Rev. 1.0.0 May. 2019

Samsung® *Key-value* SSD API

Specification

Revision History

Revision No.	History	Draft Date	Remark
Version 1.0.0	Samsung Key-value SSD API spec 1.0	05/10/2019	
Version 0.8.0	Refined iterator open operation	02/28/2019	
Version 0.7.0	Refined the existing APIs	12/12/2018	
Version 0.6.0	Add async APIs and refined the existing APIs	09/07/2018	
Version 0.5.0	Samsung Key-value SSD API spec first draft	08/10/2018	

Contents

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 v0.6.0	PM983	NVMe 1.2

2 TERMINOLOGY

2.1 Acronyms and Definitions

Acronym/Term	Description
API	Application Programming Interface
KVS	Key-value SSD
NVMe	NVM Express (Non-Volatile Memory Express)
PCIe	PCI Express (Peripheral Component Interconnect Express)
SSD	Solid State Drive
Tuple	Object defined by a tuple of key and value
Container	A collection of tuples identified by a name and it is a unit of management in KVS

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_info	V0.6.0	
Kvs_get_device_capacity	V0.6.0	
Kvs_get_device_utilization	V0.6.0	
Kvs_get_device_waf	V1.0.0	
Kvs_get_min_key_length	V0.6.0	
Kvs_get_max_key_length	V0.6.0	
Kvs_get_min_value_length	V0.6.0	
Kvs_get_max_value_length	V0.6.0	
Kvs_get_optimal_value_length	V0.6.0	
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	V0.6.0	
Kvs_retrieve_tuple	V0.5	
Kvs_retrieve_tuple_async	V0.5	
Kvs_store_tuple	V0.5	
Kvs_store_tuple_async	V0.5	
Kvs_delete_tuple	V0.5	
Kvs_delete_tuple_async	V0.5	
Kvs_exist_tuples	V0.6.0	
Kvs_exist_tuples_async	V0.6.0	
Kvs_open_iterator	V0.5	
Kvs_close_iterator	V0.5	
Kvs_iterator_next	V0.5	
Kvs_iterator_next_async	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/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.

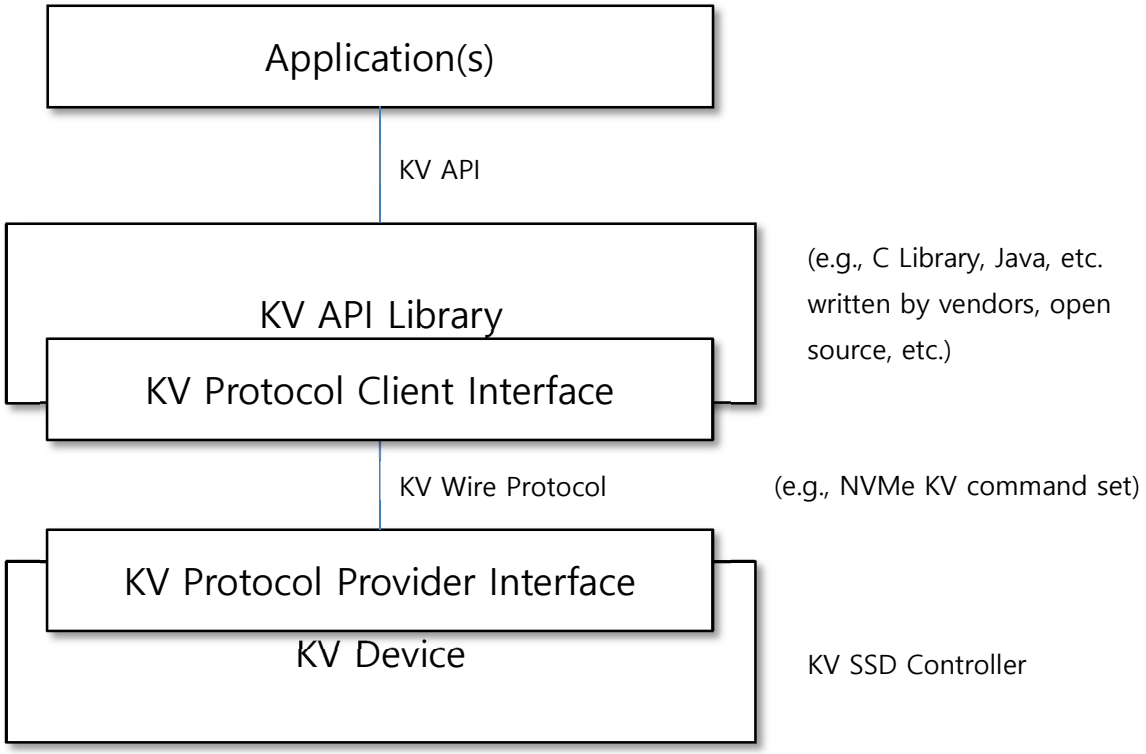

Figure 1. Key-value Architecture

(**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

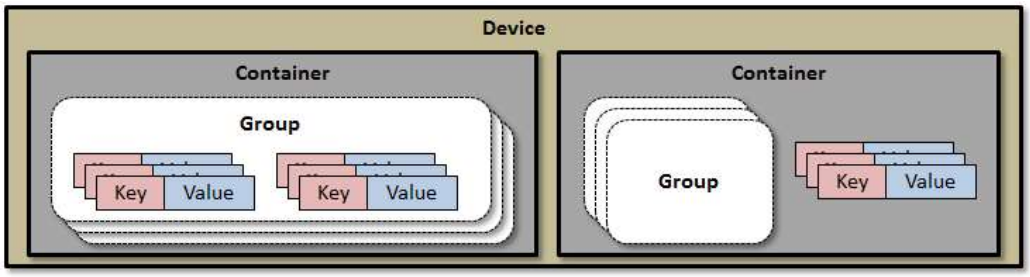

Figure 1. Key-value Objects

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, NVMoF, 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.

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 latest Samsung key-value SSD does not have this limiation. *[deprecated]*

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 may be used to report the maximum key length that a device can supports.

[SAMSUNG] The Samsung KV SSD supports up to 255-byte key.

6.1.3 KVS_MIN_KEY_LENGTH

The minimum key length that a device is able to support. This is set when a device is opened (e.g., the Identify Namespace Data Structure in the NVMe spec may be used to report the minimum key length that a device is capable of supporting) and is the same for all Containers in the device.

[SAMSUNG] The minimum key length for Samsung KV SSD is 4 bytes.

