

Samsung[®] *Key-value* SSD API

Specification

SAMSUNG ELECTRONICS RESERVES THE RIGHT TO CHANGE PRODUCTS, INFORMATION AND SPECIFICATIONS WITHOUT NOTICE.

Products and specifications discussed herein are for reference purposes only. All information discussed herein is provided on an "AS IS" basis, without warranties of any kind.

This document and all information discussed herein remain the sole and exclusive property of Samsung Electronics. No license of any patent, copyright, mask work, trademark or any other intellectual property right is granted by one party to the other party under this document, by implication, estoppel or otherwise.

Samsung products are not intended for use in life support, critical care, medical, safety equipment, or similar applications where product failure could result in loss of life or personal or physical harm, or any military or defense application, or any governmental procurement to which special terms or provisions may apply.

For updates or additional information about Samsung products, contact your nearest Samsung office.

All brand names, trademarks and registered trademarks belong to their respective owners.

©2018 Samsung Electronics Co., Ltd. All rights reserved.

Revision History

Revision No.	History	Draft Date	Remark
Version 0.5.1	Change device/iterator/container open calls API to pass structure pointer as parameter, and error code as return value	08/22/2018	
Version 0.5.0	Samsung Key-value SSD API spec first draft	08/10/2018	

Contents

1	Device Support Information.....	5
1.1	Supported Devices.....	5
2	Terminology.....	6
2.1	Acronyms and Definitions.....	6
2.2	Feature option.....	6
3	API version.....	7
4	Introduction.....	8
4.1	Scope.....	9
4.2	Assumption.....	9
5	Key-value Entities.....	10
5.1	Device.....	10
5.2	Container.....	10
5.3	Group.....	10
5.4	Tuple.....	10
6	Constants & Data Structures.....	12
6.1	Constants.....	12
6.1.1	KVS_ALIGNMENT_UNIT.....	12
6.1.2	KVS_MAX_KEY_LENGTH.....	12
6.2	Enum Constants.....	13
6.2.1	kvs_delete_option.....	13
6.2.2	kvs_retrieve_option.....	13
6.2.3	kvs_store_option.....	13
6.2.4	kvs_iterator_option.....	14
6.2.5	kvs_container_option.....	14
6.2.6	kvs_result.....	14
6.3	Data Structures.....	17
6.3.1	kvs_device_handle.....	17
6.3.2	kvs_container_handle.....	17
6.3.3	kvs_group_condition.....	17
6.3.4	kvs_iterator_handle.....	17
6.3.5	kvs_iterator_list.....	18
6.3.6	kvs_device.....	19
6.3.7	kvs_container.....	19
6.3.8	kvs_container_name.....	20
6.3.9	kvs_key.....	20
6.3.10	kvs_value.....	20
6.3.11	kvs_tuple_info.....	21
6.3.12	kvs_iterator_context.....	21
6.3.13	kvs_delete_context.....	21
6.3.14	kvs_store_context.....	21
6.3.15	kvs_retrieve_context.....	22
6.3.16	kvs_container_context.....	22

7	Key Value SSD APIs.....	23
7.1	Device-level APIs.....	23
7.1.1	kvs_open_device.....	23
7.1.2	kvs_close_device.....	24
7.1.3	kvs_get_device_info.....	25
7.1.4	kvs_get_device_capacity.....	26
7.1.5	kvs_get_device_utilization.....	27
7.1.6	kvs_get_min_key_length.....	28
7.1.7	kvs_get_max_key_length.....	29
7.1.8	kvs_get_min_value_length.....	30
7.1.9	kvs_get_max_value_length.....	31
7.1.10	kvs_get_optimal_value_length.....	32
7.1.11	kvs_create_container.....	33
7.1.12	kvs_delete_container.....	34
7.1.13	kvs_list_containers.....	35
7.2	Container-level APIs.....	36
7.2.1	kvs_open_container.....	36
7.2.2	kvs_close_container.....	37
7.2.3	kvs_get_container_info.....	38
7.3	Key-value tuple APIs.....	39
7.3.1	kvs_get_tuple_info.....	39
7.3.2	kvs_retrieve_tuple.....	40
7.3.3	kvs_store_tuple.....	42
7.3.4	kvs_delete_tuple.....	44
7.3.5	kvs_exist_tuples.....	45
7.4	Iterator APIs.....	46
7.4.1	kvs_open_iterator.....	46
7.4.2	kvs_close_iterator.....	47
7.4.3	kvs_iterator_next.....	48

1 DEVICE SUPPORT INFORMATION

This document describes a Samsung® *Key-value SSD (KVS)* Application Program Interface (API) library.

1.1 Supported Devices

API Version	Supported Device	NVMe Interface(s)
Key-value SSD API v0.5.1	PM983	NVMe 1.2

2.1 Acronyms and Definitions

2.1 Acronyms and Definitions

[illegible]

2.2 Feature option

[DEFAULT]: a default value or selection if not specified explicitly

[OPTION]: a feature marked as OPTION is optional and vendor-specific

[SAMSUNG]: an optional feature that Samsung key-value SSDs support

3 API VERSION

The tables shows the API implementation status.

API	version	comment
Kvs_open_device	V0.5	
Kvs_close_device	V0.5	
Kvs_get_device	V0.5	RetrieveKV SSD device information
Kvs_get_device_capacity	V0.5	
Kvs_get_device_utilization	V0.5	
Kvs_get_min_key_length	TBD	
Kvs_get_max_key_length	TBD	
Kvs_get_min_value_length	TBD	
Kvs_get_max_value_length	TBD	
Kvs_get_optimal_value_length	TBD	
Kvs_create_container	V0.5	
Kvs_delete_container	V0.5	
Kvs_list_container	TBD	
Kvs_open_container	V0.5	
Kvs_close_container	V0.5	
Kvs_get_container_info	TBD	
Kvs_get_tuple_info	TBD	
Kvs_retrieve_tuple	V0.5	
Kvs_store_tuple	V0.5	
Kvs_delete_tuple	V0.5	
Kvs_exist_tuple	TBD	
Kvs_open_iterator	V0.5	
Kvs_close_iterator	V0.5	
Kvs_iterator_next	V0.5	

4 INTRODUCTION

This document describes a device-level, key-value SSD (KVS) Application Program Interface (API) for new SSD storage devices with native key-value interfaces.

The library routines this document defines allow applications to create and use objects, called in tuple, in KV SSDs while permitting portability. The library:

- Extends the C++ language with host and device APIs
- Provides support for container, atomic operation, asynchronous operation, and callback

Library routines and environment variables provide the functionality to control the behavior of KVS. Figure 1 shows the hierarchical KVS architecture.

