2015

Virtual Storage 1.0a

DATA STORE MANAGEMENT PLATFORM

Doc Rev 1.01

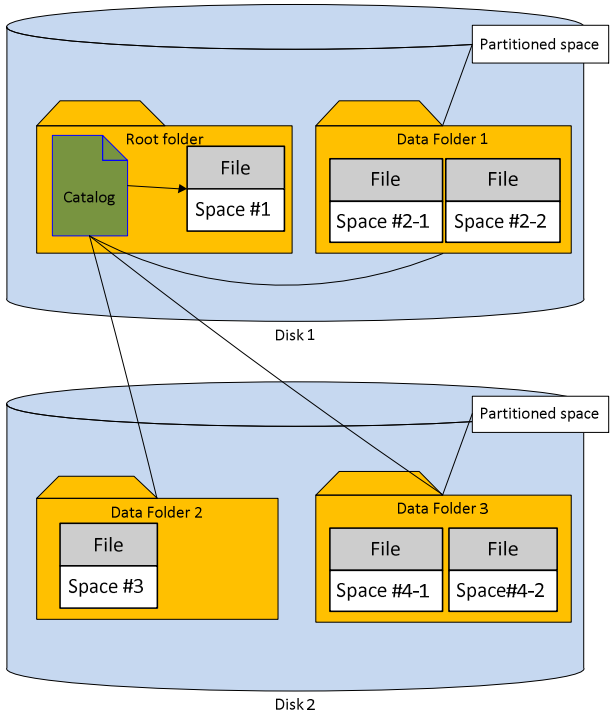
mailto: ivvvsx@gmail.com

Virtual Storage is a core platform that provides capabilities to create the customized local application data storage.
It is not a kind of relational or hierarchical database but platform that includes the basic tool to create random or sequential data access mechanism.
The basic principles are:

1. Data storage is a set of random-access address spaces.
2. Each space supports the following functions:
    - Allocate/release space chunks.
    - Direct read/write operations by address or assigned unique keys.
    - Indexing data within space by user-defined keys.
3. All write operations performed within cross-space transactions that guarantees data consistency on the physical level.
4. The basic access method for each space is virtual memory mechanism that uses real-memory cache.
5. All physical input/output operations performed on the page level (page size is defined for each space and may vary within storage).
6. Address space boundaries are limited by physical space file(s) size.
7. Each space can be expanded by adding disk space to the space file or creation of the new partition file(s).
8. Access is available in read-write mode by only one process or many processes in read only mode.

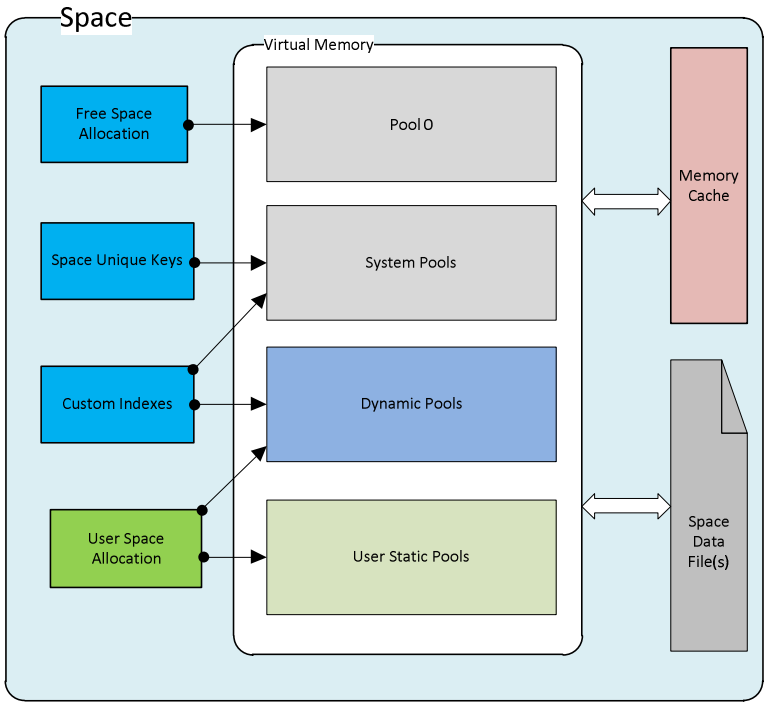
Physical storage organization (generally):

mailto: ivvvsx@gmail.com



Data storage is set of the logical spaces – address ranges where system and user data can be allocated and released.

Logical space organization:



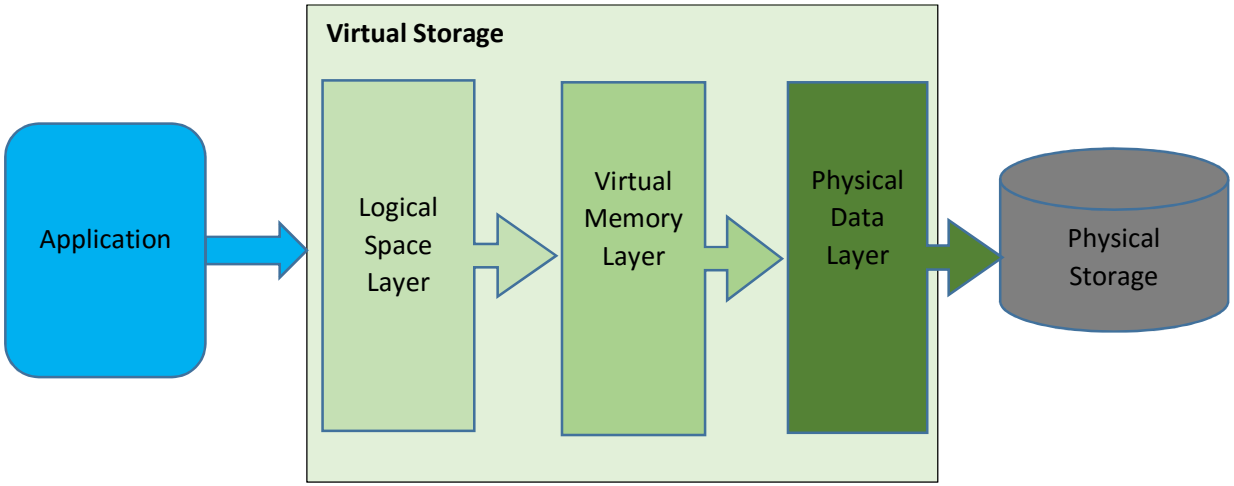
The pool number is always assigned to the space allocation; either predefined (static) pool number or assigned by system (dynamic).