6.1.4 KVS_MAX_VALUE_LENGTH

The maximum value length that a device is able to support. This is set when a device is opened (e.g., the Identify Namespace Data Structure in the NVMe spec may be used to report the maximum value length that a device is capable of supporting) and is the same for all Containers in the device.

[SAMSUNG] The Samsung KV SSD supports up to 2MB value length.

6.1.5 KVS_MIN_VALUE_LENGTH

The minimum vlaue length that a device is able to support. This is set when a device is opened (e.g., the Identify Namespace Data Structure in the NVMe spec may be used to report the minimum value length that a device is capable of supporting) and is the same for all Containers in the device.

[SAMSUNG] The minum value length for Samsung KV SSD is 0 byte.

6.1.6 KVS_MAX_ITERATE_HANDLE

The maximum number of iterators a device is able to support.

[SAMSUNG] The maximum number of iterators for Samsung KV SSD is 16.

6.2 Enum Constants

6.2.1 kvs_iterator_type

```
typedef enum {
    KVS_ITERATOR_KEY          =0,   // [DEFAULT] iterator command retrieves only key entries without values
KVS_ITERATOR_KEY_VALUE        =1,   // [OPTION] iterator command retrieves key and value tuples
KVS_ITERATOR_WITH_DELETE      = 2,  // [OPTION] iterator command retrieves key and delete
} kvs_iterator_type;
```

6.2.2 kvs_key_order

```
typedef enum {
   KVS_KEY_ORDER_NONE        =0,   // [DEFAULT] key ordering is not defined in a Container
   KVS_KEY_ORDER_ASCEND,     =1,   // [OPTION] key value tuples are sorted in ascending key order in a Container
    KVS_KEY_ORDER_DESCEND    =2,   // [OPTION] key value tuples are sorted in descending key order in a Container
} kvs_key_order;
```

A user can define a *container* operation option.

- **KVS_KEY_ORDER_NONE**, no key order is defined in a container.
- **KVS_KEY_ORDER_ASCENDING**, tuples are sorted in ascending key order in a container
- **KVS_KEY_ORDER_DESCENDING**, tuples are sorted in descending key order in a container

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

6.2.3 kvs_store_type

```
typedef enum {
    KVS_STORE_POST            =0,   // [DEFAULT] store key value tuple
    KVS_STORE_UPDATE_ONLY     =1,   // [OPTION] update only
    KVS_STORE_NOOVERWRITE     =2,   // [OPTION] no overwrite (=idempotent)
    KVS_STORE_APPEND          =3,   // [OPTION] append
} kvs_store_type;;
```

The application is able to specify a store operation option.

- KVS_STORE_POST: if the key exist, the operation overwrites value. if the key does not exist, it inserts the key value tuple.
- KVS_STORE_UPDATE_ONLY: if the key exist, the operation overwrites value. if the key does not exist, it returns KVS_KEY_NOT_EXIST error.
- KVS_STORE_NOOVERWIRTE: if the key exist, the operation returns KVS_ERR_VALUE_UPDATE_NOT_ALLOWED. If the key does not exist, it inserts the key value tuple. (=idempotent)
- KVS_STORE_APPEND: if the key exist, the operation appends the value to the existing value. if the key does not exist, it inserts the key value tuple.

[SAMSUNG] The Samsung Key-value SSD supports the KVS_STORE_POST and KVS_STORE_NOOVERWRITE (=idempotent) only.