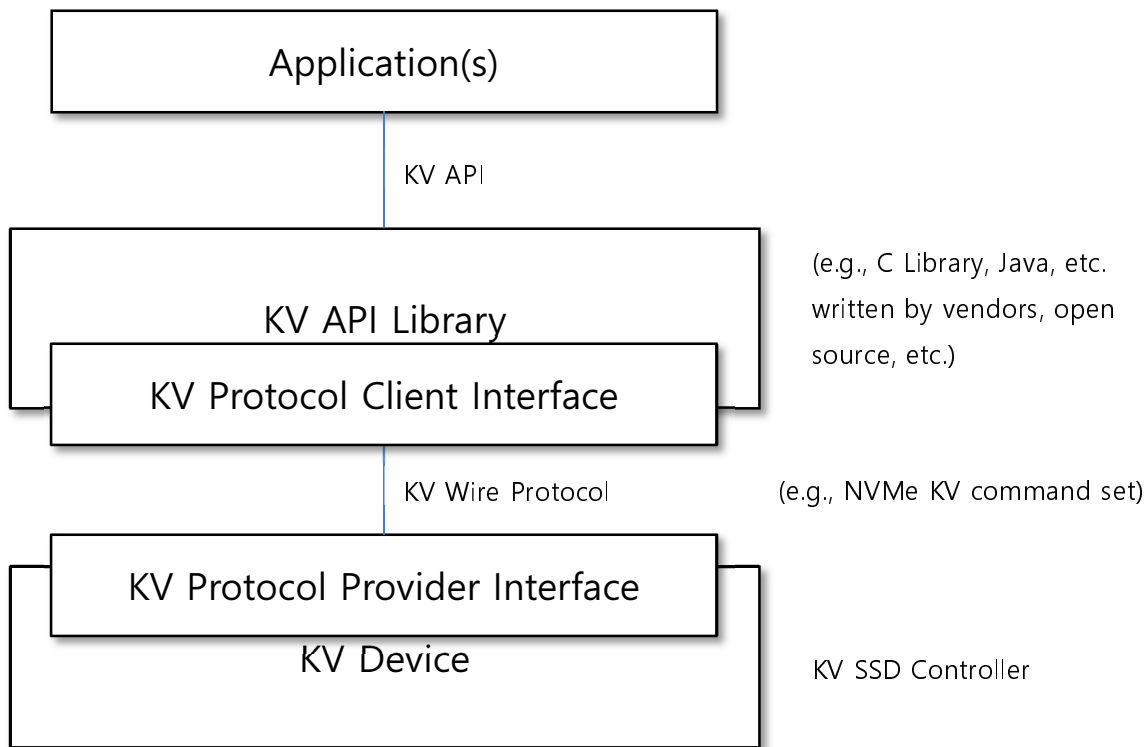


Figure 1. Key-value Architecture

(WARNING) This document is being updated. Until finalized, the API syntax and semantics can [change without notice](#).

4.1 Scope

This key-value SSD API specification only covers APIs and their semantics. It does not discuss specific protocols such as ATA, SCSI, and NVMe, and the API's internal device implementation. For more NVMe command protocol information, please refer to NVMe Key-value command spec.

4.2 Assumption

These device-level APIs have several assumptions:

1. Users of this API conduct device memory management. Any input and output buffers of APIs must be allocated before calling the routines. No memory the library allocates is accessible by user programming.
2. Both host and device use *little endian* memory and transport format. If a host uses big endian byte ordering (e.g., POWER architecture), the host needs to convert it to a little endian format.

5 KEY-VALUE ENTITIES

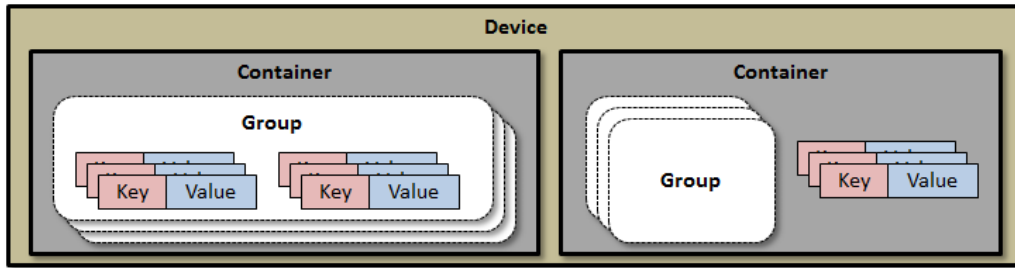


Figure 1. Key-value Objects

5.1 Device

A key-value device is a storage device such as a HDD or SSD which has native storage command protocol of key-value interface. Form factors (2.25", 2.5", M.2, M.3, HHHL, etc.) or command protocols (SATA, SCSI, NVMe, NVMeoF, etc.) are beyond the scope of this specification.

5.2 Container

A *container* is a type of logical unit which provides similar management functionalities as an NVMe namespace, SCSI LUN, or disk partition. A container can store a VM, a database, a file system, etc. A device can simultaneously have multiple containers. A key-value device must support at least one container.

[SAMSUNG] The current implementation supports only one container and container APIs will be added in the future.

5.3 Group

[OPTION] A *group* is a logical set of objects within a container which users can dynamically create. This can be used to represent a shard, a document collection, an iterator, etc. A container can simultaneously have multiple groups.

[SAMSUNG] The current implementation supports only iterator for a key group.

5.4 Tuple

A *tuple* is an object consisting of a *key* and a *value*. It is a unit of access. A key is user-defined and unique within a container. A key length can be fixed or variable but its maximum length is limited. A value length is variable and its maximum is limited as well.

6 CONSTANTS & DATA STRUCTURES

This section defines Key-value SSD core constants, data structures, and functions.

6.1 Constants

6.1.1 KVS_ALIGNMENT_UNIT

This is an alignment unit. An offset of *value* must be a multiple of this value.

[SAMSUNG] The default alignment unit for the Samsung key-value SSD is 32 bytes.

6.1.2 KVS_MAX_KEY_LENGTH

The maximum key length that KVS can support. The default value is 255. This is set when a device is initialized. For example, the Identify Namespace Data Structure in the NVMe spec can be used to report the maximum key length that a device can supports.

[SAMSUNG] The current implementation supports only up to 16B keys and larger key will be supported in the future.

6.2 Enum Constants

6.2.1 kvs_delete_option

```
enum kvs_delete_option {
    KVS_DELETE_TUPLE,      = 0,          // [DEFAULT] delete a tuple if a key exists
}
```

A user can specify a *delete* operation option.

- **KVS_DELETE_TUPLE** defines an idempotent and atomic operation that removes a single tuple matching a given key. This is the default delete operation behavior.

6.2.2 kvs_retrieve_option

```
enum kvs_retrieve_option {
    KVS_RETRIEVE_IDEMPOTENT = 0,          // [DEFAULT] default retrieve option, read the value of tuple
    KVS_RETRIEVE_DELETE     = 1          // [OPTION] read the value of tuple and delete the tuple
}
```

A user can specify a *retrieve* operation option.

- **KVS_RETRIEVE_IDEMPOTENT** specifies that a retrieve operation is idempotent and a user can repeatedly read the tuple value safely. This is the default retrieve operation behavior.
- **KVS_RETRIEVE_DELETE** specifies that an operation reads the tuple value but the tuple is atomically deleted after completing the read. This operation is not idempotent.

[SAMSUNG] The Samsung Key-value SSD supports the KVS_RETRIEVE_IDEMPOTENT option only.

6.2.3 kvs_store_option

```
enum kvs_store_option {
    KVS_STORE_POST      = 0,          // [DEFAULT] non-idempotent store operation; insert or overwrite
    KVS_STORE_IDEMPOTENT = 1,          // insert a new tuple, returns an error if a tuple with the same key exists
    KVS_STORE_APPEND     = 2          // [OPTION] non-idempotent store operation; insert or append value to the existing tuple
}
```

A user can define a *store* operation option.

- **KVS_STORE_POST** defines that a store operation is atomic but nonidempotent. Regardless of existence of a key, all store operations will be executed atomically and the value of tuple will be determined by the order of successful operations. If a key does not exist, a new tuple is stored. If a key exists, the tuple is replaced with a new value.