Pool numbers:

- Pool 0 – memory containing apace allocation descriptors (system use only).
- System pools (<0) – memory containing internal system objects and structures (system use only).
- User Static Pools (1-4095) – user-defined pool number; it must be specified in the allocation method.
- Dynamic Pools (>4095) – pool number assigned automatically by system; can be used by both user application and system; using dynamic pools is reasonable if it is supposed to release the whole pool including all objects by the single method call.

II. Architecture and Design

Virtual Storage implemented on the basis of a multi-layer architecture and Q&C performance improvement technology:



1. **Application**
   Client application that uses Virtual Storage API (VStorage.dll).

2. **Virtual Storage**

2.1 Layers

Virtual Storage is based on 3 layers.

2.1.1 Physical Data Layer

- Physical files and folders management:
- Read/Write operations for space files.
- Read/Write operations for storage catalog file.
- Read/Write operations for transaction log.
- Create/Delete space files.
- Create space files partitions.
- Extend space files (by stand-alone utility and on-the-fly).

2.1.2 Virtual Memory Layer

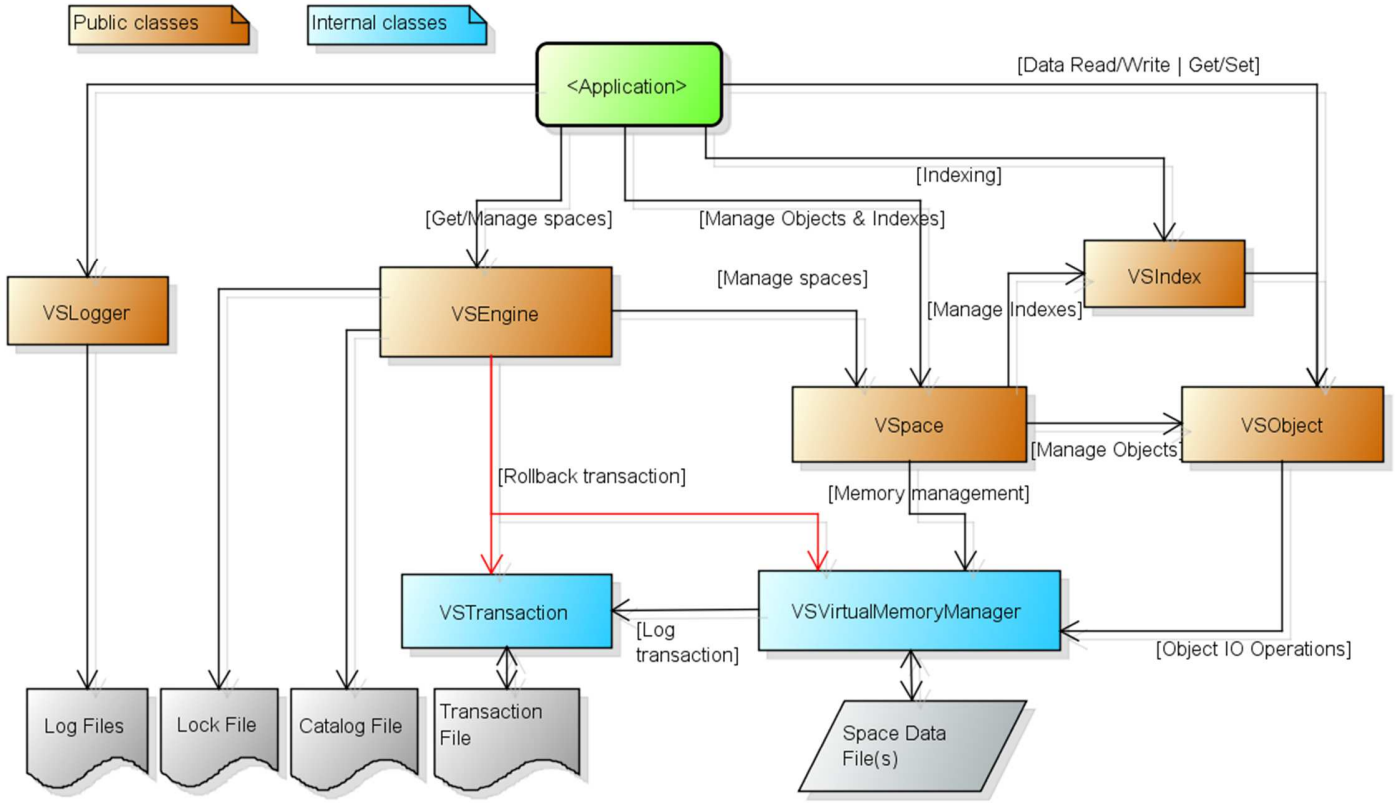
- Virtual memory emulation for space, address space matching physical file(s) size, including partitions.
- Managing real memory pool and page swapping.
- Read/Write operation in the space by virtual address.

2.1.3 Logical Space Layer

- Multiple virtual address spaces.
- Allocate virtual memory for objects within space in the memory pools.
- Assign unique 64-bit object identifiers (IDs)
- Release individual object memory allocation or all memory in the pool.
- Read/Write operations within object allocation memory.
- Logical fields Get/Set operations within object allocation memory.
- Unique and non-unique indexing by user-defined key (index is created within space and always contains object ID as reference).
- Data consistency – support cross-space transactions.
- Dump and Restore of the allocated memory capabilities for individual spaces or whole storage.