6.2.4 kvs_result

An API returns a return value after finishing its operation.

```
// kvs_result messages (enum type);
   // generic command status
   KVS_SUCCESS                          0       // success

   // errors
   KVS_ERR_BUFFER_SMALL                 0x001   // provided buffer size too small
   KVS_ERR_COMMAND_INITIALIZED          0x002   // initialized by caller before submission
   KVS_ERR_COMMAND_SUBMITTED            0x003   // the beginning state after being accepted into a submission
                                                queue
   KVS_ERR_DEV_CAPACITY                 0x004   // device does not have enough space
   KVS_ERR_DEV_INIT                     0x005   // device initialization failed
   KVS_ERR_DEV_INITIALIZED              0x006   // device was already initialized
   KVS_ERR_DEV_NOT_EXIST                0x007   // no device exists
   KVS_ERR_DEV_SANITIZE_FAILED          0x008   // the previous sanitize operation failed
   KVS_ERR_DEV_SANIZE_IN_PROGRESS       0x009   // the sanitization operation is in progress
   KVS_ERR_ITERATOR_COND_INVALID        0x00A   // iterator condition is not valid
   KVS_ERR_ITERATOR_MAX                 0x00B   // Exceeded max number of opened iterators
   KVS_ERR_ITERATOR_NOT_EXIST           0x00C   // no iterator exists
   KVS_ERR_ITERATOR_OPEN                0x00D   // iterator is already open
   KVS_ERR_KEY_EXIST                    0x00E   // given key already exists (with KVS_STORE_IDEMPOTENT option)
                                                // key format is invalid
   KVS_ERR_KEY_INVALID                  0x00F   // key length is out of range (unsupported key length)
   KVS_ERR_KEY_LENGTH_INVALID           0x010   // given key doesn't exist
   KVS_ERR_KEY_NOT_EXIST                0x011   // device does not support the specified options
   KVS_ERR_OPTION_INVALID               0x012   // no input pointer can be NULL
   KVS_ERR_PARAM_INVALID                0x013   // purge operation is in progress
   KVS_ERR_PURGE_IN_PROGRESS            0x014   // completion queue identifier is invalid
   KVS_ERR_QUEUE_CQID_INVALID           0x015   // cannot delete completion queue since submission queue has not
   KVS_ERR_QUEUE_DELETION_INVALID       0x016   been fully deleted
                                                // queue in shutdown mode
   KVS_ERR_QUEUE_IN_SUTDOWN             0x017   // queue is full, unable to accept mor IO
   KVS_ERR_QUEUE_IS_FULL                0x018   // maximum number of queues are already created
   KVS_ERR_QUEUE_MAX_QUEUE              0x019   // queue identifier is invalid
   KVS_ERR_QUEUE_QID_INVALID            0x01A   // queue size is invalid
   KVS_ERR_QUEUE_QSIZE_INVALID          0x01B   // submission queue identifier is invalid
   KVS_ERR_QUEUE_SQID_INVALID           0x01C   //iterator next call that can return empty results, retry is
   KVS_ERR_SYS_BUSY                     0x01D   recommended
                                                // host failed to communicate with the device
   KVS_ERR_SYS_IO                       0x01E   // timer expired and no operation is completed yet.
   KVS_ERR_TIMEOUT                      0x01F   // uncorrectable error occurs
   KVS_ERR_UNCORRECTIBLE                0x020   // value length is out of range
   KVS_ERR_VALUE_LENGTH_INVALID         0x021   // value length is misaligned. Value length shall be multiples of 4
   KVS_ERR_VALUE_LENGTH_MISALIGNED      0x022   bytes.[deprecated]
                                                // value offset is invalid meaning that offset is out of bound.
   KVS_ERR_VALUE_OFFSET_INVALID         0x023   // key exists but value update is not allowed
   KVS_ERR_VALUE_UPDATE_NOT_ALLOWED     0x024   // vendor-specific error is returned, check the system log for more
   KVS_ERR_VENDOR                       0x025   details
                                                // unable to open device due to permission
   KVS_ERR_PERMISSION                   0x026
```

```
// From user driver
KVS_ERR_CACHE_INVALID_PARAM                0x200   // (kv cache) invalid parameters
KVS_ERR_CACHE_NO_CACHED_KEY                0x201   // (kv cache) cache miss
KVS_ERR_DD_INVALID_QUEUE_TYPE              0x202   // queue type is invalid
KVS_ERR_DD_NO_AVAILABLE_RESOURCE           0x203   // no more resource is available
KVS_ERR_DD_NO_DEVICE                       0x204   // no device exist
KVS_ERR_DD_UNSUPPORTED_CMD                 0x205   // invalid command (no spport)
KVS_ERR_DECOMPRESSION                      0x206   // retrieveing uncompressed value with
                                                      KVS_RETRIEVE_DECOMPRESSION option
                                                   // heap allocation fail for sdk operations
KVS_ERR_HEAP_ALLOC_FAILURE                 0x207   // fail to open iterator with given prefix/bitmask as it is already
KVS_ERR_ITERATE_HANDLE_ALREADY_OPENED      0x208   opened
                                                   // fail to process the iterate request due to FW internal status
KVS_ERR_ITERATE_REQUEST_FAIL               0x209   // value of given key is already full
KVS_ERR_MAXIMUM_VALUE_SIZE_LIMIT_EXCEEDED  0x20A   (KVS_MAX_TOTAL_VALUE_LEN)
                                                   // misaligned key length(size)
KVS_ERR_MISALIGNED_KEY_SIZE                0x20B   // misaligned value offset
KVS_ERR_MISALIGNED_VALUE_OFFSET            0x20C   // device close failed
KVS_ERR_SDK_CLOSE                          0x20D   // invalid parameters for sdk operations
KVS_ERR_SDK_INVALID_PARAM                  0x20E   // device open failed
KVS_ERR_SDK_OPEN                           0x20F   // slab allocation fail for sdk operations
KVS_ERR_SLAB_ALLOC_FAILURE                 0x210   // internal I/O error
KVS_ERR_UNRECOVERED_ERROR                  0x211

// from emulator and Kernel driver
KVS_ERR_NS_ATTACHED                        0x300   // namespace is already attached
KVS_ERR_NS_CAPACITY                        0x301   // namespace does not have enough space
KVS_ERR_NS_DEFAULT                         0x302   // default namespace can not be modified, deleted, attached or
                                                   detached
                                                   // namespace does not exist
KVS_ERR_NS_INVALID                         0x303   // maximum number of namespaces were created
KVS_ERR_NS_MAX                             0x304   // device cannot detach a namespace since it has not been fully
KVS_ERR_NS_NOT_ATTACHED                    0x305   deleted

// Container
KVS_ERR_CONT_CAPACITY                      0x400   // conatainer does not have enough space
KVS_ERR_CONT_CLOSE                         0x401   // container is closed
KVS_ERR_CONT_EXIST                         0x402   // container is already created with the same name
KVS_ERR_CONT_INDEX                         0x404   // index is not valid
KVS_ERR_CONT_NAME                          0x405   // container name is invalid
KVS_ERR_CONT_NOT_EXIST                     0x406   // container does not existi
KVS_ERR_CONT_OPEN                          0x407   // container is already opened
```

6.3 Data Structures

6.3.1 kvs_init_options

```
typedef struct {
   struct {
      int use_dpdk;                          // use DPDK as a memory allocator. It should be 1 if SPDK driver is in use.
      int dpdk_mastercoreid;                 // specify a DPDK master core ID
      int nr_hugepages_per_socket;           // number of 2MB huge pages per socket available in the socket mask
      uint16_t socketmask;                   // a bitmask for CPU sockets to be used
      uint64_t max_memorysize_mb;            // the maximum amount of memory
      uint64_t max_cachesize_mb;             // the maximum cache size in MB
   } memory;                                 // Only initialize this memory option for SPDK driver

   struct {
      uint64_t iocoremask;                   // a bitmask for CPUs to be used for I/O
      uint32_t queuedepth;                   // the maximum queue depth
   } aio;

   struct {
      char core_mask_str[256];               // core ids used for submission queue when using spdk driver
      char cq_thread_mask[256];              // core ids used for completion queue when using spdk driver
      uint32_t mem_size_mb;                  // shared memory size in MB
      int syncio;                            // 1: sync I/O; 0: async I/O
   } udd;                                    // Only initialize this option for SPDK driver
      const char *emul_config_file;          // path to the emulator config file if using kvssd emulator
   } kvs_init_options;
```

A kvs_init_options contains all available options in the library. kvs_init_env_opts() initializes this with default values. kvs_init_env() takes this option to initializes the library.

6.3.2 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.

6.3.3 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.

6.3.4 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    size;             // 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_interator_next()* operation. *num_entries* specifies how many entries in the returned iterator list(*it_list*). *size* is the buffer(it_list) size and the total amount of data returned in bytes. *it_list* has *num_entries* of iterator elements;

- *num_entries entries of <key_length, key>* when iterator is set with *KVS_ITERATOR _KEY* (**Figure 2**) and *num_entries* entries of *<key_length, key, value_length, value>* when iterator is set with *KVS_ITERATOR _KEY_VALUE (***Figure 3***)*.