- **KVS_STORE_IDEMPOTENT** specifies that a store operation is idempotent and a user can only write the tuple value once. If the key does not exist, this operation succeeds and stores a new tuple. Otherwise, this operation fails without affecting the previously-stored tuple. This is the default store operation behavior.
- **KVS_STORE_APPEND** allows a store operation to append a new value to the end of the existing value atomically. If a key does not exist, a new tuple is stored. This operation is not idempotent.

[SAMSUNG] The Samsung Key-value SSD supports KVS_STORE_POST and KVS_STORE_IDEMPOTENT with a zero offset.

6.2.4 kvs_iterator_option

```
typedef enum {
    KVS_ITERATOR_OPT_KEY           = 0x00,    // [DEFAULT] iterator command gets only key entries without value
    KVS_ITERATOR_OPT_KV           = 0x01,    // iterator command gets one key and value pair
    KVS_ITERATOR_OPT_KV_WITH_DELETE = 0x02,    // iterator command gets key and value pair and delete the returned pair
} kvs_iterator_option;
```

[SAMSUNG] The Samsung Key-value SSD supports KVS_ITERATOR_OPT_KEY and KVS_ITERATOR_OPT_KV.

6.2.5 kvs_container_option

```
typedef enum {
    KVS_CONT_KEY_ORDER_NONE       = 0x00,    // [DEFAULT] key ordering is not defined in a container
    KVS_CONT_KEY_ORDER_ASCENDING  = 0x01,    // tuples are sorted in ascending key order in a container
    KVS_CONT_KEY_ORDER_DESCENDING = 0x02,    // tuples are sorted in descending key order in a container
} kvs_container_option;
```

A user can define a *container* operation option.

- **KVS_CONT_KEY_ORDER_NONE**, no key order is defined in a container.
- **KVS_CONT_KEY_ORDER_ASCENDING**, tuples are sorted in ascending key order in a container
- **KVS_CONT_KEY_ORDER_DESCENDING**, tuples are sorted in descending key order in a container

[SAMSUNG] The Samsung Key-value SSD supports KVS_CONT_KEY_ORDER_NONE.

6.2.6 kvs_result

An API returns a return value after finishing its operation.

```

enum kvs_result {
    // generic command status
    KVS_SUCCESS                                0           // success

    // Warnings
    KVS_WRN_COMPRESS                          0x0B0         // compression is not support and ignored
    KVS_WRN_MORE                              0x300         // more data is available, but buffer is not enough

    // errors
    KVS_ERR_DEV_CAPACITY                      0x012         // device does not have enough space
    KVS_ERR_DEV_INIT                         0x070         // device initialization failed
    KVS_ERR_DEV_INITIALIZED                  0x071         // device was already initialized
    KVS_ERR_DEV_NOT_EXIST                    0x072         // no device exists
    KVS_ERR_DEV_SANITIZE_FAILED              0x01C         // the previous sanitize operation failed
    KVS_ERR_DEV_SANITIZE_IN_PROGRESS         0x01D         // the sanitization operation is in progress
    KVS_ERR_ITERATOR_IN_PROGRESS             0x090         // iterator in progress

    KVS_ERR_ITERATOR_NOT_EXIST               0x394         // no iterator exists
    KVS_ERR_KEY_INVALID                      0x395         // key format is invalid
    KVS_ERR_KEY_LENGTH_INVALID               0x003         // key length is out of range (unsupported key length)
    KVS_ERR_KEY_NOT_EXIST                    0x010         // given key doesn't exist
    KVS_ERR_NS_DEFAULT                       0x095         // default namespace cannot be modified, deleted, attached, or
                                                detached
    KVS_ERR_NS_INVALID                       0x00B         // namespace does not exist
    KVS_ERR_OPTION_INVALID                   0x004         // device does not support the specified options
    KVS_ERR_PARAM_INVALID                    0x101         // no input pointer can be NULL
    KVS_ERR_PURGE_IN_PROGRESS                0x084         // purge operation is in progress
    KVS_ERR_SYS_IO                           0x303         // host failed to communicate with the device
    KVS_ERR_SYS_PERMISSION                   0x415         // caller does not have a permission to call this interface
    KVS_ERR_VALUE_LENGTH_INVALID             0x001         // value length is out of range
    KVS_ERR_VALUE_LENGTH_MISALIGNED          0x008         // value length is misaligned. Value length shall be multiples of 4
                                                bytes.
    KVS_ERR_VALUE_OFFSET_INVALID             0x002         // value offset is invalid meaning that offset is out of bound.
    KVS_ERR_VENDOR                           0x0F0         // vendor-specific error is returned, check the system log for more
                                                details

    // command specific status(errors)
    KVS_ERR_BUFFER_SMALL                     0x301         // provided buffer size too small for iterator_next operation
    KVS_ERR_DEV_MAX_NS                       0x191         // maximum number of namespaces was created
    KVS_ERR_ITERATOR_COND_INVALID            0x192         // iterator condition is not valid
    KVS_ERR_KEY_EXIST                        0x080         // given key already exists (with KVS_STORE_IDEMPOTENT option)
    KVS_ERR_NS_ATTACHED                      0x118         // namespace was already attached
    KVS_ERR_NS_CAPACITY                      0x181         // namespace capacity limit exceeds
    KVS_ERR_NS_NOT_ATTACHED                  0x11A         // device cannot detach a namespace since it has not been
                                                attached yet
    KVS_ERR_QUEUE_CQID_INVALID               0x401         // completion queue identifier is invalid
    KVS_ERR_QUEUE_SQID_INVALID               0x402         // submission queue identifier is invalid
    KVS_ERR_QUEUE_DELETION_INVALID           0x403         // cannot delete completion queue since submission queue has not
                                                been fully deleted
    KVS_ERR_QUEUE_MAX_QUEUE                  0x104         // maximum number of queues are already created
    KVS_ERR_QUEUE_QID_INVALID                0x405         // queue identifier is invalid
    KVS_ERR_QUEUE_QSIZE_INVALID              0x406         // queue size is invalid
    KVS_ERR_TIMEOUT                          0x195         // timer expired and no operation is completed yet.

