

# Flash-Aware Application APIs

NVMKV API Library  
NON-VOLATILE MEMORY KEY-  
VALUE STORE API  
SPECIFICATION 0.7

THURSDAY, AUGUST 29, 2013



## Legal Notices

---

The information contained in this document is subject to change without notice.

Copyright © 2013 Fusion-io, Inc. All rights reserved.

Fusion-io, the Fusion-io logo, ioMemory, Virtual Storage Layer, ioCache, and ioDrive are registered trademarks, and VSL and ioFX are trademarks of Fusion-io, Inc. in the United States and other countries.

The names of other organizations and products referenced herein are the trademarks or service marks (as applicable) of their respective owners. Unless otherwise stated herein, no association with any other organization or product referenced herein is intended or should be inferred.

Fusion-io  
2855 E. Cottonwood Parkway, Box 100  
Salt Lake City, UT 84121  
USA

(801) 424-5500



# Table of Contents

<b>List of Tables</b> .....	<b>v</b>
<b>List of Figures</b> .....	<b>vi</b>
<b>Introduction</b> .....	<b>1</b>
NVMKV library .....	1
NVMKV Store .....	2
<b>Principle of Operation</b> .....	<b>3</b>
Containers for KV store .....	3
Handling KEY hash collisions .....	3
Handling key expiry .....	3
Iteration .....	4
<b>nvm_kv_open</b> .....	<b>5</b>
<b>nvm_kv_pool_create</b> .....	<b>7</b>
<b>nvm_kv_set_global_expiry</b> .....	<b>8</b>
<b>nvm_kv_put</b> .....	<b>9</b>
<b>nvm_kv_get</b> .....	<b>10</b>
<b>nvm_kv_get_val_len</b> .....	<b>12</b>
<b>nvm_kv_delete</b> .....	<b>13</b>
<b>nvm_kv_exists</b> .....	<b>14</b>
<b>nvm_kv_pool_delete</b> .....	<b>15</b>
<b>nvm_kv_delete_all</b> .....	<b>16</b>
<b>nvm_kv_begin</b> .....	<b>17</b>
<b>nvm_kv_get_current</b> .....	<b>18</b>
<b>nvm_kv_next</b> .....	<b>19</b>
<b>nvm_kv_iteration_end</b> .....	<b>20</b>



nvm_kv_get_store_info .....	21
nvm_kv_get_pool_info .....	22
nvm_kv_get_pool_metadata .....	23
nvm_kv_get_key_info .....	24
nvm_kv_close .....	25
nvm_kv_batch_put .....	26
Appendix A: errno .....	27
Appendix B: Sample Code .....	32



## List of Tables

---

Table 1: <code>nvm_kv_put()</code> with different values of replace flag .....	9
Table 2: <code>nvm_kv</code> Error Codes .....	27



## List of Figures

---

Figure 1: NVMKV Library Overview .....	1
Figure 2: NVMKV Store Overview .....	2



# Introduction

A Key-Value (KV) store is a type of NoSQL database used in high-performance, data-intensive, and scale-out environments. Persistent KV stores today use flash as a block device and are unable to fully leverage powerful capabilities that a Flash Translation Layer (FTL) offers, such as dynamic mapping, transaction persistence, and auto-expiry. Additionally, non-FTL-aware KV stores maintain some of the same metadata that are maintained by the underlying FTL, resulting in wasted memory.

## NVMKV library

NVMKV, described in this API specification, is a lightweight user space library that provides basic Key-Value operations such as get, put, delete, and advanced operations such as batch put/get/delete, pools, iterator, and lookup. The library leverages highly-optimized primitives such as sparse addressing, atomic-writes, Persistent TRIM, etc., provided by the underlying FTL. The strong consistency guarantees of the underlying primitives allow KV to achieve high performance combined with ACID compliance.

