclusive lock on that device. Helium allows a volume to be open and accessible to a single process at a time. If multiple processes need to share a volume, start a Helium server.

HE_ERR_DEVICE_GEOMETRY – Helium determined that one or more of a volume's block devices (or fixed-size files) have an unusual capacity or block size. For example, the total capacity of the device is not an integer multiple of the block size.

HE_ERR_DEVICE_READ – Helium failed to read a device. This is a serious error and the application must cleanup and exit immediately.

HE_ERR_DEVICE_WRITE – Helium failed to write a device. This is a serious error and the application must cleanup and exit immediately.

HE_ERR_VOLUME_TOO_SMALL – The volume is too small. This error is unlikely if the volume is at least 1 GiB. A good design principle is to make your volume about 20 percent larger than the eventual data size that is expected to reside on the volume.

HE_ERR_VOLUME_INVALID – The volume is invalid. The specified volume has never been formatted/initialized as a volume, or it was subsequently corrupted. Create a datastore on a new volume using the **HE_O_VOLUME_CREATE** flag.

HE_ERR_VOLUME_CORRUPT – The volume is corrupt. This volume was, at some point, formatted properly as a Helium volume, but subsequently corrupted.

HE_ERR_VOLUME_IN_USE – The volume is in use, but an **he_open** flag to truncate the volume is specified.

HE_ERR_VOLUME_FULL – The volume is fully utilized. There are no garbage to reclaim in order to accommodate the latest update to the datastore. Delete some data to free up space. A good design principle is to make your volume about 20 percent larger than the eventual data size that is expected to reside on the volume.

HE_ERR_DATASTORE_NOT_FOUND – An attempt to open a datastore failed because such a datastore does not exist. Consider using the **HE_O_CREATE** flag when opening the datastore.

HE_ERR_DATASTORE_EXISTS – An attempt to rename a datastore to a new name that already exists on the volume has failed. Don't rename a datastore using an existing name.

HE_ERR_DATASTORE_IN_USE – An attempt to remove (i.e., permanently delete) a datastore has failed because additional open handles exist. Close all open handles, except for the one held by the thread attempting to remove the datastore.

HE_ERR_ITEM_NOT_FOUND – An operation on an item cannot proceed because the item does not exist in the datastore.

HE_ERR_ITEM_EXISTS – An operation, most likely an insert, has failed because that item already exists in the datastore.

HE_ERR_NETWORK_LISTEN – An attempt to listen to incoming network connections has failed.

HE_ERR_NETWORK_ACCEPT – An attempt to accept an incoming network connection has

failed.

HE_ERR_NETWORK_CONNECT – An attempt to connect to a Helium server has failed.

HE_ERR_NETWORK_READ – An attempt to read from the network has failed.

HE_ERR_NETWORK_WRITE – An attempt to write to the network has failed.

HE_ERR_NETWORK_PROTOCOL – Data received over the network is corrupt or poorly formed with respect to the established communication protocol.

RETURN VALUE

A constant C string is returned by **he_strerror**.

THREADS & REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)

NAME

he_strerror - return a string describing recent Helium error

SYNOPSIS

#include <he.h>

void he_perror(const char *_s_);
const char *he_strerror(int _err_);

DESCRIPTION

Nearly all Helium functions return a negative integer error code if an error is encountered and 0 otherwise. In addition, the *errno* variable is set to the same error number.

The **he_perror**() function prints a short string message to the screen corresponding to the most recent error number (i.e., the value currently in the *errno* variable). The only argument to **he_perror**() is a, possibly NULL, string that gets printed ahead of the error message. This function, in turn, calls the standard C's **perror**(3) when *errno* is not a Helium error number.

The **he_strerror**(3) function converts an error code to a short string message. The only argument to **he_strerror**(3) is an integer error number. The function returns a pointer to a string that may be used by the caller in a formatted output. The caller must not write to the buffer returned by **he_strerror**(3). This function, in turn, calls the standard C's **strerror**(3) when *err* is not a Helium error number.

HELIUM ERROR CODES

HE_ERR_SOFTWARE – An internal software error has been encountered. This is an unlikely error and is returned by Helium functions if a more appropriate error code is not found.

HE_ERR_ARGUMENTS – A Helium function was called with an invalid argument.

HE_ERR_MEMORY – Helium was unable to allocate needed system memory. Consider closing handles, adding more DRAM, etc.

HE_ERR_CHECKSUM – A checksum computation failed. This typically happens if some meta information (e.g., compressed index) was not successfully stored or loaded during shutdown or startup. In these cases, the meta information is reconstructed by scanning the volume and harvesting datastore items.

HE_ERR_TERMINATED – An operation failed as a result of a user termination. For example, during a backup process, if the user callback function returns a value other than 0, the backup operation will fail with this error code.