2.2 Core Object Model

Virtual Storage core classes:



1) **VSEngine**

Virtual Storage engine class, object instance created by user's application.
Storage root folder path passed to constructor as parameter.
Open method opens storage in the requested mode ('open', 'open read', 'create', 'open or create').
Default transaction started automatically when storage opened (if not 'open read' mode).
All existing storage spaces initialized when opening and requested by user application using *GetSpace* method.

The array of space names can be obtain by *GetSpaceList* method when storage opened or by GetSpaceHeaderList method from the catalog entries if storage not opened yet.
Close method closes storage and makes spaces unavailable.
Some VSEngine are methods designated for storage management and used only if storage not opened:

- Create
- Remove
- Extend
- AddPartition
- List
- Dump
- Restore

2) VSpace

VSpace object created by VSEngine for each storage space during storage opening.
VSpace object obtained by VSEngine *GetSpace* method.
The purpose of this object is to manage memory allocation and indexing within space.
Each memory allocation represented by VSObject instance.
Each index represented by VSIndex instance.

Memory management methods:

- Allocate - allocate memory within space (VSObject is returned);
- Release - release object and free allocated memory (VSObject shall be specified);
- ReleaseID - release object and free allocated memory by object ID;
- ReleasePool - release all objects in the pool and free allocated memory;
- GetFreePoolNumber - obtain free dynamic pool number;
- GetObject - get object by assigned unique ID;
- GetRootId - get ID of the first object in the pool;
- GetRootObject - get first object in the pool (represented by VSObject).

Index management methods:

- CreateIndex - create new index;
- DeleteIndex - delete existing index;

- IndexExists - check if index with the specified name already exists;
- GetIndex - get existing index by name (represented by VSIndex).

3) VSObject

Core object for data management that supports all I/O operations within allocated virtual memory.
This object contains descriptor with the system specific information and local address space.
Address is started at 0, the size is equal to requested in *VSpace.Allocate* method.
This object includes two sets of data access methods for typed data:

a) Direct Read/Write by address methods
   - ReadByte
   - ReadBytes
   - ReadShort
   - ReadUShort
   - ReadInt
   - ReadLong
   - ReadDecimal
   - ReadDataTime
   - ReadString
   - Write – methods group (byte, bytes, short, ushort, int, long, decimal, DateTime, string)

b) Get/Set by field name methods
   - GetByte
   - GetBytes
   - GetShort
   - GetInt
   - GetLong
   - GetDecimal
   - GetDataTime
   - GetString
   - Set – methods group (byte, bytes, short, int, long, decimal, DateTime, string)

4) VSIndex

Index management object, provides methods for search, create and delete custom indexes for objects inside space.
Index can be unique or non-unique. Each index element consists of index key and one or more (for non-unique indexes) reference values. Each reference is 64-bit key of the object within space.
VSpace object provide methods for index management (create index, delete index, get index by name).
VSIndex general methods:

- Delete
- Exists
- Find
- FindAll
- Insert

VSIndex supports key enumeration and correspondent methods:

- Reset
- Next

Enumeration properties for the current index:

- CurrentKey
- CurrentRefs

5) **VSTransaction**

VSTransaction is internal class that is responsible for cross-space page-level transactions and supports:

- Create transaction log record in 'write' operations;
- Read transaction log record in the 'rollback' operation.

6) **VSVirtualMemoryManager**

VSVirtualMemoryManager is internal class that supports virtual memory mechanism for each storage space:

- Manage memory cache for space virtual pages;
- Fetch/flush physical pages from/to the space data file(s);
- Direct I/O operations by virtual address;
- Auto-extension of the space (if allowed by creation options).

7) **VSLogger**

VSLogger is supplementary class that allows user application to log custom events.
Log files do not use virtual memory mechanism and transactions.
They are store in the separate folder inside the storage root folder.
The log consists of two physical files – data and index ('vdat' and 'vidx' file types respectively).

The following functions supported:

- Write log records sequentially;
- Read log records sequentially;
- Read log record by record number;
- Read the range of log records by the first record number;
- Delete log files;
- Purge log files (delete all records but keep files);
- Archive log files (move log files content to archive subfolder).

3. Physical File Storage

Physical storage contains all necessary physical data (files and folders):

a) Storage Root Folder

This folder contains at least storage catalog file. It must be specified as an entry point for VStorage component.
All other files can be located in this folder as well as in any other folder that has reference in the catalog.
Root Folder must exist when running VStorage.

b) Storage Catalog File

This file contains references to Storage Data Files and their properties.
Any data file created, deleted or modified only through the catalog entries.
The catalog entry creates association between the physical file(s) and storage logical space.
The name of catalog file is always *vsto0001.0000.vctl* (system-generated).
If the catalog does not exist when you start VStorage, a new, empty catalog created.
Every time this file updated, the backup copy created with the name *vsto0001.0000.vctl.bak*.

c) Space Data Files

Logical space data files are located in the Storage Root Folder (by default) or any other user-specified location (reference will be added to the Storage Catalog File during creation).
Each space can contain one or more partition files.
The name pattern for Space Data File is *vsto0001.<space-name>.<nnnnn>.vspc,* where:

- <space-name> - the space name (specified during creation);
- <nnnnn> - partition number (e.g. 00000, 00001, …)

d) Transaction File
This file used to support data consistency in case of the application crash and manual rolling back storage to the previous state.
The Transaction File name is *vsto0001.0000.vsta*.
This file created automatically.

e) Lock File
This file used to prevent data damage when running several VStorage processes on the same storage simultaneously.
The Lock File name is *vsto0001.0000.vlck*.
This file created automatically.