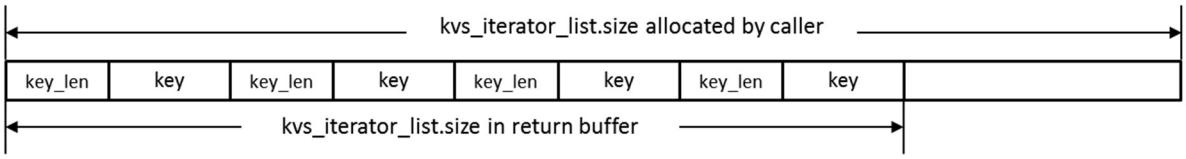

Figure 2. Iterator option: KVS_ITERATOR_KEY

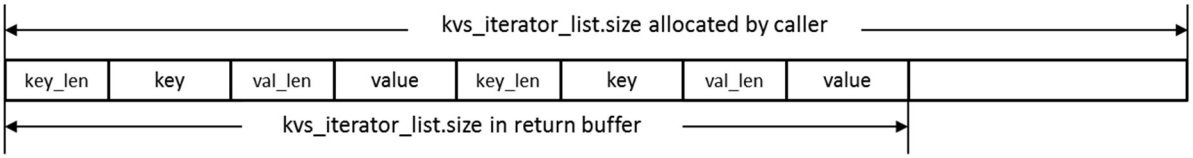

Figure 3. iterator option: KVS_ITERATOR_KEY_VALUE

[SAMSUNG] buffer (it_list) size must be 32KB.

6.3.6 kvs_iterator_info

```
typedef struct {
    uint8_t iter_handle;        // iterator handle
    uint8_t status;             // 1 (The handle is opened), 0 (The handle is closed)
    uint8_t type;               // Type of iterator
    uint8_t keyspace_id;        // KSID that the iterate handle deals with
    uint32_t bit_pattern;       // bit pattern for condition
    uint32_t bitmask;           // bit mask for bit pattern to use
    uint8_t is_eof;             // 1 (The iterate is finished), 0 (The iterate is not finished)
    uint8_t reserved[3];
} kvs_iterator_info;
```

This data structure contains iterator metadata associated with an iterator.

6.3.7 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.

6.3.8 kvs_container

```
typedef struct {
    bool opened;                    // is this container opened
    uint64_t capacity;              // container capacity in byte 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.

6.3.9 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.

6.3.10 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.

6.3.11 kvs_value

```
typedef struct {
   void *value;                 // start address of buffer for value byte stream
   uint32_t length;             // the buffer length in bytes for value byte stream for input and the returned length for output
   uint32_t actual_value_size;  // actual value size in bytes that is stored in a device
   uint32_t offset;             // [OPTION] offset to indicate the offset of value stored in device
} kvs_value;
```

A value consists of a void pointer, a length, actual_value_size and offset. The *value* pointer refers to a byte string without null termination. The *value* pointer variable cannot be a null pointer. The *length* variable holds the byte count. The *length* indicates value buffer size for an input and the returned value size for an output. *actual_value_size* contains the actual value size that is stored in a device. *offset* specifies the offset within a value stored in the device.

[SAMSUNG] The valid value size ranges between 0B and 2MB.

[SAMSUNG] Samsung Key-value SSD **does not** supports a partial retrieval and partial store of a tuple based on an offset.

[SAMSUNG] buffer (value) size must be aligned to four bytes for kvs_retrieve_tuple() and kvs_retrieve_tuple_asyn() APIs.

6.3.12 kvs_tuple_info

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

This data structure contains tuple metadata associated with a key.

6.3.13 kvs_delete_option

```
typedef struct {
    bool kvs_delete_error;          // [OPTION] delete a tuple if a key exists otherwise it returns
} kvs_delete_option;                KVS_ERR_KEY_NOT_EXIST.
```

A user can specify a *delete* operation option.

- *kvs_delete_error set to FALSE* specifies that the operation deletes a key value tuple and always returns KVS_SUCCESS even though a key does not exists. *kvs_delete_error* set to TRUE specifies that the operation deletes a key value tuple if the key exists. It returns KVS_ERR_KEY_NOT_EXIST error when a key does not exist.

6.3.14 kvs_retrieve_option

```
typedef struct {
    bool kvs_retrieve_decompress;   // [OPTION] decompress value before retrieval from storage device if
                                    device has compress/decompress capability
    bool kvs_retrieve_delete;       // [OPTION] read the value of tuple and atomically delete the tuple
} kvs_retrieve_option;
```

A user can specify a *retrieve* operation option.

- *kvs_retrieve_decompress* set to TRUE specifies that an operation reads the key-value tuple and if a device has compress/decompress capability, the device first decompress the data and then returns the decompressed data. *kvs_retrieve_decompress* set to *FALSE* specifies that an operation reads the key-value tuple and returns the data without performing any decompression.
- *kvs_retrieve_delete* set to TRUE specifies that an operation reads the key-value tuple and the key value tuple is atomically deleted after completing the read. *kvs_retrieve_delete* set to FALSE specifies that an operation reads the key-value tuple and no deletion is atomically performed

  [SAMSUNG] The Samsung Key-value SSD supports the kvs_retrieve_delete=FALSE and kvs_retrieve_decompress=FALSE option only.

6.3.15 kvs_store_option

```
typedef struct {
    kvs_store_type      st_type;        // store option type (refer to 6.2.3)
    bool kvs_store_compress;            // [OPTION] compress data before storing to media if a device has
} kvs_store_option;                     compression/decompression capability
```

A user can define a *store* operation option.

- kvs_store_compress set to TRUE specifies that if a device has compress/decompress capability, the device first compress the data and then stores the data on media. kvs_store_compress set to FALSE specifies that an operation stores the key-value tuple without performing any compression

[SAMSUNG] The Samsung Key-value SSD supports the KVS_STORE_POST and KVS_STORE_NOOVERWRITE (=idempotent) only in st_type.

[SAMSUNG] The Samsung Key-value SSD supports the kvs_store_compress=FALSE only.

6.3.16 kvs_iterator_option

```
typedef struct {
    kvs_iterator_type    iter_type;                    // iterator type (refer to 6.2.1)
} kvs_iterator_option;
```

[SAMSUNG] The Samsung Key-value SSD currently supports only KVS_ITERATOR_KEY option only. When KVS_ITERATOR_KEY is set, multiple keys are retrieved for every kvs_iterator_next() API call.

6.3.17 kvs_container_option

```
typedef struct {
    kvs_key_order    ordering;                        // key ordering option (refer to    6.2.2)
} kvs_container_option;
```

A user can define a *container* operation option.

[SAMSUNG] The Samsung Key-value SSD supports no key order option (KVS_KEY_ORDER_NONE) only.

6.3.18 kvs_iterator_context

```
typedef struct {
    kvs_iterator_option option;          // 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
    void *private1;                      // data pointer that is used by a user
    void *private2;                      // data pointer that is used by a user
} kvs_iterator_context;
```

This data structure contains delete operation context. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.19 kvs_delete_context