```

<i>// media and data integratiy status(error)</i>		
KVS_ERR_UNCORRECTIBLE	0x781	<i>// uncorrectable error occurs</i>
KVS_ERR_QUEUE_IN_SUTDOWN	0x900	<i>// queue in shutdown mode</i>
KVS_ERR_QUEUE_IS_FULL	0x901	<i>// queue is full, unable to accept mor IO</i>
KVS_ERR_COMMAND_SUBMITTED	0x902	<i>// the beginning state after being accepted into a submission queue</i>
KVS_ERR_TOO_MANY_ITERATORS_OPEN	0x091	<i>// Exceeded max number of opened iterators</i>
KVS_ERR_ITERATOR_END	0x093	<i>// Indicate end of iterator operation</i>
KVS_ERR_SYS_BUSY	0x095	<i>//iterator next call that can return empty results, retry is recommended</i>
KVS_ERR_COMMAND_INITIALIZED	0x999	<i>// initialized by caller before submission</i>
<i>// From uDD</i>		
KVS_ERR_MISALIGNED_VALUE_OFFSET	0x09	<i>// misaligned value offset</i>
KVS_ERR_MISALIGNED_KEY_SIZE	0x0A	<i>// misaligned key length(size)</i>
KVS_ERR_UNRECOVERED_ERROR	0x11	<i>// internal I/O error</i>
KVS_ERR_MAXIMUM_VALUE_SIZE_LIMIT_EXCEEDED	0x81	<i>// value of given key is already full(KVS_MAX_TOTAL_VALUE_LEN)</i>
KVS_ERR_ITERATE_HANDLE_ALREADY_OPENED	0x92	<i>// fail to open iterator with given prefix/bitmask as it is already opened</i>
KVS_ERR_ITERATE_REQUEST_FAIL	0x94	<i>// fail to process the iterate request due to FW internal status</i>
KVS_ERR_DD_NO_DEVICE	0x100	<i>// no device exist</i>
KVS_ERR_DD_INVALID_QUEUE_TYPE	0x102	<i>// queue type is invalid</i>
KVS_ERR_DD_NO_AVAILABLE_RESOURCE	0x103	<i>// no more resource is available</i>
KVS_ERR_DD_UNSUPPORTED_CMD	0x105	<i>// invalid command (no spport)</i>
KVS_ERR_SDK_OPEN	0x200	<i>// device open failed</i>
KVS_ERR_SDK_CLOSE	0x201	<i>// device close failed</i>
KVS_ERR_CACHE_NO_CACHED_KEY	0x202	<i>// (kv cache) cache miss</i>
KVS_ERR_CACHE_INVALID_PARAM	0x203	<i>// (kv cache) invalid parameters</i>
KVS_ERR_HEAP_ALLOC_FAILURE	0x204	<i>// heap allocation fail for sdk operations</i>
KVS_ERR_SLAB_ALLOC_FAILURE	0x205	<i>// slab allocation fail for sdk operations</i>
KVS_ERR_SDK_INVALID_PARAM	0x206	<i>// invalid parameters for sdk operations</i>
KVS_ERR_DECOMPRESSION	0x302	<i>// retrieveing uncompressed value with KVS_RETRIEVE_DECOMPRESSION option</i>
<i>// Container</i>		
KVS_ERR_CONT_EXIST	0x800	<i>// container is already created with the same name</i>
KVS_ERR_CONT_NOT_EXIST	0x801	<i>// container does not existi</i>
KVS_ERR_CONT_OPEN	0x802	<i>// container is already opened</i>
KVS_ERR_CONT_CLOSE	0x803	<i>// container is closed</i>
KVS_ERR_CONT_NAME	0x804	<i>// container name is invalid</i>
KVS_ERR_GROUP_BY	0x805	<i>// group_by option is invalid</i>
KVS_ERR_INDEX	0x806	<i>// index is not valid</i>
}		

6.3 Data Structures

6.3.1 kvs_device_handle

```
struct _kvs_device_handle;           // forward declaration of _kvs_device_handle
typedef (struct _kvs_device_handle *) kvs_device_handle; // type definition of kvs_device_handle
```

A *kvs_device_handle* is an opaque data structure pointer, *struct _kvs_device_handle*. The actual data structure is implementation-specific. API programmers may define an actual data structure *_kvs_device_handle* which contains the device id and other device-related information and use the pointer type as a device handle. Or, API programmers may use an *int32_t* type with a cast to the *kvs_device_handle* type as a device handle without defining an actual data structure.

6.3.2 kvs_container_handle

```
struct _kvs_container_handle;       // forward declaration of _kvs_container_handle
typedef (struct _kvs_container_handle *) kvs_container_handle; // type definition of kvs_container_handle
```

A *kvs_container_handle* is an opaque data structure pointer, *struct _kvs_container_handle*. The actual data structure is implementation-specific. API programmers may define an actual data structure *_kvs_container_handle* which contains the container id and other container related information and use the pointer type as a container handle. Or, API programmers may use an *int32_t* type with a cast to the *kvs_container_handle* type as a container handle without defining an actual data structure.

6.3.3 kvs_group_condition

```
typedef struct {
    uint32_t bitmask;           // bit mask for bit pattern to use
    uint32_t bit_pattern;       // bit pattern for condition
} kvs_group_condition;
```

This structure defines group information for *kvs_open_iterator()* that sets up a group of keys matched with a given *bit_pattern* within a range of bits masked by *bitmask* and for *kvs_delete_group()* such that it can delete a group of key-value pairs. For more details, see *kvs_open_iterator()* (section 7.4.1) and *kvs_delete_group()* (section **Error! Reference source not found.**).

6.3.4 kvs_iterator_handle

```
struct _kvs_iterator_handle;       // forward declaration of _kvs_iterator_handle
typedef (struct _kvs_iterator_handle *) kvs_iterator_handle; // type definition of kvs_iterator_handle
```

A *kvs_iterator_handle* is an opaque data structure pointer, *struct _kvs_iterator_handle*. The actual data structure is implementation-specific. API programmers may define an actual data structure *_kvs_iterator_handle* which contains the iterator id and other iterator related information and use the pointer type as an iterator handle. Or, API programmers may use an *int32_t* type with a cast to the *kvs_iterator_handle* type as an iterator handle without defining an actual data structure.

6.3.5 kvs_iterator_list

```
typedef struct {
    uint32_t  num_entries;           // the number of iterator entries in the list
    bool_t    end;                  // represent if there are more keys to iterate (end =0) or not (end = 1)
    uint32_t  length;               // the total data length in the buffer in bytes
    uint8_t   *it_list;             // iterator list.
} kvs_iterator_list;
```

kvs_iterator_list represents entries within an iterator group. It is used for retrieved iterator entries as a return value for *kvs_iterator_next()* operation. *num_entries* specifies how many entries in the returned iterator list(*it_list*). *length* is the total amount of data returned in bytes. *it_list* has *num_entries* of iterator elements;

- When the key length is fixed, *num_entries* entries of <key> when iterator is set with *KVS_ITERATOR_OPT_KEY* (**Error! Reference source not found.**Figure 2) and *num_entries* entries of <key, value_length, value> when iterator is set with *KVS_ITERATOR_OPT_KV*(Figure 3)
- When keys have variable length, *num_entries* entries of <key_length, key> when iterator is set with *KVS_ITERATOR_OPT_KEY* (Figure 4) and *num_entries* entries of <key_length, key, value_length, value> when iterator is set with *KVS_ITERATOR_OPT_KV* (Figure 5).