III. API

The **VStorage.dll** is virtual storage core component that provides such capabilities as:

- Virtual memory read/write operations
- Physical storage management (creation, extension, removal, partitioning)
- Space management (allocation, extension, releasing)
- Cross-space transaction management
- Reserved copies management (dump, restore), single-space and multi-space
- Unique keys assigned to the allocated areas
- Complimentary generic indexing function
- Complimentary logging function

Programming language – C#

Framework - .NET Framework 4.5

1. Space Allocation Pools

The virtual memory objects in each space allocated in the queue by the pool number (static or dynamic).
All objects in the pool could be freed individually (by VSpace.Release method) or all at once (by VSpace.FreePool method).
All objects in the same pool has next-previous relationship.
Pools number specified directly in the VSpace.Create method.
Pool number can be:

- Static (assigned by user). The range is 1 – 4095. Assuming user associates the specific data type with the pool number.
- Dynamic (assigned by system). The range 4096 - 32767. Dynamic pool number must be acquired by the VSpace.GetFreePoolNumber() method (unallocated pool number will be provided).

2. Public Constants

2.1 public static class DEFS

This class defines some public constants that can be useful in the application using VStorage API.

Constant	Description
public const string APP_ROOT_DATA	Directory name where all default application data stored, relative to the user's system directory (e.g. 'C:\Users\Iam\AppData\Roaming\IVVS').
public const string DELIM_NEWLINE	New line chars: "\r\n".
public const string KEY_DIRECTORY	Directory name where all default application keys are stored "vs.default", relative to APP_ROOT_DATA.
public const string KEY_DUMP_RESTORE	The key file name where the default path to dump directory is stored: "dump-path.vs.default".
public const string KEY_STORAGE_ROOT	The key file name where the default path to root storage directory is stored: "root-path.vs.default".
public const int MODE_OPEN_READ	Storage open mode read-only (storage must exist): 0.
public const int MODE_OPEN	Storage open mode read-write (storage must exist) 1.
public const int MODE_CREATE	Create new storage (if storage already exists – this will cause error): 2.
public const int MODE_OPEN_OR_CREATE	Create new storage or open storage in read-write mode if exists: 3.
public static string POOL_MNEM(short n)	Pool mnemonic by number
public const string SYSTEM_OWNER_UNDEFINED = "$UNDEFINED$"	Space owner is not defined (assigned when space is created).

3. Classes

3.1 public class VSEngine

3.1.1 Properties

N/A

3.1.2 Constructors

Constructor	Parameter(s)	Description
public VSEngine(string path)	**path** - path to the storage root directory, must exist if specified . Default value is "" (current directory).	If not specified then current application directory is used.

3.1.3 Methods

3.1.3.1 Basic Methods

Name	Parameter(s)	Description
public void Begin()		Begin new transaction. Close storage. Ignored if transaction is already started.
public void Close()		Close storage. Ignored if storage is not opened.
public void Commit()		Commit transaction. Ignored if transaction is not started.
public string[] GetFreeSpaceInfo(string name)	**name** – space name	Return array of strings with the free space areas description.
public VSpace GetSpace(string name)	**name** – space name	Returns VSpace object by name, null if space is not found
public string[] GetSpaceList()		Return array of strings with the space names.
public string Open(int mode, bool exception)	**mode** – open mode, one of the following: DEFS.MODE_OPEN_READ DEFS.MODE_OPEN DEFS.MODE_CREATE DEFS.MODE_OPEN_OR_CREATE (default) **exception** – true: rise exception if error occurred; false: no.	Open storage.
public void RollBack()		Rollback current transaction. New transaction will not start automatically – use 'Begin'.

3.1.3.2 Storage Management Methods

Note: these methods used only if storage is not opened.

Name	Parameter(s)	Description
public void AddPartition(string name, int size)	**name** – space name **size** - partition size (Mb)	Add new space partition.
public void Create(string name, int pagesize, int size, int extension, string path)	**name** – space name **pagesize** - page size(Kb), by default - 16 **size** - space size (Mb) **extension** - space extension (Mb), by default - 0 **path** – path, by default - storage location	Create new space.

public string Dump(string path, string name)	**path** – path to dump file(s) location , by default – current application directory. **name** – space name, '*' – all spaces in the storage (default). Any name pattern can be used including '*' and '?' symbols.	Create reserved copy of the storage space(s). Only allocated space will be stored. Returns empty string if successful, otherwise – error message.
public bool Exists(string name)	**name** – space name (mandatory, no wildcards).	Returns true if space already exista in the storage catalog.
public void Extend(string name, int size)	**name** – space name **size** - additional size (Mb)	Extend existing space if only one partition exists. Otherwise 'AddPartition' method will be invoked.
public string[] GetSpaceNameList()		Return the array of strings with all space names in the storage.
public long GetStorageSize()		Returns storage size in bytes (all spaces).
public string[] List(string name)	**name** – space name, '*' – all spaces in the storage (default). Any name pattern can be used including '*' and '?' symbols.	Return array of strings with the space description.
public void Remove(string name)	**name** – space name	Remove space from storage (including physical file(s).
public string Restore(string path, string name)	**path** – path to dump file(s) location , by default – current application directory. **name** – space name, '*' – all spaces in the storage (default). Any name pattern can be used including '*' and '?' symbols.	Restore all spaces matching 'name' criteria from the reserved copy created by 'Dump' command. Returns empty string if successful, otherwise – error message.

3.2 public class VSException:Exception

Inherited from: Exception

3.2.1 Constructors

Constructor	Parameter(s)	Description
public VSException(int code, string message_ext)	**code** – error code **message_ext** – message to append to the default system message (empty by default).	VStorage exception object. Any exception thrown has its own code and message.