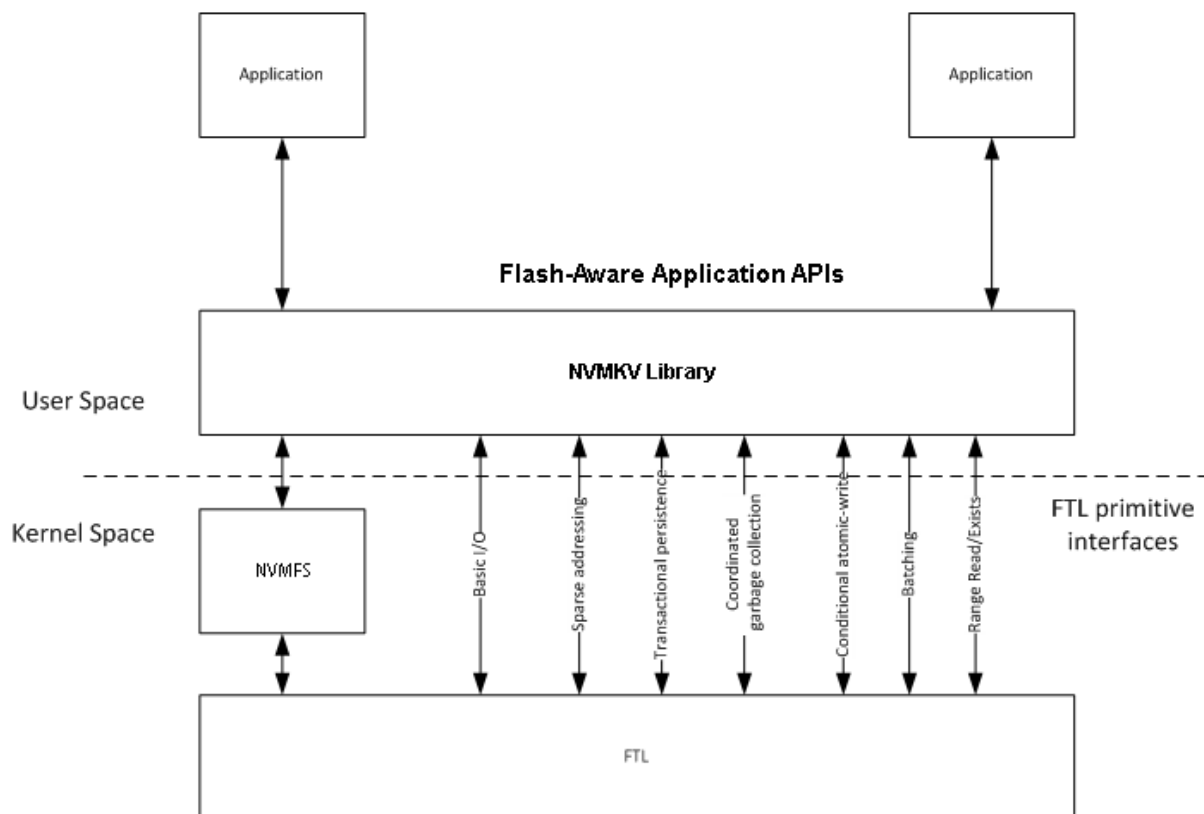


Figure 1: NVMKV Library Overview



## NVMKV Store

The KV store is a persistent store for Key-Value pairs, which reside on containers that could be a sparse storage (aka raw block) device, file system, or any other container exported by the underlying FTL. The KV API library is intended to work with all of these container models, provided that the underlying container supports KV operations. (With v0.7, NVMKV is supported on a sparsely-formatted raw block device. See README for formatting instructions.) Creating and administering underlying containers are performed outside of the KV library. The library expects containers to be accessible via an identifier (such as file descriptor in Linux). Once a container identifier is provided, the library provides an API ([nvm\\_kv\\_open\(\)](#)) to use that container identifier to create/open a KV store. One or more pools may be created within a KV store. A pool allows users to group related keys into buckets, which can be accessed and managed separately within a KV store. Users can create and delete pools, plus perform get/put/delete/iterate operations in a pool. A KV store currently supports up to 1 million pools. Keys are currently limited to 128 bytes and values are limited to 1MiB–1 KiB (1MiB minus 1KiB). The total number of concurrent iterators is limited to 128.