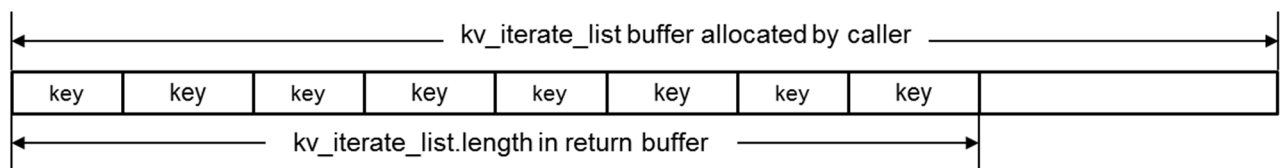


Figure 2. Fixed Key Length: KVS_ITERATOR_OPT_KEY

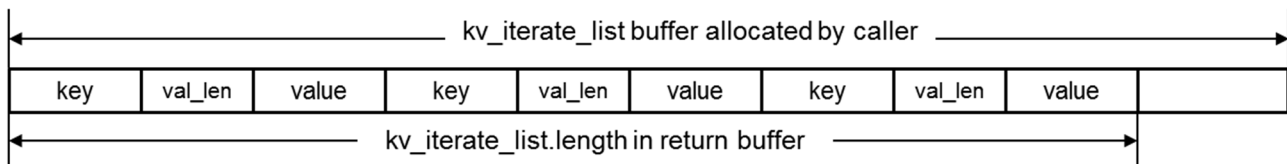


Figure 3. Fixed Key Length: KVS_ITERATOR_OPT_KV

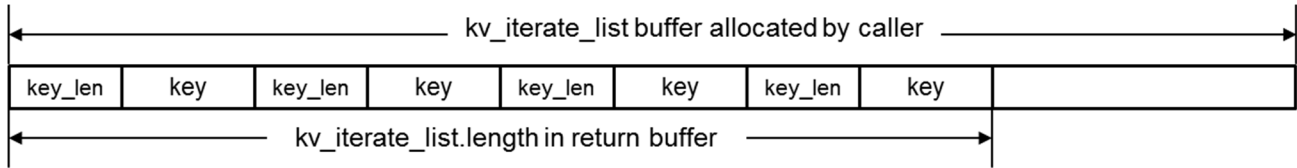


Figure 4. Variable Key Length: KVS_ITERATOR_OPT_KEY

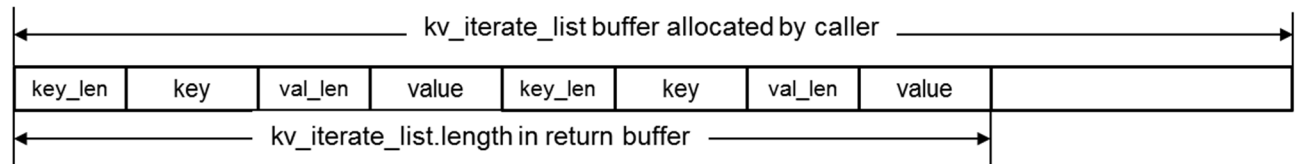


Figure 5. Variable Key Length: KVS_ITERATOR_OPT_KV

6.3.6 kvs_device

```
typedef struct {
    uint128_t capacity;           // device capacity in bytes
    uint128_t unalloc_capacity;   // device capacity in bytes that has not been allocated to any
                                // container

    uint32_t max_value_len;       // max length of value in bytes that device is able to support
    uint32_t max_key_len;         // max length of key in bytes that device is able to support
    uint32_t optimal_value_len;   // optimal value size
    uint32_t optimal_value_granularity; // optimal value granularity
    void *extended_info;         // vendor specific extended device information.
} kvs_device;
```

`kvs_device` structure represents a device and has device-wide information.

6.3.7 kvs_container

```
typedef struct {
    bool opened;                 // is this container opened
    uint8_t scale;               // indicates the scale of the capacity and free space as defined
                                // in Error! Reference source not found.

    uint64_t capacity;           // container capacity in scale units
    uint64_t free_size;          // available space of container in scale units
    uint64_t count;              // # of Key Value tuples that exist in this container
    kvs_container_name *name;    // container name
} kvs_container;
```

A container is a unit of management and represents a collection of key value tuples or key groups.

Scale specifies the scale of the capacity and free space values as defined in **Error! Reference source not found.**

Table 1. Definition of scale

Scale	Units for Capacity and Free Space
01h	4 Kbytes
All others	Reserved

6.3.8 kvs_container_name

```

type def struct {
    uint32_t name_len;           // container name length
    char *name;                 // container name
} kvs_container_name;

```

This structure contains container identification information. A device assigns unique id and an application assigns a unique name. A device is not required to check the uniqueness of container name

6.3.9 kvs_key

```

type def struct {
    void *key;                   // a void pointer refers to a key byte string
    uint16_t length;            // key length in bytes
} kvs_key;

```

A key consists of a void pointer and its length. For a container with variable keys (i.e., character string or byte string), the void *key* pointer holds a byte string without a null termination, and the integer variable of *length* holds the string byte count. The void *key* pointer must not be a null pointer.

[SAMSUNG] The valid key size ranges between 4 and 255 bytes.

6.3.10 kvs_value

```

typedef struct {
    void *value;                 // start address of buffer for value byte stream
    uint32_t length;             // the length of buffer in bytes for value byte stream
    uint32_t offset;             // [OPTION] offset to indicate the offset of value stored in device
} kvs_value;

```

A value consists of a void pointer and a length. The *value* pointer refers to a byte string without null termination, and the *length* variable holds the byte count. The *value* pointer variable cannot be a null pointer. *Offset* specifies the offset within a value stored in the device.

[SAMSUNG] The valid value size ranges between 64B and 2MB.

[SAMSUNG] Samsung Key-value SSD supports a partial retrieval of a tuple based on an offset. For example, given a tuple of 4096 byte value size, you can retrieve a portion of value ranging between the 1024th byte and the 4096th byte from a device into the *value* buffer by specifying *length* = 3072 and *offset* = 1024. However, partial stores are not supported.

6.3.11 kvs_tuple_info

```
typedef struct {  
    uint32_t key_length : 16;           // key length in bytes  
    uint32_t reserved : 16;            //  
    uint32_t value_length;              // value length in bytes  
    uint8_t key [KVS_MAX_KEY_LENGTH];  // key  
    uint8_t padding;  
} kvs_tuple_info;
```

This data structure contains tuple metadata associated with a key.

6.3.12 kvs_iterator_context

```
typedef struct {  
    uint32_t option : 8;                // option for iterator operation. one of the iterator options  
    uint32_t bitmask;                  // bit mask for bit pattern to use  
    uint32_t bit_pattern;              // bit pattern for condition  
} kvs_iterator_context;
```

This data structure contains delete operation context. The possible options are defined in the *kvs_delete_option* enum list. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.13 kvs_delete_context

```
typedef struct {  
    uint32_t option : 8;                // enum kvs_delete_option  
    uint32_t reserved : 24;  
} kvs_delete_context;
```

This data structure contains delete operation context. The possible options are defined in the *kvs_delete_option* enum list. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.14 kvs_store_context

```
typedef struct {  
    uint32_t option : 8;                // enum kvs_store_option  
    uint32_t reserved : 24;  
} kvs_store_context;
```

This data structure contains store operation context. The *kvs_store_option* enum list defines possible options. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.15 kvs_retrieve_context

```
typedef struct {  
    uint32_t option : 8;                // enum kvs_retrieve_option  
    uint32_t reserved : 24;  
} kvs_retrieve_context;
```

This data structure contains retrieve operation context. The *kvs_retrieve_option* enum list defines possible options. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.16 kvs_container_context

```
typedef struct {  
    uint32_t option : 8;                // enum kvs_container_option  
    uint32_t reserved : 24;  
} kvs_container_context;
```

This data structure contains container operation context. The *kvs_container_option* enum list defines possible options. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

7 KEY VALUE SSD APIS

7.1 Device-level APIs

7.1.1 kvs_open_device

*int32_t kvs_open_device(const char *dev_path, kvs_device_handle *dev_hd)*

This API opens a KVS device. This API internally checks device availability and initializes it. It returns a KVS_SUCCESS and set up kvs_device_handle data structure if successful. Otherwise, it returns an error code. This kvs_device_handle is used for other operations.