3.2.2 Properties

Property	Description
public int ErrorCode	Error code for exception
public string Message	Full error message for exception

3.2.3 Error codes and messages

Constant	Message

public int E0001_UNABLE_TO_LOCK_CODE = 1	Unable to lock storage
public int E0002_SPACE_NOT_FOUND_CODE = 2	Space is not found
public int E0003_EXTEND_ERROR_CODE = 3	Cannot extend multipartition space
public int E0004_STORAGE_NOT_FOUND_CODE = 4	Storage path is not found
public int E0005_FILE_ALREADY_EXISTS_CODE = 5	Space file already exists
public int E0006_INVALID_SIGNATURE_CODE = 6	Invalid block signature
public int E0007_INVALID_ADDRESS_CODE = 7	Address is out of space boundaries
public int E0008_INVALID_DESCRIPTOR_CODE = 8	Invalid descriptor address
public int E0009_RESTORE_NOT_COMPLETED_CODE = 9	Restore has not been completed for space. Re-creation is required.
public int E0010_READ_ONLY_CODE = 10	Write attempt in Read-Only mode
public int E0011_TRANSACTION_PENDING_CODE = 11	Transaction is pending in read-only mode
public int E0012_INVALID_POOL_NUMBER_CODE = 12	Allocate space invalid pool number
public int E0013_SPACE_NOT_AVAILABLE_CODE = 13	Space is not available for allocation
public int E0014_INVALID_POOL_NUMBER_CODE = 14	Free space pool: invalid pool number
public int E0015_INVALID_ADDRESS_CODE = 15	Free space: invalid address
public int E0016_OPEN_STORAGE_ERROR_CODE = 16	Open storage error
public int E0017_ATTACH_SPACE_ERROR_CODE = 17	Attach space error
public int E0018_TRANSACTION_ERROR_CODE = 18	Transaction error
public int E0019_INVALID_OP_ADDRESS_ERROR_CODE = 19	Invalid relative read/write address error
public int E0020_INVALID_EXTENSION_PARAMETERS_ERROR_CODE = 20	Invalid extension parameters: pool and generateId cannot be specified for extension
public int E0021_MAX_ALLOCATION_CHUNKS_REACHED_CODE = 21	Maximum space allocation chunks number is reached
public int E0022_KEY_NOT_FOUND_CODE = 22	Key is not found (probably DB structure error)
public int E0023_INVALID_KEY_SEQUENCE_CODE = 23	Invalid predefined key sequence
public int E0024_CREATE_SPACE_ERROR_CODE = 24	Create space error
public int E0025_STORAGE_OPENED_CODE = 25	Storage is opened - unable to complete operation
public int E0026_FIELD_READ_ERROR = 26	Object field read error
public int E0027_FIELD_WRITE_ERROR_CODE = 27	Object field write error
public int E0028_INVALID_LENGTH_ERROR_CODE = 28	Invalid read/write length
public int E0029_INVALID_FIELD_TYPE_CODE = 29	Invalid field type (object structure error)
public int E0030_INVALID_WRITE_OP_CODE = 30	Invalid read/write operation, allocation is not raw or out of fixed adderess
public int E0050_CREATE_INDEX_ERROR_CODE = 50	Create index error
public int E0051_OPEN_INDEX_ERROR_CODE = 51	Open index error
public int E0052_DELETE_INDEX_ERROR_CODE = 52	Delete index error
public int E0053_INDEX_NOT_OPENED_CODE = 53	Index is not opened
public int E0054_ID_NOT_SPECIFIED_CODE = 54	ID is not specified for non-unique index

3.2.4 Methods

Name	Parameter(s)	Description

public static string GetMessage(int code)	**code** – error code	Static method, returns the basic error message for error code.

3.3 public class VSIndex

3.3.1 Properties

Property	Description
public string CurrentKey	Current key for enumerator. Empty string if enumerator has not been initiated.
public byte[] CurrentKeyBytes	Current key for enumerator (byte representation). Empty array if enumerator has not been initiated.
public long[] CurrentRefs	Array of references for the CurrentKey. For unique index, only one item returned. Empty array if CurrentKey is undefined.
public string Error	Error message if any error occurred, otherwise empty.
public string[] Keys	Returns the array of all keys in this index.
public string Name	Index name if opened, otherwise empty.
bool UniqueIndex	True if index is opened and unique, otherwise false.

3.3.2 Constructors

N/A

VSIndex can only be returned by VSpace

3.3.3 Methods

Name	Parameter(s)	Description
public bool Delete(byte[] key, long id)	**key** – byte representation of the key. **id** – reference id for non-unique index; 0 – delete all references (default)	Delete key from index. 'id' is optional and not required if index is unique. If id=0 for non-unique index then all references will be deleted for this key.
public bool Delete(string key, long id)	**key** – string representation of the key. **id** – reference id for non-unique index; 0 – delete all references (default)	Delete key from index. 'id' is optional and not required if index is unique. If id=0 for non-unique index then all references will be deleted for this key.
public bool Exists(byte[] key, bool partial)	**key** – byte representation of the key **partial** – true for partial search, otherwise false (default)	Check if key already exists.
public bool Exists(string key, bool partial)	**key** – string representation of the key **partial** – true for partial search, otherwise false (default)	Check if key already exists.

public long Find(byte[] k, bool partial)	**key** – byte representation of the key **partial** – true for partial search, otherwise false (default)	Find reference by the specified key. Partial search allowed (if partial=true). If more than one reference exists for non-unique index then the first reference returned. -1 returned if key not found.
public long Find(string key, bool partial)	**key** – string representation of the key **partial** – true for partial search, otherwise false (default)	Find reference by the specified key. Partial search allowed (if partial=true). If more than one reference exists for non-unique index then the first reference returned. -1 returned if key not found.
public long[] FindAll(byte[] key, bool partial)	**key** – byte representation of the key **partial** – true for partial search, otherwise false (default)	Find reference by the specified key. Partial search allowed (if partial=true). All references returned for non-unique index. For unique index the array will always contain one reference. Empty array returned if key not found.
public long[] FindAll(string key, bool partial)	**key** – string representation of the key **partial** – true for partial search, otherwise false (default)	Find reference by the specified key. Partial search allowed (if partial=true). All references returned for non-unique index. For unique index the array will always contain one reference. Empty array returned if key not found.
public bool Insert(byte[] key, long value)	**key** – byte representation of the key **value** – reference id	Inserts new key, or key reference for non-unique index. Returns true if successful, otherwise false (e.g. unique key already exists).
public bool Insert(string key, long value)	**key** – string representation of the key **value** – reference id	Inserts new key, or key reference for non-unique index. Returns true if successful, otherwise false (e.g. unique key already exists).
public bool Next()		Move to the next key in enumerator. Returns true if key exists.
public void Reset()		Reset enumerator.
public void Reset(byte[] key, bool partial)	**key** – byte representation of the key **partial** – true for partial search, otherwise false (default)	Reset enumerator using the specified key.
public void Reset(string key, bool partial)	**key** – string representation of the key **partial** – true for partial search, otherwise false (default)	Reset enumerator using the specified key.