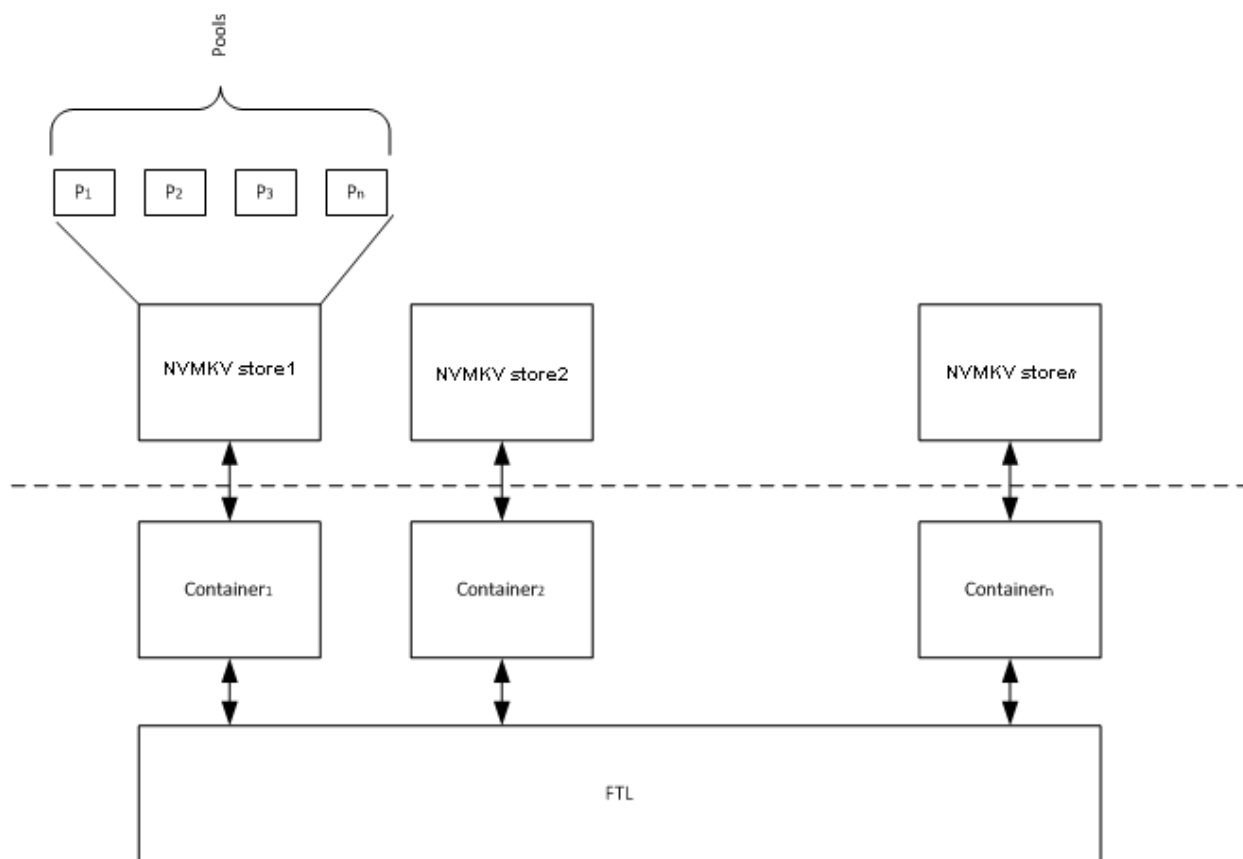


Figure 2: NVMKV Store Overview





# Principle of Operation

KV leverages sparse addressing primitives of an FTL to encode key hashes into sparse virtual addresses. By contrast, for KV stores built without the use of sparse addressing primitives, two layers of translations are needed to map keys to locations in the flash device. One translation layer maps key hashes to Logical Block Addresses (LBAs) in the FTL, while the other translation layer maps LBAs to physical locations in flash. By leveraging an FTL's sparse addressing, a key is hashed directly into an LBA.

The sparse address space is divided into two areas: the bits corresponding to a key (and pool id if created) hash and the bits corresponding to the key's value range. The sparse address space is large enough to ensure that hash collisions are kept at very low levels (<3% for 1TiB device considering KV pairs of smallest size). Collisions are handled within the library by deterministically computing alternate hash locations. Up to eight hash locations are tried before a KV store refuses to accept a new key which produced a hash collision. The probability of such failures is very rare. For a 1 TB device, the probability of such a failure is one per trillion PUT operations for a worst-case scenario where all KV pairs are of 1 sector in size. For the current implementation, the minimum unit of storage for a KV pair is 512B.

For each KV pair, the KV store maintains some persistent metadata, including the actual key, key length, value length, pool id and other necessary information. The metadata is compacted with the key's actual value as appropriate and stored along with actual values. KV provides a zero-copy implementation where data is directly DMA'ed to/from the user buffer provided by the calling function.

## Containers for KV store

Currently, KV supports sparse storage device. To create a KV store on sparse storage device, the user needs to open the device in read-write mode with the O\_DIRECT flag set and pass the returned handle to [nvm\\_kv\\_open\(\)](#). This is described later in this document.

## Handling KEY hash collisions

KV performs an exists operation (in-memory) on the computed hash to determine if that hash has already been used. In case the hash has been used, KV performs a read operation on the media to determine whether the stored key on the media is identical to the key supplied by [nvm\\_kv\\_put\(\)](#), or whether these are just two different keys which produced the same hash. In case the keys are identical, the replace flag in [nvm\\_kv\\_put\(\)](#), as described in the table under [nvm\\_kv\\_put\(\)](#), governs whether the new value replaces the key's previous value.

## Handling key expiry

The KV library auto-deletes expired Key-Value pairs. There are two expiry modes that this library supports:

- a. "Arbitrary expiry mode" – This should be set if each key has different expiry time.
- b. "Global expiry mode" – This should be set if each key expires after a fixed interval since insertion.

In "arbitrary expiry mode", KV periodically scans the media and then removes any expired keys. In the current release, scanning is triggered every 24 hours and if KV store occupies 25% or more capacity, in increments of 5%, whichever happens first assuming completion of previous scanning. As "arbitrary expiry mode" scans the media, it could have some negative performance impact. In contrast, "global expiry mode" is closely integrated with the FTL and doesn't require media scanning. As such, "global expiry mode" is expected to perform better than "arbitrary expiry mode".

Note that keys on media could live longer than the expiry time depending on the next periodic scan; however, it is critical to note that KV guarantees not to return expired keys if requested.



## Iteration

The KV library supports iteration functionality, which can be performed on a specific pool, on all the pools, or on a KV store (in the absence of pools). Before starting iteration, applications need to get an iterator id by calling [`nvm\_kv\_begin\(\)`](#). A call to [`nvm\_kv\_begin\(\)`](#) places the iterator to the next valid key location. [`nvm\_kv\_get\_current\(\)`](#) and [`nvm\_kv\_next\(\)`](#) must be called in that order to get the Key-Value pair and place the iterator to the next valid key location respectively.



## nvm\_kv\_open

### NAME

nvm\_kv\_open – Opens and validates an existing KV store or creates a new KV store if it doesn't exist.