PARAMETERS

IN	dev_path	absolute path to a device (e.g., /dev/nvme0n1)
OUT	dev_hd	device handle data structure

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	the device does not exist
KVS_ERR_IO	communication with device failed
KVS_ERR_PARAM_INVALID	dev_path cannot be NULL or configfile cannot be NULL for emulator
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.2 kvs_close_device

int32_t kvs_close_device (kvs_device_handle dev_hd)

This API closes a KVS device. This API will store all metadata for the device into the KVS device. *dev* must be a valid pointer for the device.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

KVS_SUCCESS closing a device is successful

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev_id</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.3 kvs_get_device_info

*int32_t kvs_get_device_info(kvs_device_handle dev_hd, kvs_device *dev_info)*

This interface retrieves the kvs_device data structure, which includes device information.

PARAMETERS

IN	dev_hd	kvs_device_handle data structure that includes unique device id
OUT	dev_info	kvs_device data structure (device information)

RETURNS

kvs_device data structure

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.4 kvs_get_device_capacity

int64_t kvs_get_device_capacity(kvs_device_handle dev_hd)

This API returns KV SSD device capacity in bytes similar to block devices.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

OUT int64_t device capacity in bytes.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.5 kvs_get_device_utilization

int32_t kvs_get_device_utilization(kvs_device_handle dev_hd)

This interface returns the device utilization (i.e, used ratio of the device) by the given device identifier. The utilization is from 0(0.00% utilized) to 10000(100%).

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

Used ratio of the device (0 ~ 10000)

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.6 kvs_get_min_key_length

int32_t kvs_get_min_key_length (kvs_device_handle dev_hd)

This interface returns the minimum length of key that the device supports.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

The minimum length of keys that the device supports.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.7 kvs_get_max_key_length

int32_t kvs_get_max_key_length (kvs_device_handle dev_hd)

This interface returns the maximum length of key that the device supports.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

The maximum length of keys that the device supports.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.8 kvs_get_min_value_length

int32_t kvs_get_min_value_length (kvs_device_handle dev_hd)

This interface returns the minimum length of value that the device supports.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

The minimum length of value that the device supports.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.9 kvs_get_max_value_length

int32_t kvs_get_max_value_length (kvs_device_handle dev_hd)

This interface returns the maximum length of value that the device supports.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

The maximum length of value that the device supports.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.10 kvs_get_optimal_value_length

int32_t kvs_get_optimal_value_length (kvs_device_handle dev_hd)

This interface returns the optimal length of value that the device supports. The device will perform best when the value size is the same the optimal value size.

PARAMETERS

IN dev_hd kvs_device_handle data structure that includes unique device id

RETURNS

The optimal length of value that the device supports.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the device identifier
KVS_ERR_IO	communication with device failed

7.1.11 kvs_create_container

*int32_t kvs_create_container (kvs_device_handle dev_hd, const char *name, uint64_t sz_4kb, const kvs_container_context *ctx)*

This API creates a new container in a device. A user needs to specify a unique container name as a null terminated string, and its capacity. The capacity is defined in 4KB units. A 0 (numeric zero) capacity of means no limitation where device capacity limits actual container capacity. The device assigns a unique id while a user assigns a unique name.

If a *ctx.option* is set to:

- **KVS_GROUP_ORDER_NONE**, no group order is defined.
- **KVS_GROUP_ORDER_ASCENDING**, tuples are sorted in ascending key order in the container. .
- **KVS_GROUP_ORDER_DESCENDING**, tuples are sorted in descending key order in the container.

[SAMSUNG] Samsung KV SSD supports only one container. Samsung may support multiple containers in future KV SSD generations.

[SAMSUNG] The Samsung KV SSD supports **KVS_GROUP_ORDER_NONE** only.

PARAMETERS

IN dev_hd	kvs_device_handle data structure that includes unique device id
IN name	name of container
IN sz_4kb	capacity of a container with respect to tuple size (key size + value size) in 4KB units
IN ctx	container context (i.e., key ordering option in a container)

RETURNS

container handle data structure.

ERROR CODE

KVS_ERR_DEV_CAPACITY	the container size is too big
KVS_ERR_CONT_EXIST	container with the same name already exists
KVS_ERR_CONT_NAME	container name does not meet the requirement (e.g., too long)
KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev_id</i> exists
KVS_ERR_GROUP_BY	<i>ctx.option</i> is not valid
KVS_ERR_IO	communication with device failed
KVS_ERR_PARAM_INVALID	<i>name</i> or <i>group_by</i> is NULL
KVS_ERR_OPTION_INVALID	multiple containers are not supported

7.1.12 kvs_delete_container

*int32_t kvs_delete_container (kvs_device_handle dev_hd, const char *cont_name)*

This API destroys a container identified by the given device and container name. It drops all tuples within the container as well as container itself.

PARAMETERS

IN dev_hd	kvs_device_handle data structure that includes unique device id
IN cont_name	container name

RETURNS

KVS_SUCCESS	a container is destroyed successfully
--------------------	---------------------------------------

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_path</i> does not exist
KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev_id</i> exists
KVS_ERR_IO	communication with device failed

7.1.13 kvs_list_containers

*int32_t kvs_list_containers (kvs_device_handle dev_hd, uint32_t index, uint32_t buffer_size, kvs_container_names *names, uint32_t *cont_cnt)*

For a KVS device, this API returns the names of up to the number of containers specified in *num_cont*. A device may define a unique order of container ID and index is defined relative to that order. The *index* specifies a start list entry offset, *buffer_size* specifies the size of *kvs_container_names* array, and *names* is a buffer to store container name data structure. This returns a number of container names in the device. *cont_cnt* is set by the number of entries in the *names* array as an output.

PARAMETERS

IN	dev_hd	kvs_device_handle data structure that includes unique device id
IN	buffer_size	names buffer size in bytes
IN/OUT	names	buffer to store container names. This buffer must be preallocated before calling this routine.
OUT	cont_cnt	the number of <i>kvs_container_names</i> stored in the buffer

RETURNS

KVS_SUCCESS if the operation is successful

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device with the <i>dev_id</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_INDEX	<i>index</i> is not valid
KVS_ERR_PARAM_INVALID	<i>name_ids</i> or <i>conts_cnt</i> is NULL

7.2 Container-level APIs

7.2.1 kvs_open_container

int32_t kvs_open_container (kvs_device_handle dev_hd, const char name, kvs_container_handle *cont_hd)*

This API opens a container with a given name. This API communicates with a device to initialize the corresponding container. The device is capable of recognizing and initializing the container. If the container is already open, this API returns [KVS_ERR_CONT_OPEN](#). This container handle is unique within a process.

PARAMETERS

IN dev_hd	kvs_device_handle data structure that includes unique device id
IN cont	container name
OUT cont_hd	container handle

RETURNS

KVS_SUCCESS if it is successful, otherwise it returns error.

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	Container with the given <i>cont_hd</i> does not exist,
KVS_ERR_IO	Communication with device failed
KVS_ERR_CONT_OPEN	container has been opened already

7.2.2 kvs_close_container

int32_t kvs_close_container (kvs_container_handle cont_hd)

This API closes a container with a given container handle. This API communicates with the device to close the corresponding container. This API may clean up any internal container states in the device. If the given container was not open, this returns a `KVS_ERR_CONT_CLOSE` error.

PARAMETERS

IN cont_hd container handle

RETURNS

`KVS_SUCCESS` to indicate that closing a container is successful.