HE_ERR_UNSUPPORTED – An attempt to perform an operation (e.g. backup) has failed because the operation (e.g., next/prev on unsorted datastore) is unsupported.

HE_ERR_DEVICE_STAT – Helium determined that one or more devices do not exist.

HE_ERR_DEVICE_OPEN – Helium failed to open a block device (or fixed-size file) for reasons other than permissions (reported as **HE_ERR_DEVICE_ACCESS**) or exclusive lock (reported as **HE_ERR_DEVICE_LOCK**).

HE_ERR_DEVICE_ACCESS – Helium failed to open a block device (or fixed-size file) due to permissions. Run as root/administrator or change the device's permission accordingly.

HE_ERR_DEVICE_LOCK – Helium failed to open a block device (or fixed-size file) because it could not obtain an exclusive lock on that device. Helium allows a volume to be open and accessible to a single process at a time. If multiple processes need to share a volume, start a Helium server.

HE_ERR_DEVICE_GEOMETRY – Helium determined that one or more of a volume's block devices (or fixed-size files) have an unusual capacity or block size. For example, the total capacity of the device is not an integer multiple of the block size.

HE_ERR_DEVICE_READ – Helium failed to read a device. This is a serious error and the application must cleanup and exit immediately.

HE_ERR_DEVICE_WRITE – Helium failed to write a device. This is a serious error and the application must cleanup and exit immediately.

HE_ERR_VOLUME_TOO_SMALL – The volume is too small. This error is unlikely if the volume is at least 1 GiB. A good design principle is to make your volume about 20 percent larger than the eventual data size that is expected to reside on the volume.

HE_ERR_VOLUME_INVALID – The volume is invalid. The specified volume has never been formatted/initialized as a volume, or it was subsequently corrupted. Create a datastore on a new volume using the **HE_O_VOLUME_CREATE** flag.

HE_ERR_VOLUME_CORRUPT – The volume is corrupt. This volume was, at some point, formatted properly as a Helium volume, but subsequently corrupted.

HE_ERR_VOLUME_IN_USE – The volume is in use, but an **he_open** flag to truncate the volume is specified.

HE_ERR_VOLUME_FULL – The volume is fully utilized. There are no garbage to reclaim in order to accommodate the latest update to the datastore. Delete some data to free up space. A good design principle is to make your volume about 20 percent larger than the eventual data size that is expected to reside on the volume.

HE_ERR_DATASTORE_NOT_FOUND – An attempt to open a datastore failed because such a datastore does not exist. Consider using the **HE_O_CREATE** flag when opening the datastore.

HE_ERR_DATASTORE_EXISTS – An attempt to rename a datastore to a new name that already exists on the volume has failed. Don't rename a datastore using an existing name.

HE_ERR_DATASTORE_IN_USE – An attempt to remove (i.e., permanently delete) a datastore has failed because additional open handles exist. Close all open handles, except for the one held by the thread attempting to remove the datastore.

HE_ERR_ITEM_NOT_FOUND – An operation on an item cannot proceed because the item does not exist in the datastore.

HE_ERR_ITEM_EXISTS – An operation, most likely an insert, has failed because that item already exists in the datastore.

HE_ERR_NETWORK_LISTEN – An attempt to listen to incoming network connections has failed.

HE_ERR_NETWORK_ACCEPT – An attempt to accept an incoming network connection has

failed.

HE_ERR_NETWORK_CONNECT – An attempt to connect to a Helium server has failed.

HE_ERR_NETWORK_READ – An attempt to read from the network has failed.

HE_ERR_NETWORK_WRITE – An attempt to write to the network has failed.

HE_ERR_NETWORK_PROTOCOL – Data received over the network is corrupt or poorly formed with respect to the established communication protocol.

RETURN VALUE

A constant C string is returned by **he_strerror**.

THREADS & REENTRANT CODE

Helium functions are reentrant and specifically designed for use in multi-threaded applications. Applications do not need to serialize calls to Helium, subject to the following two exceptions:

1 - Closing of the datastore handle must take place as the only and last operation on the datastore. Closing of a datastore handle while Helium update, lookup, or transactions on the datastore are in progress will result in undefined behavior.

2 - Committing or discarding transactions must take place as the only and last operation on the transaction. Committing or discarding a transaction while Helium update or lookup on the transaction are in progress will result in undefined behavior.

HELIUM API LISTING

he_is_valid(3), **he_is_transaction**(3), **he_enumerate**(3), **he_open**(3), **he_close**(3), **he_remove**(3), **he_rename**(3), **he_stats**(3), **he_transaction**(3), **he_commit**(3), **he_discard**(3), **he_update**(3), **he_insert**(3), **he_replace**(3), **he_delete**(3), **he_delete_lookup**(3), **he_exists**(3), **he_lookup**(3), **he_next**(3), **he_prev**(3), **he_iterate**(3), **he_version**(3), **he_perror**(3), **he_strerror**(3)