### SYNOPSIS

```
#define NVM_KV_MAX_KEY_SIZE    128           //in bytes
#define NVM_KV_MAX_VALUE_SIZE  1047552      //in bytes. (1MiB - 1KiB) if block size
                                           //is 1K or smaller. Otherwise max value
                                           //size is (1Mib - BlockSize)
                                           //e.g. for 2K blocksize max value size is
                                           //1046528 bytes

#define NVM_KV_MAX_POOLS 1048576(1 Million)

#define NVM_KV_MAX_ITERATORS 128           //max iterators supported per KV store
#define NVM_KV_MAX_VECTORS    16           //max vector size for batch operations

typedef enum
{
    KV_DISABLE_EXPIRY = 0,           //Keys never expire
    KV_ARBITRARY_EXPIRY = 1,         //Each key has arbitrary expiry time
    KV_GLOBAL_EXPIRY = 2             //Each key expires at a certain interval after
insertion
} nvm_kv_expiry_t;

int nvm_kv_open(int id, uint32_t version, uint32_t max_pools, uint32_t
expiry);
```

### DESCRIPTION

The `nvm_kv_open()` API creates a KV store and sets the version number `version`. The underlying device (`/dev/fctx`) needs to be formatted with sparse storage and atomic-write options enabled. (Example: `# fio-format -A -e /dev/fctx`). The version is set at the KV store creation time. Subsequent `nvm_kv_open()` calls validate the version and return an error if there is a mismatch. A KV Store can be created on a sparse storage device. To create a KV store on a sparse storage device, the device needs to be opened with the `O_DIRECT` flag prior to calling `nvm_kv_open()`, and the returned descriptor should be passed as `id` to `nvm_kv_open()`.



The initial call to `nvm_kv_open()` configures the maximum number of pools by setting the required `max_pools` parameter. Once set, it cannot be changed without destroying the KV store. `max_pools` is ignored in subsequent `nvm_kv_open()` calls. If the KV store already exists, then `nvm_kv_open()` validates the KV store. A KV store currently supports up to 1048576 pools. This limit excludes the default pool (with a pool id 0) created by `nvm_kv_open()`. Applications that do not require pool support can set `max_pools` to zero. A KV store can have one or more pools and each pool can have one or more keys.

The parameter `expiry` sets the key expiry mode for the entire KV store. The caller must set `expiry` to `KV_DISABLE_EXPIRY` if the keys inserted in KV store are never to expire. The caller must set `expiry` to `KV_ARBITRARY_EXPIRY` if every key has different expiry time. The caller must set `expiry` to `KV_GLOBAL_EXPIRY` if every key expires at a certain interval from the time of insertion. By default keys will not expire if global expiry time is not set for a global expiry mode. [nvm\\_kv\\_set\\_global\\_expiry\(\)](#) can be used to set global expiry time. Setting `expiry` to an unsupported value is considered an invalid input.

## RETURN VALUE

Upon successful completion, the `nvm_kv_open()` call returns the KV store id, which is a non-zero positive number. In case of an error, -1 is returned and `errno` is set to indicate the error. The return KV store id is based on the default pool (with pool id 0).

## ERRORS

Refer to [Appendix A](#) for the `errno` set by the KV store.



## nvm\_kv\_pool\_create

---

### NAME

nvm\_kv\_pool\_create – Creates a pool in a given KV store.

### SYNOPSIS

```
typedef struct
{
    uint8_t pool_tag[16];          //< 16 bytes pool tag
} nvm_kv_pool_tag_t;

int nvm_kv_pool_create(int kv_id, nvm_kv_pool_tag_t *tag);
```

### DESCRIPTION

nvm\_kv\_pool\_create() creates a pool within a KV store and returns the pool id. If you provide a pool\_tag that already exists in the system, a new pool will not be created; rather, the existing pool's id will be returned. The input parameter kv\_id is the id returned by [nvm\\_kv\\_open\(\)](#).

The API also associates the user-specified tag, if non-null, with pool id. tag is null terminated string with maximum length of 16 bytes. The KV store persists this mapping. The creation of pools does not consume any extra capacity; however, a slight performance impact is observed when the number of pools created exceeds 2048. If the pool with specified tag already exists, then this API returns the associated pool id without creating a new pool.

### RETURN VALUE

Upon successful completion, nvm\_kv\_pool\_create() returns an id (aka pool id) which is a non-zero positive number starting with 1. In case of an error, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_set\_global\_expiry

---

### NAME

nvm\_kv\_set\_global\_expiry – Sets the global KV pair expiry time.

### SYNOPSIS

```
int nvm_kv_set_global_expiry(int kv_id, uint32_t expiry);
```

### DESCRIPTION

nvm\_kv\_set\_global\_expiry() sets the global expiry time if *expiry* in [nvm\\_kv\\_open\(\)](#) is set to KV\_GLOBAL\_EXPIRY. The global expiry time sets every key to expire at *expiry* seconds from the time of key insertion. For other *expiry* modes, this API does not have any impact. The unit of *expiry* is in seconds. The new *expiry* time will also be applicable for all the existing as well as new keys in a KV store.