ERROR CODE

<code>KVS_ERR_CONT_CLOSE</code>	cannot close the container
<code>KVS_ERR_CONT_NOT_EXIST</code>	container with a given <i>cont</i> does not exist
<code>KVS_ERR_IO</code>	communication with device failed

7.2.3 kvs_get_container_info

*int32_t kvs_get_container_info (kvs_container_handle cont_hd, kvs_container *cont)*

This API retrieves container information. This API can be called anytime, regardless of whether the container is open or not.

PARAMETERS

IN cont_hd	Container handle
OUT cont	container information

RETURNS

KVS_SUCCESS to indicate that getting container info is successful.

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_IO	communication with device failed
KVS_ERR_PARAM_INVALID	<i>cont</i> is NULL

7.3 Key-value tuple APIs

7.3.1 kvs_get_tuple_info

*int32_t kvs_get_tuple_info (kvs_container_handle cont_hd, const kvs_key *key, kvs_tuple_info *info)*

This API retrieves tuple metadata information. Tuple metadata includes a key length, a key byte stream, and a value length. Please refer to section 6.3.11 *kvs_tuple_info* for details. This API is intended to be used when a buffer length for a value is not known. The caller should create *kvs_tuple_info* object before calling this API.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN key	key to find for tuple metadata info
OUT info	tuple metadata information

RETURNS

KVS_SUCCESS indict that retrieving tuple metadata info is successful.

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given <i>key</i> is not supported (e.g., length)
KVS_ERR_PARAM_INVALID	<i>key</i> or <i>info</i> is NULL
KVS_ERR_KEY_NOT_EXIST	<i>key</i> does not exist

7.3.2 kvs_retrieve_tuple

*int32_t kvs_retrieve_tuple (kvs_container_handle cont_hd, const kvs_key *key, kvs_value *value, const kvs_retrieve_context *ctx)*

This API retrieves a tuple value with the given key. The value parameter contains output buffer information for the value. Value.value contains the buffer to store the tuple value and value.size contains the buffer size. The tuple value is copied to value.value buffer and value.size is set to the actual size of value. If the offset of value is not zero, the value of tuple is copied into the buffer, skipping the first offset bytes of the value of tuple. The actual value size copied to the output buffer is set to value.size. That is, value.size is equal to the total size of (*value* – *offset*). The offset must align to KVS_ALIGNMENT_UNIT. This API supports two modes section 6.2.2 kvs_retrieve_option describes. If an allocated value buffer is not big enough to hold the value, it will set kvs_value.length by the actual value length and return KVS_ERR_MEMORY.

A user can define a retrieve operation option:

- **KVS_RETRIEVE_IDEMPOTENT** defines that a retrieve operation is idempotent and a user can safely repeatedly read the tuple value. This is the default retrieve operation behavior.
- **KVS_RETRIEVE_DELETE** defines that an operation reads the tuple value but the tuple is atomically deleted with completion. Therefore, this optional operation is not idempotent and unsafe (a cached tuple may be stale).

[SAMSUNG] Samsung only supports the **KVS_RETRIEVE_IDEMPOTENT** option.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN key	key of the tuple to get value
OUT value	value to receive the tuple's value from device
IN ctx	retrieve context. It can be NULL. In that case, the default get context will be used. It is vendor specific.

RETURNS

KVS_SUCCESS indict that the retrieval operation is successful.

ERROR CODE

KVS_ERR_VALUE_OFFSET_INVALID	<i>kvs_value.offset</i> is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given <i>key</i> is not supported (e.g., length)
KVS_ERR_MEMORY	buffer space of <i>value</i> is not allocated or not enough
KVS_ERR_PARAM_INVALID	<i>key</i> or <i>value</i> is NULL

KVS_ERR_VALUE_OFFSET_INVALID	<i>kvs_value.offset</i> is invalid
KVS_ERR_OPTION_INVALID	the option in the <i>ctx</i> is not supported
KVS_ERR_KEY_NOT_EXIST	key does not exist

7.3.3 kvs_store_tuple

*int32_t kvs_store_tuple (kvs_container_handle cont_hd, const kvs_key *key, const kvs_value *value, const kvs_store_context *ctx)*

This API writes a Key-value tuple into a device. This API supports several modes section 6.2.3 kvs_store_option defines. A user can define a store operation option:

- **KVS_STORE_POST** defines an atomic, non-idempotent store operation. If a key does not exist and the offset of the *value* parameter is equal to zero, a new tuple is created. If a key does not exist and the offset is non-zero, it returns KVS_ERR_OFFSET. If a key exists, a new value replaces the current tuple value. This is similar to a database *upsert*. The *offset* of the *value* parameter is only valid with a KVS_STORE_POST option. If an offset is positive, the tuple head portion remains. That is, the content between 0 and offset of the stored value of the tuple is kept and the remaining portion of the value is replaced with the new value in the buffer. If the *offset* is larger than the size of the stored tuple, it returns an error of KVS_ERR_OFFSET. The *value* size of the new tuple is equal to the sum of *offset* and *value.size*. If an *offset* is negative, the tail portion of tuple remains. That is, the content between *value.size+offset* and value size of the stored value of the tuple is kept and the remaining portion of the value is replaced with the new value in the buffer. If the absolute value of offset is larger than the size of the stored tuple, it returns a KVS_ERR_OFFSET error. The *value* size of the new tuple is equal to the sum of the absolute value of offset and *value.size*. The offset must align to KVS_ALIGNMENT_UNIT.
- **KVS_STORE_IDEMPOTENT** specifies a store operation is idempotent and a user can only write a tuple value once. If a key does not exist, this operation succeeds and a new tuple is stored. Otherwise, this operation fails, without affecting the stored tuple, and it returns a KVS_ERR_KEY_EXIST error. The offset of the *value* parameter is ignored with this option. This is the default store operation behavior.
- **KVS_STORE_APPEND** allows a user to append a value to a tuple. It is an atomic, non-idempotent store operation. If a tuple does not exist, a new tuple is created. This option can be used when (1) a user stores a large tuple which cannot be stored with a single store operation or (2) creates a log-like object. The offset of the value parameter is ignored with this option.

Regardless of the existence of *key*, all store operations atomically execute and the final *key* value will be determined by the order of successful operations. If the device does not have enough space to store a tuple, a KVS_ERR_CAPACITY error message is returned.

[SAMSUNG] Samsung supports the *KVS_STORE_IDEMPOTENT* option and the *KVS_STORE_POST* option with the *offset* of value equal to zero.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN key	key of the tuple to put into device
IN value	value of the tuple to put into device

IN *ctx* store context. It can be NULL. In that case, the default put context will be used. It is vendor specific.

RETURNS

KVS_SUCCESS indict that writing a tuple is successful.

ERROR CODE

KVS_ERR_OFFSET_INVALID	<i>kvs_value.offset</i> is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given <i>key</i> is not supported (e.g., length)
KVS_ERR_PARAM_INVALID	a <i>key</i> or a <i>value</i> is NULL
KVS_ERR_OFFSET	<i>kvs_value.offset</i> is invalid
KVE_ERR_OPTION_INVALID	unsupported option is specified in <i>ctx</i>
KVS_ERR_CAPACITY	device does not have enough space to store this tuple
KVS_ERR_KEY_EXIST	a key exists but overwrite is not permitted
KVS_ERR_VALUE_LENGTH_INVALID	given value is not supported (e.g., length)

7.3.4 kvs_delete_tuple

int32_t kvs_delete_tuple (kvs_container_handle cont_hd, const kvs_key key, const kvs_delete_context* ctx)*

This API deletes a key-value tuple with a given key. A user can define a delete operation option in *kvs_delete_option*.