```
typedef struct {
    kvs_delete_option option;            // kvs_delete_option
    void *private1;                      // data pointer that is used by a user
    void *private2;                      // data pointer that is used by a user
} kvs_delete_context;
```

This data structure contains delete operation context. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.20 kvs_exist_context

```
typedef struct {
    void *private1;                 // data pointer that is used by a user
    void *private2;                 // data pointer that is used by a user
} kvs_exist_context;
```

6.3.21 kvs_store_context

```
typedef struct {
    kvs_store_option option;        // kvs_store_option
    void *private1;                 // data pointer that is used by a user
    void *private2;                 // data pointer that is used by a user
} kvs_store_context;
```

This data structure contains store operation context. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.22 kvs_retrieve_context

```
typedef struct {
    kvs_retrieve_option option;     // kvs_retrieve_option
    void *private1;                 // data pointer that is used by a user
    void *private2;                 // data pointer that is used by a user
} kvs_retrieve_context;
```

This data structure contains retrieve operation context. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.23 kvs_container_context

```
typedef struct {
    kvs_container_option option;    // kvs_container_option
    void *private1;                 // data pointer that is used by a user
    void *private2;                 // data pointer that is used by a user
} kvs_container_context;
```

This data structure contains container operation context. Code must not rely on the size of this data structure since the size can increase in the future as more features are added.

6.3.24 kvs_callback_context

```
typedef struct {
    uint8_t    opcode;                  // operation opcode
    kvs_container_handle *cont_hd;      // container handle
    kvs_key *key;                       // key data structure
    kvs_value *value;                   // value data structure
    uint32_t key_cnt;                   // key count for kvs_exist() call
    uint8_t *result_buffer;             // a pointer to the result buffer for kvs_exist() call
    void *private1;                     // a pointer passed from a user
    void *private2                      // a pointer passed from a user
    kvs_result result;                  // IO result
    kvs_iterator_handle*iter_hd;        // iterator handle
} kvs_callback_context;
```

kvs_callback_context is IO context that carries IO information including key and value tuples and operation return value. It is mainly used for call back function for async operations. The kvs_callback_context must be valid until the callback function completes the operation. Therefore it must be allocated in heap not in stack.

6.3.25 kvs_callback_function