### RETURN VALUE

Upon successful completion, nvm\_kv\_set\_global\_expiry() returns a zero. In case of an error, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_put

### NAME

nvm\_kv\_put – Inserts a key and its associated value in the KV store.

### SYNOPSIS

```
typedef uint8_t nvm_kv_key_t;

int nvm_kv_put(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t key_len,
void *value, uint32_t value_len, uint32_t expiry, bool replace, uint32_t gen_
count);
```

### DESCRIPTION

nvm\_kv\_put() sets key to the specified value. This either updates an existing key, or adds a new Key-Value pair if the key does not currently exist. The size of the value is in bytes and is specified by value\_len. The user buffer specified by value must be sector-aligned. expiry, in seconds relative to the insertion time, specifies the key expiration. The expiry parameter is ignored if expiry in nvm\_kv\_open is set to a value other than KV\_ARBITRARY\_EXPIRY. gen\_count indicates the generation count for the key. This is transparent to the KV store. replace, if set to TRUE, replaces the value of the specified key as described in the below table. The maximum value\_len supported is 1MiB -1KiB (1 MiB less 1 KiB). The 1KiB is reserved for internal use. key\_len can vary between 1 bytes and 128 bytes. pool\_id, if set to 0, inserts the key into default pool which is created implicitly at the successful completion of [nvm\\_kv\\_open\(\)](#). Note that value must be sector-aligned.

**Table 1: nvm\_kv\_put() with different values of replace flag**

Key already exists	Replace flag	Outcome
Yes	True	Replace the value of the key.
Yes	False	Return an error.
No	True	Write the value of the key.
No	False	Write the value of the key.

### RETURN VALUE

Upon successful completion, nvm\_kv\_put() returns the number of bytes written; otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get

### NAME

nvm\_kv\_get – Retrieves the value associated with a given key.

### SYNOPSIS

```
typedef struct
{
    uint32_t pool_id;
    uint32_t key_len;
    uint32_t value_len;
    uint32_t expiry;        //Set to 0 if keys never expiry otherwise set to actual
                             expiry
    uint32_t gen_count;
    uint32_t reserved1;
}nvm_kv_key_info_t;

int nvm_kv_get(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t key_len,
void *value, uint32_t value_len, bool read_exact, nvm_kv_key_info_t *key_
info);
```

### DESCRIPTION

nvm\_kv\_get() retrieves the value associated with a key in a given pool; otherwise, it returns -1 and sets value to NULL.

If the length of the value provided (as indicated by value\_len) is less than the length of the actual value, then the reduced number of bytes requested is returned. If the length of the value (value\_len) provided exceeds the length of the actual value, then nvm\_kv\_get() returns the actual length (as reflected in return value of the API or in key\_info data structure); however, the user buffer (value) could have in-deterministic data out beyond the actual value\_len which should be ignored. Note that value must be sector-aligned and allocated in multiples of sector size. value\_len needs to be a multiple of sector size. The caller needs to set the read\_exact flag to TRUE if value\_len depicts the expected value size. In case the caller has pre-allocated buffer with largest value\_len and doesn't have info about expected value\_len, this flag should be set to FALSE.

If key\_info is NULL, then errno is set and the operation fails; otherwise, it is filled with the current info for this key.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get() returns the number of bytes of data returned in the user-specified buffer and the information about the key; otherwise, -1 is returned, and errno is set to indicate the error. In case the length of





the data requested is less than the length of the actual value, the number of bytes requested is returned. In any case, the value\_len field of key\_info indicates the actual value length.

## **ERRORS**

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_val\_len

---

### NAME

nvm\_kv\_get\_val\_len – Return the value length associated with the key without any I/O.

### SYNOPSIS

```
int nvm_kv_get_val_len(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t
key_len);
```

### DESCRIPTION

nvm\_kv\_get\_val\_len() returns the value length, rounded up to the next sector, associated with a key in a given pool without any I/O to underlying device. In case the key specified has a hash collision with an existing key, the API returns the maximum of value length associated with any collided key. In some cases, this API can return value length which is a sector more than the actual value length associated with the given key.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_val\_len() returns the value length; otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_delete

---

### NAME

nvm\_kv\_delete – Deletes a key from a given pool.

### SYNOPSIS

```
int nvm_kv_delete(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t key_
len);
```

### DESCRIPTION

nvm\_kv\_delete() deletes key and the associated value from the pool pool\_id.

### RETURN VALUE

Upon successful completion, nvm\_kv\_delete() returns zero; otherwise, -1 is returned and errno is set to indicate the error. The key, if not found or invalid, is considered a success and is reflected in the return value, which is zero.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_exists

---

### NAME

nvm\_kv\_exists – Determines whether a key exists in a given pool.

### SYNOPSIS

```
int nvm_kv_exists(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t key_len,
nvm_kv_key_info_t *key_info);
```

### DESCRIPTION

nvm\_kv\_exists() determines whether the key exists in the pool pool\_id.

### RETURN VALUE

nvm\_kv\_exists() returns 1 if the key exists; 0 if the key does not exist. In case of an error, -1 is returned and errno is set to indicate the error. key\_info, if non-null, is populated in case the key exists.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_pool\_delete

---

### NAME

nvm\_kv\_pool\_delete – Deletes all Key-Value pairs from a given pool and then deletes the specified pool.