- **KVS_DELETE_TUPLE** defines an idempotent, atomic operation that removes a tuple matching the given key. This is the default delete operation behavior.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN key	key to delete
IN ctx	delete context. It can be NULL. In that case, the default drop context will be used. It is vendor specific.

RETURNS

KVS_SUCCESS	indicate that dropping is successful
--------------------	--------------------------------------

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_PARAM_INVALID	<i>key</i> is NULL
KVS_ERR_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given <i>key</i> is not supported (e.g., length)
KVS_ERR_OPTION_INVALID	option in <i>ctx</i> is not supported
KVS_ERR_KEY_NOT_EXIST	<i>key</i> does not exist

7.3.5 kvs_exist_tuples

*int32_t kvs_exist_tuples (kvs_container_handle cont_hd, uint32_t key_cnt, const kvs_key *keys, uint32_t *key_num, uint32_t buffer_size, uint8 *result_buffer)*

This API checks if a set of one or more keys exists and returns a *bool* type status. The existence of a key value pair is determined during an implementation-dependent time window while this API executes. Therefore, repeated routine calls may return different outputs in multi-threaded environments. One bit is used for each key. Therefore when 32 keys are intended to be checked, a caller should allocate 32 bits (i.e., 4 bytes) of memory buffer and the existence information is filled. The LSB (Least Significant Bit) of the *result_buffer* indicates if the first key exist or not.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN key_cnt	the number of keys to check
IN keys	a set of keys to check
IN buffer_size	result buffer size in bytes
OUT key_num	the number of bits in the result buffer that represent the existence of a key
OUT result_buffer	a list of bool value whether corresponding key(s) exists or not

RETURNS

KVS_SUCCESS	Indict that the routine is successful.
-------------	--

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given <i>cont_hd</i> does not exist
KVS_ERR_MEMORY	the buffer space of <i>results</i> is not big enough
KVS_ERR_PARAM_INVALID	<i>keys</i> or <i>results</i> parameter is NULL
KVS_ERR_IO	communication with device failed

7.4 Iterator APIs

7.4.1 kvs_open_iterator

int32_t kvs_open_iterator(kvs_container_handle cont_hd, const kvs_iterator_context *ctx, kvs_iterator_handle *iter_hd)

This interface enables applications to set up a key group such that the keys in that key group may be iterated. (i.e., *kvs_open_iterator()* enables a device to prepare a key group of keys for iteration by matching a given bit pattern (*ctx.bit_pattern*) to all keys in the device considering bits indicated by *ctx.bitmask* and the device sets up a key group of keys matching that “(*bitmask* & key) == *bit_pattern*”). For example, if the *bitmask* and *bit_pattern* are 0xF0000000 and 0x30000000 respectively, then *kvs_open_iterator* will prepare a subset of keys which has 0x3XXXXXXX in keys.

Below are some examples with a group size of 4.

- 1) If applications want to get all the existing keys within the device with the first bit of a key is 1, *kvs_open_iterator()* should be called with *bitmask* = 0x80000000 (1000 0000 0000 0000 0000 0000 0000) and *bit_pattern* = 0x80000000 (1000 0000 0000 0000 0000 0000 0000).
- 2) If applications want to get all the existing keys within the device with the first bit of key is 0, *bitmask* should be 0x80000000 (1000 0000 0000 0000 0000 0000 0000) and *bit_pattern* should be 0x0 (0000 0000 0000 0000 0000 0000 0000).
- 3) If applications want to get all the existing keys with the second bytes (bit 8 ~ bit15) is equal to 0x04, *bitmask* should be 0xFFFF0000 (1111 1111 1111 1111 0000 0000 0000 0000) and *bit_pattern* should be 0x00040000 (0000 0000 0000 0100 0000 0000 0000 0000).
- 4) If application wants to get all the existing keys with bit 1 ~ bit 4 is equal to (0101), *bitmask* should be 0xF8000000 (1111 1000 0000 0000 0000 0000 0000 0000) and *bit_pattern* should be 0x28000000 (0010 1000 0000 0000 0000 0000 0000 0000).

It also sets up the iterator option (i.e., *ctx.option*); *kvs_iterator_next()* will only retrieve keys when the *kvs_iterator_option* is *KVS_ITERATOR_OPT_KEY* while *kvs_iterator_next()* will retrieve key and value pairs when the *kvs_iterator_option* is *KVS_ITERATOR_OPT_KV*.

Finally it will return an *iterator identifier*.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN ctx	iterator context (refer to the section 6.3.12)
OUT iter_hd	iterator handle

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with <i>cont_hd</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_OPTION_INVALID	the device does not support the specified iterator options
KVS_ERR_ITERATOR_COND_INVALID	iterator filter(match bitmask and pattern) is not valid

7.4.2 kvs_close_iterator

*int32_t kvs_close_iterator(kvs_container_handle cont_hd, kvs_iterator_handle *iter_hd, const kvs_iterator_context *ctx)*

This interface releases the given iterator key group of *iter_hd* in the given container. So the iterator operation ends.

PARAMETERS

IN cont_hd kvs_container_handle data structure that includes unique container id
IN iter_hd iterator handle
IN ctx iterator context

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with <i>cont_hd</i> exists
KVS_ERR_IO	communication with device failed
KVS_ERR_ITERATOR_NOT_EXIST	the iterator Key Group does not exist

7.4.3 kvs_iterator_next

*int32_t kvs_iterator_next(kvs_container_handle cont_hd, kvs_iterator_handle iter_hd, uint32_t iter_size, kvs_iterator_list *iter_list, const kvs_iterator_context *ctx)*

This interface obtains a subset of key or a key-value pair(s) from a key group of *iter_hd* in a device (i.e., *kvs_iterator_next()* retrieves the next key group of keys or a key-value pair(s) in the iterator key group (*iter_hd*) that is set with *kvs_open_iterator()* command). *iter_list.size* is the iterator buffer (*iter_list*) size in bytes. The retrieved values (*iter_list.it_list*) are either keys or key-value pairs based on the iterator option which is set by *kvs_open_iterator()*.

When *kvs_store_tuple()* or *kvs_delete_tuple()* command whose key matches with an existing key group is received, the keys may or may not be included in the iterator and the inclusion of the updated keys is unspecified.

In the output of this operation, *iter_list.num_entries* provides the number of iterator elements in *iter_list.it_list*. The KVS_ERR_ITERATE_END error message is returned when there is no more iterator group elements meaning that iterator reaches the end. If the return value is not KVS_ERR_ITERATE_END, there are more iterator key group elements and the host may run *kvs_iterator_next()* again to retrieve those elements.

Output values (*iter_list.it_list*) are determined by the iterator option set by an application.

- **KVS_ITERATOR_OPT_KEY [MANDATORY]**: a subset of keys are returned in *iter_list.it_list* data structure
- **KVS_ITERATOR_OPT_KV**: a subset of key-value pairs are returned in *iter_list.it_list* data structure

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN iter_hd	iterator handle
IN iter_size	iterator array (<i>iter_list</i>) buffer size
IN ctx	iterator context
IN/OUT	output buffer for a set of keys or key-value pairs stored in the buffer

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with <i>cont_hd</i> exists
KVS_ERR_PARAM_INVALID	<i>iter_list</i> parameter is NULL
KVS_ERR_IO	communication with device failed
KVS_ERR_ITERATOR_NOT_EXIST	the iterator Key Group does not exist