3.4 static public class VSLib

3.4.1 Methods

Name	Parameter(s)	Description
static public bool Compare(string pattern, string value)	**pattern** – pattern for comparison; can contain '?' and '*' characters **value** – string for comparison	Returns true if input string matches pattern
public static decimal ConvertByteToDecimal(byte[] value)	**value** – byte array to convert	Convert byte array starting at specified index to decimal value
public static int ConvertByteToInt(byte[] value, int offset)	**value** – byte array to convert **offset** – byte array offset (default 0)	Convert byte array starting at specified index to int value
public static long ConvertByteToLong(byte[] value)	**value** – byte array to convert	Convert byte array to long value
public static short ConvertByteToShort(byte[] value)	**value** – byte array to convert	Convert byte array to short value
public static string ConvertByteToString(byte[] value)	**value** – byte array to convert	Convert byte array to string value
public static string ConvertByteToString(byte[] value, int index, int length)	**value** – byte array to convert **index** – offset in array **length** – number of bytes	Convert byte array to string value, offset and length must be defined
public static uint ConvertByteToUint(byte[] value)	**value** – byte array to convert	Convert byte array to uint value
public static byte[] ConvertIntToByte(int value)	**value** – int value to convert	Convert int value to byte array
public static string ConvertIntToHexString(int value)	**value** – int value to convert	Convert int value to hexadecimal string representation
public static byte[] ConvertLongToByte(long value)	**value** – long value to convert	Convert long value to byte array
public static string ConvertLongToHexString(long value)	**value** – long value to convert	Convert long value to hexadecimal string representation
public static byte[] ConvertShortToByte(short value)	**value** – short value to convert	Convert short value to byte array
public static byte[] ConvertStringToByte(string value)	**value** – string value to convert	Convert string value to byte array
public static string ConvertStringToHexString(string value)	**value** – string value to convert	Convert string value to hexadecimal string representation
public static int ConvertStringToInt(string value)	**value** – string value to convert	Convert string representation of the numeric value to int value
public static long ConvertStringToLong(string value)	**value** – string value to convert	Convert string representation of the numeric value to long value
public static byte[] ConvertUintToByte(uint value)	**value** – uint value to convert	Convert uint value to byte array

public static string ConvertULongToHexString(ulong value)	**value** – ulong value to convert	Convert ulong value to byte array
static public string[] Parse(string value, string delimiters)	**value** – string value to parse **delimiters** – string containing one or more delimiter characters ('/' by default)	Parse string to string array by delimiters. Leading and ending character of the value cannot be '/' (they will be removed if appeared)
public static byte[] ReadBytes(FileStream fs, long offset, int len)	**fs** – FileStream to read **offset** – offset to read data **len** – number of bytes to read	Read bytes from the specified FileStream
public static int ReadInt(FileStream fs, long offset)	**fs** – FileStream to read **offset** – offset to read data	Read int value from the specified FileStream
public static long ReadLong(FileStream fs, long offset)	**fs** – FileStream to read **offset** – offset to read data	Read long value from the specified FileStream
public static short ReadShort(FileStream fs, long offset)	**fs** – FileStream to read **offset** – offset to read data	Read short value from the specified FileStream
public static string ReadString(FileStream fs, long offset, int len)	**fs** – FileStream to read **offset** – offset to read data **len** – number of bytes to read	Read string value from the specified FileStream
public static string VSGetKey(string key)	**key** – key file name	Read key value from the file. Key directory is defined in DEFS.KEY_DIRECTORY
public static void VSSetKey(string key, string value)	**key** – key file name **value** – key value	Write key value to the file. Key directory is defined in DEFS.KEY_DIRECTORY
public static void Write(FileStream fs, long offset, <data>)	**fs** – FileStream to read **offset** – offset to read data **data** – data to write, one of following: ref byte[] data short data int data long data string data	A group of methods to write data to the FileStream. If offset is -1 then data is written at the current position of the FileStream

3.5 public class VSLogger

3.5.1 Properties

Property	Description
public long Length	The number of records in the log file; -1 if log is not opened

3.5.2 Constructors

Constructor	Parameter(s)	Description
public VSLogger()		Create logger object

3.5.3 Methods

Name	Parameter(s)	Description
public void Archive()		The current log content moved to archive file
public void Close()		Close current log
public void Delete()		Delete current log files
public void Open(string path, string name)	**path** – path to the log files location **name** – log name (it will be use in the log file name)	Open log; new log files will be created if doesn't exist
public void Purge()		Delete all content from the log file; the file itself not deleted
public string Read()		Read next record from the log
public string ReadAt(long n)	**n** – record number, starting at 0	Read record by the specified number
public string[] ReadRecords(long f, long n)	**f** – first record number (0 by default) **n** – the number of records (0 by default). 0 means all records starting at **f**	Read array of records from the log file
public void Write(string data)	**data** – string to write	Write record to the log file

3.6 public class VSObject

Basic space allocation object, created by VSpace.Create method. It provides unique identifier (ID) if requested, basic navigation (Next, Previous), protected read/write methods for all supported data types using relative allocation address (starting at 0). Read/write attempt out of the allocated space address will cause exception

It also provides capability to manage allocated space as set of the typed fields. Get/Set methods allows assign and change field value, as well as remove existing field.

Field names are NOT case-sensitive.

Set methods: if allocated space is not sufficient to store the specified value, this allocation extended automatically.

Get methods: if requested field has not been set, the default value returned (0 for numeric types, string with zero length for string data, byte array with zero length for byte[] data).