### SYNOPSIS

```
int nvm_kv_pool_delete(int kv_id, int pool_id);
```

### DESCRIPTION

nvm\_kv\_pool\_delete() deletes all Key-Value pairs from pool pool\_id, and then deletes the specified pool. Pool deletion is an asynchronous background operation; however, once pool deletion starts, it disallows any key insertions in the specified pool. If you pass in a pool\_id with value of -1, all user-created pools with a pool\_id greater than or equal to 1 will be deleted from the KV store.

Zero is not a valid value for the pool\_id for this API and will result in failure with errno set to invalid input parameter. Note that until pool deletion is completed on the flash media, the total pool count is not decremented. The deletion of a pool deletes all the Key-Value pairs in that pool.

### RETURN VALUE

Upon successful completion, nvm\_kv\_pool\_delete() returns zero; otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_delete\_all

---

### NAME

nvm\_kv\_delete\_all – Deletes all Key-Value pairs from a KV store in all pools, including the default pool.

### SYNOPSIS

```
int nvm_kv_delete_all(int kv_id);
```

### DESCRIPTION

nvm\_kv\_delete\_all() deletes all Key-Value pairs from all pools (including Key-Value pairs from the default pool) within the KV store, represented by kv\_id. Note that [nvm\\_kv\\_pool\\_delete\(\)](#) should be used to delete all user-defined pools and leave the KV store's default pool intact.

### RETURN VALUE

Upon successful completion, nvm\_kv\_delete\_all() returns zero; otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_begin

### NAME

nvm\_kv\_begin – Sets the iterator to the beginning of a given pool.

### SYNOPSIS

```
int nvm_kv_begin(int kv_id, int pool_id);
```

### DESCRIPTION

nvm\_kv\_begin() sets the iterator to the beginning of the KV store kv\_id and returns an iterator id. pool\_id, if set to -1, allows iteration over all pools in a KV store. pool\_id, if set to 0, allows iteration over the default pool. The maximum number of iterators supported by a KV store at any given time is 128. The way to iterate over all Key-Value pairs is to invoke nvm\_kv\_begin(), followed by [nvm\\_kv\\_get\\_current\(\)](#), followed by [nvm\\_kv\\_next](#). nvm\_kv\_begin() sets up the iterator to the next valid key location for the requested pool. For an empty pool, nvm\_kv\_begin returns -1 and errno is set to indicate the error.

### RETURN VALUE

Upon successful completion, nvm\_kv\_begin() returns an iterator id (note that zero is a valid iterator id); otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_current

### NAME

nvm\_kv\_get\_current – Retrieves the Key-Value pair at the current iterator location in a given pool.

### SYNOPSIS

```
int nvm_kv_get_current(int kv_id, int iterator_id, nvm_kv_key_t *key, uint32_t
*key_len, void *value, uint32_t value_len, nvm_kv_key_info_t *key_info);
```

### DESCRIPTION

nvm\_kv\_get\_current() retrieves key and value from the current iterator location, specified by iterator\_id, from the store/pool specified by kv\_id. The actual key size is returned in key\_len and the actual key is returned in key. Note that the caller must allocate the maximum size of a key for key\_len before calling nvm\_kv\_get\_current().

If the length of the buffer provided (as indicated by value\_len) is less than the length of the actual value, then the reduced number of bytes requested is returned. If the length of the value (value\_len) provided exceeds the length of the actual value, then nvm\_kv\_get\_current() returns the actual length; however, the user buffer (value) is zeroed out beyond the actual value\_len. Note that value must be sector-aligned and allocated in a multiple of sector size. value\_len does not necessarily need to be a multiple of sector size.

If key\_info is NULL, then errno is set and the operation fails; otherwise, it is filled with the current info for this key.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_current() returns the number of bytes of data returned in the user-specified buffer (value\_len) and the information about the key; otherwise, -1 is returned, and errno is set to indicate the error. The API returns -1 in case the iterator id is invalid. In case the length of data requested is less than the length of the actual value, the number of bytes requested is returned. In any case, the value\_len field of key\_info indicates the actual value length.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.





## nvm\_kv\_next

### NAME


nvm\_kv\_next – Sets the iterator to the next key location in a given pool.

### SYNOPSIS

```
int nvm_kv_next(int kv_id, int iterator_id);
```

### DESCRIPTION

nvm\_kv\_next() sets the iterator specified by iterator\_id to the next key for a given iterator.

 Note: An iterator running in parallel to active [nvm\\_kv\\_put\(\)](#) operations may miss recently inserted items.

### RETURN VALUE

Upon successful completion, nvm\_kv\_next() returns zero; otherwise, -1 is returned and errno is set to indicate the error. The API returns 1 (positive 1) if the end of the KV store (or pool) is reached. The errno is set to -85.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_iteration\_end

---

### NAME

nvm\_kv\_iteration\_end – Ends an iteration.

### SYNOPSIS

```
int nvm_kv_iteration_end(int kv_id, int iterator_id);
```

### DESCRIPTION

nvm\_kv\_iteration\_end() ends an iteration and releases the iterator id in a free pool, which can be used for future iterations.