```
typedef void (*kvs_callback_function)(kvs_callback_context* cbctx);        // callback function pointer
```

This is a post processing (callback) function pointer for async operations.

7 KEY VALUE SSD APIS

7.1 Device APIs

7.1.1 kvs_init_env_opts

kvs_result kvs_init_env_opts(kvs_init_options* options)

This API sets default values to kvs_init_options.

PARAMETERS

IN kvs_init_options options for the library

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

7.1.2 kvs_init_env

kvs_result kvs_init_env(kvs_init_options* options)

This API must be called once to initialize the environment for the library.

PARAMETERS

IN kvs_init_options options for the library

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_OPTION_INVALID the device does not support the specified options.

7.1.3 kvs_open_device

kvs_result 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 meaning that dev_path is incorrect.
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_PARAM_INVALID	*dev_path* cannot be NULL
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.4 kvs_close_device

kvs_result kvs_close_device (kvs_device_handle dev_hd)

This API closes a KVS device. *dev_hd* 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 if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_PERMISSION	a caller does not have root permission

7.1.5 kvs_get_device_info

kvs_result 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_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.6 kvs_get_device_capacity

kvs_result kvs_get_device_capacity(kvs_device_handle dev_hd, int64_t *dev_capa)

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
OUT dev_capa	device capacity in bytes

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.7 kvs_get_device_utilization

kvs_result kvs_get_device_utilization(kvs_device_handle dev_hd, int32_t *dev_util)

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
OUT dev_util	used ratio of the device (0~10000)

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.8 kvs_get_device_waf

kvs_result kvs_get_device_waf(kvs_device_handle dev_hd, float *waf)

This interface returns the device write amplification factor by the given device identifier. The device waf is calculated as follows.

Waf = (total number of bytes writte to NAND flash)/(total number of bytes written by host to SSD)

PARAMETERS

IN dev_hd	kvs_device_handle data structure that includes unique device id
OUT waf	device write amplification factor

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.9 kvs_get_min_key_length

kvs_result kvs_get_min_key_length (kvs_device_handle dev_hd, int32_t *min_key_length)

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
OUT min_key_length	the minimum length of keys that the device supports.

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.10 kvs_get_max_key_length

kvs_result kvs_get_max_key_length (kvs_device_handle dev_hd, int32_t *max_key_length)

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
OUT max_key_length	The maximum length of keys that the device supports.

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.11 kvs_get_min_value_length

kvs_result kvs_get_min_value_length (kvs_device_handle dev_hd, int32_t *min_value_length)

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
OUT min_value_length	the minimum length of value that the device supports.

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.12 kvs_get_max_value_length

kvs_result kvs_get_max_value_length (kvs_device_handle dev_hd, int32_t *max_value_length)

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
OUT max_value_length	the maximum length of value that the device supports.

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.1.13 kvs_get_optimal_value_length

kvs_result kvs_get_optimal_value_length (kvs_device_handle dev_hd, int32_t *opt_value_length)

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

PARAMETERS

IN dev_hd	kvs_device_handle data structure that includes unique device id
OUT opt_value_length	the optimal length of value that the device supports.

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.2 Container APIs

7.2.1 kvs_create_container

kvs_result kvs_create_container (kvs_device_handle dev_hd, const char *name, uint64_t size, const kvs_container_context *ctx)

This API creates a new contrainer 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 byte 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_KEY_ORDER_NONE**, no group order is defined.
- **KVS_KEY_ORDER_ASCENDING**, tuples are sorted in ascending key order in the container. .
- **KVS_KEY_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 size	capacity of a container with respect to tuple size (key size + value size) in byte units
IN ctx	container context (i.e., key ordering option in a container)

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

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 exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_PARAM_INVALID	*name* or *ctx* is NULL
KVS_ERR_OPTION_INVALID	option is invalid

7.2.2 kvs_delete_container

kvs_result 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 if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given *cont_path* does not exist
KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed

7.2.3 kvs_list_containers

kvs_result 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
IOUT	names	buffer to store container names. This buffer must be preallocated before calling this routine.
OUT	cont_cnt	the number of *kvs_container_names* stored in the buffer

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_INDEX	*index* is not valid
KVS_ERR_PARAM_INVALID	*name_ids* or *conts_cnt* is NULL

7.2.4 kvs_open_container

kvs_result 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	name	container name
OUT	cont_hd	container handle

RETURNS

KVS_SUCCESS if it is successful, otherwise it returns error.

ERROR CODE

KVS_ERR_DEV_NOT_EXIST	no device exists for the *dev_hd*
KVS_ERR_CONT_NOT_EXIST	Container with the given container *name* does not exist,
KVS_ERR_SYS_IO	Communication with device failed
KVS_ERR_CONT_OPEN	container has been opened already

7.2.5 kvs_close_container

kvs_result 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 indict that closing a container is successful.

ERROR CODE

KVS_ERR_CONT_CLOSE	cannot close the container
KVS_ERR_CONT_NOT_EXIST	container with a given *cont* does not exist
KVS_ERR_SYS_IO	communication with device failed

7.2.6 kvs_get_container_info

kvs_result 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 indict that getting container info is successful.

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_PARAM_INVALID	*cont* is NULL

7.3 Key-value tuple APIs

7.3.1 kvs_get_tuple_info

kvs_result 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 6.3.12 *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 *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_ PARAM_INVALID	*key* or *info* is NULL
KVS_ERR_KEY_NOT_EXIST	*key* does not exist

7.3.2 kvs_retrieve_tuple

kvs_result 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.length* contains the buffer size. 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 offset must align to KVS_ALIGNMENT_UNIT. The tuple value is copied to *value.value* buffer and the data size copied to the output buffer is set to *value.length*. If an allocated value buffer is not big enough to hold the whole value, it will set *value.length* by the the returned data length and return KVS_ERR_BUFFER_SMALL. *value.actual_value_size* is set to the actual size of value that is stored inside a device.

A user can define a retrieve operation option as defined in section 6.3.14:

- *kvs_retrieve_decompress* set to TRUE specifies that an operation reads the key-value tuple and if a device has compress/decompress capability, the device first decompress the data and then returns the decompressed data. *kvs_retrieve_decompress* set to *FALSE* specifies that an operation reads the key-value tuple and returns the data without performing any decompression.
- *kvs_retrieve_delete* set to TRUE specifies that an operation reads the key-value tuple and the key value tuple is atomically deleted after completing the read. *kvs_retrieve_delete* set to FALSE specifies that an operation reads the key-value tuple and no deletion is atomically performed

[SAMSUNG] The Samsung Key-value SSD supports the kvs_retrieve_delete=FALSE and kvs_retrieve_decompress=FALSE option only.
[SAMSUNG] buffer (value.value) size in kvs_value must be aligned to four bytes.

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	*kvs_value.offset* is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_BUFFER_SMALL	buffer space of *value* is not allocated or not enough
KVS_ERR_ PARAM_INVALID	*key* or *value* is NULL
KVS_ERR_OPTION_INVALID	the option in the *ctx* is not supported
KVS_ERR_KEY_NOT_EXIST	key does not exist

7.3.3 kvs_retrieve_tuple_async

kvs_result kvs_retrieve_tuple_async (kvs_container_handle cont_hd, const kvs_key *key, kvs_value *value, const kvs_retrieve_context *ctx, kvs_callback_function cbfn)