Note: 'Write' methods can be used only for raw object (contains no fields). If at least one field exists, 'Write' will cause exception.

3.6.1 Properties

Property	Description
public string[] Fields	String array containing all field names for this object

public long Id	Object unique identified; 0 if not assigned
public short Pool	Allocation pool
public VSObject Next	Next VSObject for space allocation in the pool (null if last allocation)
public VSObject Previous	Previous VSObject for allocation in the pool (null if 1st allocation)
public long Size	Allocation size (bytes)

3.6.2 Constructors

N/A

VSObject can only be returned by VSpace object methods (Create, GetObject)

3.6.3 Methods

Name	Parameter(s)	Description
public byte GetByte(string name)	**name** – field name	Get method, returns byte value
public byte[]GetBytes(string name)	**name** – field name	Get method, returns byte array
public DateTime GetDateTime(string name)	**name** – field name	Get method, returns DateTime object
public decimal GetDecimal(string name)	**name** – field name	Get method, returns decimal value
public int GetInt(string name)	**name** – field name	Get method, returns int value
public long GetLong(string name)	**name** – field name	Get method, returns long value
public short GetShort(string name)	**name** – field name	Get method, returns short value
public string GetString(string name, long length)	**name** – field name	Get method, returns string value
public void Set(long address, byte[] data, long length)	**address** – relative address of the allocated space **data** – byte array to write **length** – number of bytes to read	Get method, writes byte array (full or limited by the specified length)
public void Set (string name, <data>)	**name** – field name **<data>** - one of the following: byte data byte[] data string data int data long data short data decimal data DateTime data	A group of methods that allows set operation for different data types
public string GetType(string name)	**name** – field name	Returns string representation of the field type, one of: byte, short, int, long, decimal, datetime, string, bytes. If fiels with the specified name does not exist, 'undefined' returned.

public bool Delete(string name)	**name** – field name	Returns false if field is not defined, otherwise true.
public bool Exists(string name)		
public bool Exists(string name)	**name** – field name	Returns false if field is not defined, otherwise true.
public byte ReadByte(long address)	**address** – relative address of the allocated space	Protected read method, returns byte value
public byte[]ReadBytes(long address, long length)	**address** – relative address of the allocated space **length** – number of bytes to read	Protected read method, returns byte array
public DateTime ReadDateTime(long address)	**address** – relative address of the allocated space	Protected read method, returns DateTime object
public decimal ReadDecimal(long address)	**address** – relative address of the allocated space	Protected read method, returns decimal value
public int ReadInt(long address)	**address** – relative address of the allocated space	Protected read method, returns int value
public long ReadLong(long address)	**address** – relative address of the allocated space	Protected read method, returns long value
public short ReadShort(long address)	**address** – relative address of the allocated space	Protected read method, returns short value
public string ReadString(long address, long length)	**address** – relative address of the allocated space **length** – number of bytes to read	Protected read method, returns string value
public void Write(long address, byte[] data, long length)	**address** – relative address of the allocated space **data** – byte array to write **length** – number of bytes to read	Protected write method, writes byte array (full or limited by the specified length)
public void Write (long address, <data>)	**address** – relative address of the allocated space **<data>** - one of the following: byte data string data int data long data short data decimal data DateTime data	A group of methods that allows protected write operation for different data types

3.7 public class VSpace

3.7.1 Properties

Property	Description
public string Error	Last error message
public long Id	Unique space Id
public string Name	Space name
public string Owner	Space owner (system/application can assign this name). By default – '$UNDEFINED$'

public long Size	Space size (bytes)

3.7.2 Constructors

N/A

VSpace can only be returned by VSEngine.

3.7.3 Methods

Name	Parameter(s)	Description
public VSObject Allocate(long size, short pool, bool generateID, long chunk)	**size** – number of bytes to allocate **pool** – allocation pool number **generateID** – true if unique ID is required (default), otherwise false **chunk** – number of bytes in allocation chunk (0 by default – continuous allocation). Minimal allocation chunk is 128 bytes. Maximum number of chunks in one allocation is 32767.	Create new instance of the VSObject.
public void CreateIndex(string name, bool unique)	**name** – index name **unique** – true for unique index, otherwise false	Create new index. If index already exists then error will occur.
public void DeleteIndex(string)	**name** – index name	Delete index with the specified name.
public void Extend(VSObject a, long size)	**a** – VSObject for the existing allocation **size** – number of extension bytes	Extend existing allocation defined by VSObject by the specified number of bytes (additional chunk).
public short GetFreePoolNumber()		Get free pool number for dynamic pool allocation
public void GetIndex(string name)	**name** – index name	Open existing index. If index does not exist, error will occur.
public VSObject GetObject(long id)	**id** – object identifier	Get VSObject object for allocated space by ID
public short GetObjectPool(long id)	**id** – object identifier	Get pool allocation for specified object ID without loading object.
public long[] GetPoolPointers(short pool)	**pool** – allocation pool number	Get absolute addresses of the 1st and last objects in the specified pool allocation. If there is no allocation in this pool then 0 values returned, otherwise array element 0 contains address of the 1st object, element 1 – last object.
public short[] GetPools(short pool)		Get the list of pools (system and user) where there is apace allocation(s).

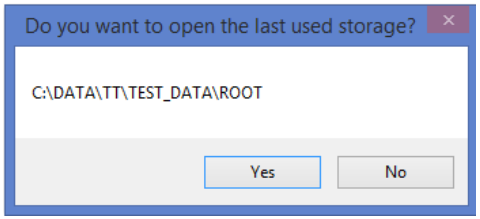
public long GetRootID(int pool)	**pool** – allocation pool number	Returns ID of the 1st object in the pool or 0 if there are no objects in this pool or ID is not assigned
public VSObject GetRootObject(int pool)	**pool** – allocation pool number	Returns VSObject for the 1st object in the pool or null if there are no objects in this pool
public bool IndexExists(string name)	**name** – index name	Returns true if index exists, otherwise false.
public void Release(VSObject a, bool deleteID)	**a** – VSObject for the existing allocation **deleteID** – true if the unique identified shall be deleted from the system (default). Otherwise, this ID will remain in the system.	Release allocated space
public int ReleaseID(long id)	**id** – identifier of the allocated space (VSObject attribute)	Release allocated space by ID.
public void ReleasePool(short n)	**n** – pool number	Release all space in allocation pool.
public void RemoveAllIndexes(long id)	**id** – identifier of the allocated space (VSObject attribute)	Release all indexes for the specified object ID.