### RETURN VALUE

Upon successful completion, nvm\_kv\_iterator\_end() returns zero; otherwise, -1 is returned and errno is set to indicate the error.

No error will be returned if an uninitialized `iterator_id` is passed to the API.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_store\_info

---

### NAME

nvm\_kv\_get\_store\_info – Returns metadata information about a KV store.

### SYNOPSIS

```
typedef struct
{
    uint32_t version;    //KV store version set using nvm\_kv\_open\(\)
    uint32_t num_pools;  //Total number of pools
    uint32_t max_pools;  //Maximum number of pools supported
    uint32_t expiry_mode;
    uint64_t num_keys;   //Total number of valid keys across all the pools in a
    given KV store
    uint64_t free_space; //In bytes
} nvm_kv_store_info_t;

int nvm_kv_get_store_info(int kv_id, nvm_kv_store_info_t *store_info);
```

### DESCRIPTION

nvm\_kv\_get\_store\_info() returns metadata information about the KV store specified by kv\_id.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_store\_info() returns zero and the structure store\_info is populated with correct values; otherwise, -1 is returned, and errno is set to indicate the error. This API is blocking API and could take several minutes before it returns. num\_keys returned by this API may not be precise and could have 10% margin of error depending on concurrent put/delete operations.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_pool\_info

### NAME

nvm\_kv\_get\_pool\_info – Returns metadata information about a given pool in a KV store.

### SYNOPSIS

```
typedef enum
{
    POOL_NOT_IN_USE = 0,           //Pool is not in use
    POOL_IN_USE = 1,               //Pool is in use
    POOL_DELETION_IN_PROGRESS = 2, //Pool is currently being deleted
    POOL_IS_INVALID = 3           //Pool is invalid
} nvm_kv_pool_status_t;

typedef struct
{
    uint32_t version;              //Version of the pools, currently set to 0
    uint32_t pool_status ;
} nvm_kv_pool_info_t;

int nvm_kv_get_pool_info(int kv_id, int pool_id, nvm_kv_pool_info_t *pool_info);
```

### DESCRIPTION

nvm\_kv\_get\_pool\_info() returns information about the specified pool. Applications can poll nvm\_kv\_get\_pool\_info() to get the status of a pool deletion operation.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_pool\_info() returns zero and the structure pool\_info is populated with correct values; otherwise, -1 is returned, and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_pool\_metadata

---

### NAME

nvm\_kv\_get\_pool\_metadata – Returns pool id and associated tag iteratively for all the pools in a KV store.

### SYNOPSIS

```
typedef struct
{
    int pool_id ;

    nvm_kv_pool_tag_t tag ;          //fixed size of 16 bytes
} nvm_kv_pool_metadata_t;

int nvm_kv_get_pool_metadata(int kv_id, nvm_kv_pool_metadata_t *pool_md,
uint32_t count, uint32_t start_count);
```

### DESCRIPTION

nvm\_kv\_get\_pool\_metadata() returns pool id and associated tag iteratively for all the pools in KV store. The caller allocates a contiguous memory of count elements of type nvm\_kv\_pool\_metadata\_t. At the start of iteration, the caller sets the start\_count to 1. Successful completion of this API call returns the total count returned which should be used as start\_count in subsequent call to nvm\_kv\_get\_pool\_metadata(). The return value, less than the count, indicates the completion of iteration.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_pool\_metadata() returns the total count of pool id and associated tag returned and the information is populated in structure pool\_md. Upon completion of iteration, nvm\_kv\_get\_pool\_metadata() returns the count other than the one specified as input. Upon failure, -1 is returned, and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_get\_key\_info

---

### NAME

nvm\_kv\_get\_key\_info – Returns the metadata associated with a given key in a given pool.

### SYNOPSIS

```
int nvm_kv_get_key_info(int kv_id, int pool_id, nvm_kv_key_t *key, uint32_t
key_len, nvm_kv_key_info_t *key_info);
```

### DESCRIPTION

The nvm\_kv\_get\_key\_info() API returns metadata information associated with a given key in a given pool.

### RETURN VALUE

Upon successful completion, nvm\_kv\_get\_key\_info() returns zero and the structure key\_info is populated with correct values if the key exists; otherwise, -1 is returned, and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_close

---

### NAME

nvm\_kv\_close – Closes a KV store.

### SYNOPSIS

```
int nvm_kv_close(int kv_id);
```

### DESCRIPTION

nvm\_kv\_close() closes a KV store. [nvm\\_kv\\_open\(\)](#) must be called to re-open the KV store to perform any KV operations.

### RETURN VALUE

Upon successful completion, nvm\_kv\_close() returns zero; otherwise, -1 is returned and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.



## nvm\_kv\_batch\_put

### NAME

nvm\_kv\_batch\_put – Sets the values for a batch of specified keys.