This API asynchronously retrieves a key value tuple value with the given key and returns immediately regardless of whether the tuple is actually retrieved from a device or not. The final execution results are returned to callback function through *kvs_callback_context*. The *value* parameter contains output buffer information for the value. *value.value* contains the buffer to store the tuple value and *value.length* contains the buffer size. 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 offset must align to KVS_ALIGNMENT_UNIT. The tuple value is copied to *value.value* buffer and the data size copied to the output buffer is set to *value.length*. If an allocated value buffer is not big enough to hold the whole value, it will set *value.length* by the the returned data length and return KVS_ERR_BUFFER_SMALL. *value.actual_value_size* is set to the actual size of value that is stored inside a device.

A user can define a retrieve operation option as defined in section 6.3.14:

- *kvs_retrieve_decompress* set to TRUE specifies that an operation reads the key-value tuple and if a device has compress/decompress capability, the device first decompress the data and then returns the decompressed data. *kvs_retrieve_decompress* set to *FALSE* specifies that an operation reads the key-value tuple and returns the data without performing any decompression.
- *kvs_retrieve_delete* set to TRUE specifies that an operation reads the key-value tuple and the key value tuple is atomically deleted after completing the read. *kvs_retrieve_delete* set to FALSE specifies that an operation reads the key-value tuple and no deletion is atomically performed

[SAMSUNG] The Samsung Key-value SSD supports the kvs_retrieve_delete=FALSE and kvs_retrieve_decompress=FALSE option only.
[SAMSUNG] buffer (value.value) size in kvs_value must be aligned to four bytes.

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.
IN cbfn	callback function

RETURNS

KVS_SUCCESS	indict that the retrieval operation is successful.

ERROR CODE

KVS_ERR_VALUE_OFFSET_INVALID	*kvs_value.offset* is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_BUFFER_SMALL	buffer space of *value* is not allocated or not enough
KVS_ERR_ PARAM_INVALID	*key* or *value* is NULL
KVS_ERR_OPTION_INVALID	the option in the *ctx* is not supported
KVS_ERR_KEY_NOT_EXIST	key does not exist

7.3.4 kvs_store_tuple

kvs_result 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. A user can define a store operation option and type.

Store option:

- ***kvs_store_compress*** set to TRUE specifies that if a device has compress/decompress capability, the device first compress the data and then stores the data on media. kvs_store_compress set to FALSE specifies that an operation stores the key-value tuple without performing any compression

Store type:

- ***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_VALUE_OFFSET_INVALID. If a key exists, a new value replaces the current tuple value. This is similar to a database *update*. 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_VALUE_OFFSET_INVALID. The *value* size of the new tuple is equal to the sum of *offset* and *value.size*. The offset must align to KVS_ALIGNMENT_UNIT. This is the default store operation behavior.
- ***KVS_STORE_UPDATE_ONLY***: if the key exist, the operation overwrites value. If the key does not exist, it returns KVS_KEY_NOT_EXIST error.
- ***KVS_STORE_NOOVERWRITE*** 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.
- ***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_CONT_CAPACITY error message is returned.

[SAMSUNG] The Samsung Key-value SSD supports the KVS_STORE_POST and KVS_STORE_NOOVERWRITE (=idempotent) only in st_type option with the *offset* of value equal to zero.

[SAMSUNG] The Samsung Key-value SSD supports the kvs_store_compress=FALSE only.

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	*kvs_value.offset* is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_ PARAM_INVALID	a *key* or a *value* is NULL
KVS_ERR_VALUE_OFFSET_INVALID	*kvs_value.offset* is invalid
KVE_ERR_OPTION_INVALID	unsupported option is specified in *ctx*
KVS_ERR_CONT_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)

7.3.5 kvs_store_tuple_async

kvs_result kvs_store_tuple_async (kvs_container_handle cont_hd, const kvs_key *key, const kvs_value *value, const kvs_store_context *ctx, kvs_callback_function cbfn)

This API asynchronously writes a key-value tuple into a container and returns immediately regardless of whether the tuple is actually written to a device or not. The final execution results are returned to callback function through kvs_callback_context. A user can define a store operation option and type.

Store option:

- ***kvs_store_compress*** set to TRUE specifies that if a device has compress/decompress capability, the device first compress the data and then stores the data on media. kvs_store_compress set to FALSE specifies that an operation stores the key-value tuple without performing any compression

Store type:

- ***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_VALUE_OFFSET_INVALID. If a key exists, a new value replaces the current tuple value. This is similar to a database *update*. 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_VALUE_OFFSET_INVALID. The *value* size of the new tuple is equal to the sum of *offset* and *value.size*. The offset must align to KVS_ALIGNMENT_UNIT. This is the default store operation behavior.
- ***KVS_STORE_UPDATE_ONLY***: if the key exist, the operation overwrites value. If the key does not exist, it returns KVS_KEY_NOT_EXIST error.
- ***KVS_STORE_NOOVERWRITE*** 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.
- ***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_CONT_CAPACITY error message is returned.

[SAMSUNG] The Samsung Key-value SSD supports the KVS_STORE_POST and KVS_STORE_NOOVERWRITE (=idempotent) only in st_type option with the *offset* of value equal to zero.

[SAMSUNG] The Samsung Key-value SSD supports the kvs_store_compress=FALSE only.

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.
IN cbfn	callback function

RETURNS

KVS_SUCCESS	indict that writing a tuple is successful.

ERROR CODE

KVS_ERR_OFFSET_INVALID	*kvs_value.offset* is not aligned to KVS_ALIGNMENT_UNIT
KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_ PARAM_INVALID	a *key* or a *value* is NULL
KVS_ERR_VALUE_OFFSET_INVALID	*kvs_value.offset* is invalid
KVE_ERR_OPTION_INVALID	unsupported option is specified in *ctx*
KVS_ERR_CONT_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)

7.3.6 kvs_delete_tuple

kvs_result 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_error set to FALSE* specifies that the operation deletes a key value tuple and always returns KVS_SUCCESS even though a key does not exists. *kvs_delete_error* set to TRUE specifies that the operation deletes a key value tuple if the key exists. It returns KVS_ERR_KEY_NOT_EXIST error when a key does not exist.
-

[SAMSUNG] *kvs_delete_error* set to FALSE would have better performance.

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 *cont_hd* does not exist
KVS_ERR_ PARAM_INVALID	*key* is NULL
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_OPTION_INVALID	option in *ctx* is not supported
KVS_ERR_KEY_NOT_EXIST	*key* does not exist

7.3.7 kvs_delete_tuple_async

kvs_result kvs_delete_tuple_async (kvs_container_handle cont_hd, const kvs_key* key, const kvs_delete_context* ctx, kvs_callback_function cbfn)

This API asynchronously deletes key value tuple with a given key and returns immediately regardless of whether the tuple is actually deleted from a device or not. The final execution results are returned to callback function through kvs_callback_context.

A user can define a delete operation option in *kvs_delete_option*.

- *kvs_delete_error set to FALSE* specifies that the operation deletes a key value tuple and always returns KVS_SUCCESS even though a key does not exists. *kvs_delete_error* set to TRUE specifies that the operation deletes a key value tuple if the key exists. It returns KVS_ERR_KEY_NOT_EXIST error when a key does not exist.

[SAMSUNG] *kvs_delete_error* set to FALSE would have better performance.

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.
IN cbfn	callback function

RETURNS

KVS_SUCCESS	indicate that dropping is successful

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	container with a given *cont_hd* does not exist
KVS_ERR_ PARAM_INVALID	*key* is NULL
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_KEY_LENGTH_INVALID	given *key* is not supported (e.g., length)
KVS_ERR_OPTION_INVALID	option in *ctx* is not supported
KVS_ERR_KEY_NOT_EXIST	*key* does not exist

7.3.8 kvs_exist_tuples

kvs_result kvs_exist_tuples (kvs_container_handle cont_hd, uint32_t key_cnt, const kvs_key *keys, uint32_t buffer_size, uint8 *result_buffer, const kvs_exist_context *ctx)

This API checks if a set of one or more keys exists and returns a *bool type* status. The existence of a key value tuple 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.