### SYNOPSIS

```
typedef struct
{
    uint32_t    key_len;
    uint32_t    value_len;
    uint32_t    expiry;
    uint32_t    gen_count;
    uint32_t    replace;
    uint32_t    reserved1;
    nvm_kv_key_t *key;
    void        *value;
} nvm_kv_iovec_t;

int nvm_kv_batch_put(int kv_id, int pool_id, nvm_kv_iovec_t *kv_iov, uint32_t
iov_count);
```

### DESCRIPTION

nvm\_kv\_batch\_put() operates similarly to [nvm\\_kv\\_put\(\)](#), but supports a batch of keys. It utilizes an array of nvm\_kv\_iovec\_t structures referenced by kv\_iov to hold the Key-Value pairs to be written. iov\_count is used to designate the number of Key-Value pairs in array kv\_iov. Note that each value must be sector-aligned and a multiple of sector size. The total number of KV pairs supported in a batch request is limited to 64.

### RETURN VALUE

Upon successful completion, nvm\_kv\_batch\_put() returns 0; otherwise, -1 is returned, and errno is set to indicate the error.

### ERRORS

Refer to [Appendix A](#) for the errno set by the KV store.





## Appendix A: errno

---

The following table describes the API, the errno, and the corresponding descriptions for each API.

**Table 2: nvm\_kv Error Codes**



API	Error Code	Description
nvm_kv_open	-11	Internal library errors
	-1	Invalid descriptor passed
	-6	Memory allocation failed
	-3	KV store verification failed
	-35	I/O error
	-36	Invalid input parameters. Reasons can include the following: - max_pool is greater than the supported maximum pools. - expiry_mode is set to an unsupported value.
	-17	Underlying NVM device is not capable of supporting KV.
	NA	In case of success, returns KV store ID which is > 0
nvm_kv_pool_create	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Maximum pools limit has been reached
	-11	Internal library errors
	-6	Memory allocation failed
	-35	I/O error
nvm_kv_pool_delete	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id
	-6	Memory allocation failed
	-25	Operation not supported
nvm_kv_pool_info	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id
nvm_kv_get_pool_metadata	-36	Invalid input parameters. Reasons can include the following: - Invalid id - pool_md is NULL
	-11	Fetching pool tag failed.
nvm_kv_put	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL - value_len - Either 0 or set to more than the max supported size (1MiB - 1KiB)



API	Error Code	Description
		- value is NULL - value is not sector-aligned
	-6	Memory allocation failed
	-11	Internal library errors
	-26	Key is already in the store and the replace flag is not set
	-35	I/O error
	NA	In case of success, return # of bytes of user-data written
nvm_kv_get	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL - value_len - Either 0 or set to more than the max supported size (1MiB - 1KiB) or not multiple of sector size - value is NULL - value is not sector-aligned - key_info is null
	-6	Memory allocation failed
	-85	Key not found in the KV store
	-35	I/O error
	NA	In case of success, return # of bytes of user-data read and stored in the input buffer
nvm_kv_get_val_len	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL
	-6	Memory allocation failed
	-11	Internal library errors
	-85	Key not found in the KV store
nvm_kv_delete	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL



API	Error Code	Description
	-6	Memory allocation failed
	-35	I/O error
	0	Success
nvm_kv_exist	-11	Internal library errors
	-6	Memory allocation failed
	-35	I/O error
	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL
nvm_kv_begin	-6	Memory allocation failed
	-11	Internal library errors
	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid pool id
	-85	Empty pool without any keys.
	-87	Maximum iterator limit reached.
nvm_kv_next	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid iterator id
	-11	Internal library errors
	-85	End of iteration – no more KV pairs
	-36	Invalid input parameters. Reasons can include the following: - Invalid id - Invalid iterator id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - key is NULL - value_len - Either 0 or set to more than the max supported size (1MiB - 1KiB) - value is NULL
nvm_kv_get_current	-6	Memory allocation failed
	-35	I/O error
nvm_kv_delete_all	-36	Invalid input parameters. Reasons can include the following: - Invalid id



API	Error Code	Description
	-35	I/O error
nvm_kv_close	-36	Invalid input parameters. Reasons can be one of the following: - Invalid id
nvm_kv_get_key_info	-36	Invalid input parameters. Reasons can be one of the following: - Invalid id - Invalid key_len - key_len is either 0 or more than the max supported size (128B) - Invalid pool id - key_info is NULL
	-6	Memory allocation failed
	-85	Key not found (doesn't exist)
	-35	I/O error
nvm_kv_get_store_info	-36	Invalid input parameters. Reasons can be one of the following: - Invalid Id - store_info is NULL
	-11	Internal library errors
nvm_kv_set_global_expiry	-36	Invalid input parameters. Reasons can be one of the following: - Invalid id
	-25	expiry mode is set to non-global expiry during <a href="#">nvm_kv_open()</a> . This API doesn't have any impact if KV store expiry mode is non-global.
nvm_kv_iteration_end	-36	Invalid input parameters. Reasons can be one of the following: - Invalid id - Invalid iterator id  Note: No error will be returned if an uninitialized iterator_id is passed to the API.
nvm_kv_get_pool_info	-36	Invalid input parameters  Reasons can be one of the following: - Invalid id - Invalid pool id - pool_info is NULL.



## Appendix B: Sample Code

---

Sample code for KV is now available as part of the KV package.