[SAMSUNG] The Samsung device supports only one key existence test.

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
IN ctx	exist context. It can be NULL. In that case, the default drop context will be used. It is vendor specific.
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 *cont_hd* does not exist
KVS_ERR_BUFFER_SMALL	the buffer space of *results* is not big enough
KVS_ERR_ PARAM_INVALID	*keys* or *results* parameter is NULL
KVS_ERR_SYS_IO	communication with device failed

7.3.9 kvs_exist_tuples_async

kvs_result kvs_exist_tuples_async (kvs_container_handle cont_hd, uint32_t key_cnt, const kvs_key *keys, uint32_t buffer_size, uint8 *result_buffer, const kvs_exist_context *ctx, kvs_callback_function cbfn)

This API asynchronously checks if a set of keys exists and returns a *bool type* status. It returns immediately regardless of whether keys are checked from a device or not. The final execution results are returned to the callback function through kvs_callback_context. The existence of a key value tuple 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.

[SAMSUNG] The Samsung device supports only one key existence test.

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
IN ctx	exist context. It can be NULL. In that case, the default drop context will be used. It is vendor specific.
IN cbfn	callback function
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 *cont_hd* does not exist
KVS_ERR_BUFFER_SMALL	the buffer space of *results* is not big enough
KVS_ERR_ PARAM_INVALID	*keys* or *results* parameter is NULL
KVS_ERR_SYS_IO	communication with device failed

7.4 Iterator APIs

7.4.1 kvs_open_iterator

kvs_result 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_patern 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 set to 1, *kvs_open_iterator()* should be called with *bitmask* = 0x80000000 (1000 0000 0000 0000 0000 0000 0000 0000) and *bit_pattern* = 0x80000000 (1000 0000 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 set to 0, *bitmask* should be 0x80000000 (1000 0000 0000 0000 0000 0000 0000 0000) and *bit_pattern* should be 0x0 (0000 0000 0000 0000 0000 0000 0000 0000).
3) If applications want to get all the existing keys with the first two bytes (bit 0 ~ bit15) equal to 0x0004, *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 0 ~ bit 4 equal to (10101), *bitmask* should be 0xF8000000 (1111 1000 0000 0000 0000 0000 0000 0000) and *bit_pattern* should be 0xA8000000 (1010 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_type is *KVS_ITERATOR_KEY while kvs_iterator_next()* will retrieve key and value tuples when the kvs_iterator_type is *KVS_ITERATOR_KEY_VALUE*.

Finally it will return an *iterator identifier*.

[SAMSUNG] *Samsung device support KVS_ITERATOR_KEY iterator type only. Samsung devices does not allow leading zeros in bitmask. For example, bitmask of 0xFF000000 is allwed but 0x0FF00000 is not allowed.*

PARAMETERS

IN	cont_hd	kvs_container_handle data structure that includes unique container id
IN	ctx	iterator context (refer to the section 6.3.18)
OUT	iter_hd	iterator handle

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_SYS_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
KVS_ERR_ITERATOR_MAX	**The maximum number of iterators are already open**

7.4.2 kvs_close_iterator

kvs_result 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

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_ITERATOR_NOT_EXIST	the iterator Key Group does not exist

7.4.3 kvs_close_iterator_all

kvs_result kvs_close_iterator_all(kvs_container_handle cont_hd)

This interface releases all iterators in the given container. So the iterator operation ends.

PARAMETERS

IN cont_hd kvs_container_handle data structure that includes unique container id

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_SYS_IO	communication with device failed

7.4.4 kvs_list_iterators

kvs_result kvs_list_iterators(kvs_container_handle cont_hd, kvs_iterator_info *kvs_iters, uint32_t count)

This interface retrieves a list of iterators in this device and returns them in the given iterator array. The API will return *count* number of iterators, the status of the returned iterators does not have to be 'open'. The caller should create kvs_iterator_info object before calling this API.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN/OUT kvs_iters	kvs_iterator_info array includes a list of kvs_iterator_info
IN count	number of iterate handles to get information

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_SYS_IO	communication with device failed

7.4.5 kvs_iterator_next

kvs_result kvs_iterator_next(kvs_container_handle cont_hd, kvs_iterator_handle iter_hd, kvs_iterator_list *iter_list, const kvs_iterator_context *ctx)

This interface obtains a subset of key or a key-value tuple(s) from a key group of *iter_hd* in a device meaning that *kvs_iterator_next()* retrieves the next key group of keys or a key-value tuple(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->it_list*) size in bytes as an input and the returned list size as an output (refer 6.3.5 kvs_iterator_list). In the output of this operation, iter_list->num_entries provides the number of iterator elements in iter_list->it_list. iter_list->end sets to one if when there is no more iterator group elements meaning that iterator reaches the end. Otherwise iter_list->end sets to zero and there are more iterator key group elements and the host may run *kvs_iterator_next()* again to retrieve those elements. The retrieved values (*iter_list->it_list*) are either keys or key-value tuples based on the iterator option type which is set by *kvs_open_iterator().*

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

- **KVS_ITERATOR _KEY [MANDATORY]**: a subset of keys are returned in *iter_list->it_list* data structure
- **KVS_ITERATOR _KEY_VALUE**; a subset of key-value tuples are returned in *iter_list->it_list* data structure

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.

[SAMSUNG] The Samsung device supports only KVS_ITERATOR_KEY type.
[SAMSUNG] buffer (iter_list->it_list) size in kvs_iterator_list must be 32KB.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN iter_hd	iterator handle
IN iter_size	iterator array (iter_list) buffer size
IN ctx	iterator context
IN/OUT iter_list	iterator list data structure including iterator status and output buffer for a set of keys or key-value tuples

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_PARAM_INVALID	*iter_list* parameter is NULL
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_ITERATOR_NOT_EXIST	the iterator Key Group does not exist

7.4.6 kvs_iterator_next_async

kvs_result kvs_iterator_next_async(kvs_container_handle cont_hd, kvs_iterator_handle iter_hd, kvs_iterator_list *iter_list, const kvs_iterator_context *ctx, kvs_callback_function cbfn)

This interface asynchronously retrieves the next key group of keys or a key-value tuple(s) in the iterator key group (*iter_hd)* that is set with *kvs_open_iterator()* command). It returns immediately regardless of whether the key group is retrieved from a device or not. The final execution results are returned to the callback function through kvs_callback_context. Iter_list->size is the iterator buffer (*iter_list->it_list*) size in bytes as an input and the returned list size as an output (refer 6.3.5 kvs_iterator_list). In the output of this operation, iter_list->num_entries provides the number of iterator elements in iter_list->it_list. iter_list->end sets to one if when there is no more iterator group elements meaning that iterator reaches the end. Otherwise iter_list->end sets to zero and there are more iterator key group elements and the host may run *kvs_iterator_next()* again to retrieve those elements. The retrieved values (*iter_list->it_list*) are either keys or key-value tuples based on the iterator option type which is set by *kvs_open_iterator().*

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

- **KVS_ITERATOR _KEY [MANDATORY]**: a subset of keys are returned in *iter_list->it_list* data structure
- **KVS_ITERATOR _KEY_VALUE**; a subset of key-value tuples are returned in *iter_list->it_list* data structure

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.

[SAMSUNG] The Samsung device supports only KVS_ITERATOR_KEY type.
[SAMSUNG] buffer (iter_list->it_list) size in kvs_iterator_list must be 32KB.

PARAMETERS

IN cont_hd	kvs_container_handle data structure that includes unique container id
IN iter_hd	iterator handle
IN iter_size	iterator array (iter_list) buffer size
IN ctx	iterator context
IN/OUT iter_list	iterator list data structure including iterator status and output buffer for a set of keys or key-value tuples

RETURNS

KVS_SUCCESS if it is successful otherwise it returns error code

ERROR CODE

KVS_ERR_CONT_NOT_EXIST	no container with *cont_hd* exists
KVS_ERR_PARAM_INVALID	*iter_list* parameter is NULL
KVS_ERR_SYS_IO	communication with device failed
KVS_ERR_ITERATOR_NOT_EXIST	the iterator Key Group does not exist