IV. Administration Tool

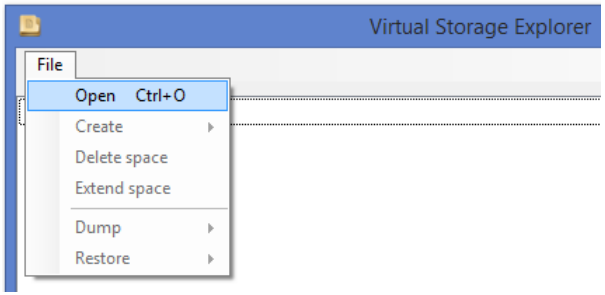
Administration tool (VStorageExplorer) allows virtual storage management via GUI.

1. Open storage

   If any storages was opened previously then system will prompt the confirmation to open this storage:



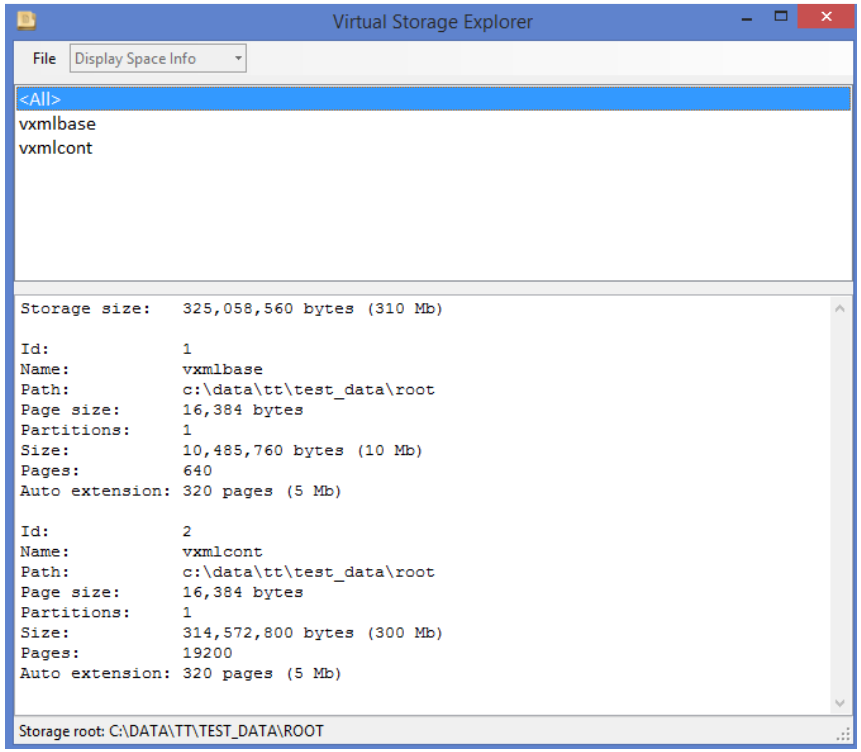
   Otherwise you should use ‘Open’ menu and select the storage directory:


Virtual Storage Explorer
File
Open Ctrl+O
Create
Delete space
Extend space
Dump
Restore

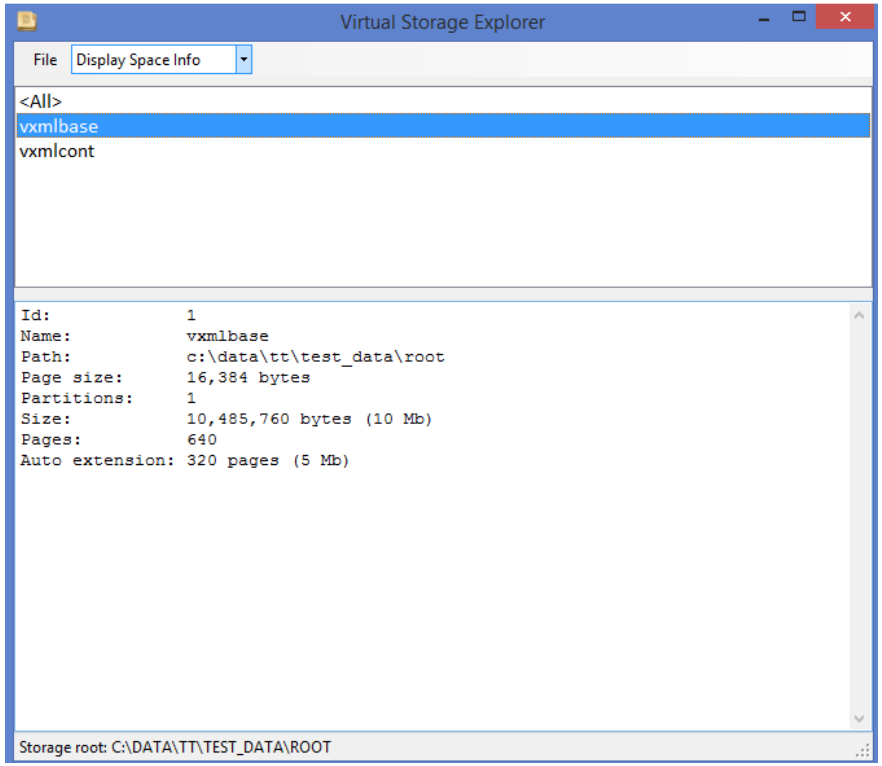
2. Display storage information

   Precondition: storage must be opened.

   If you select '<All>' in the list box then the storage summary information will be displayed:



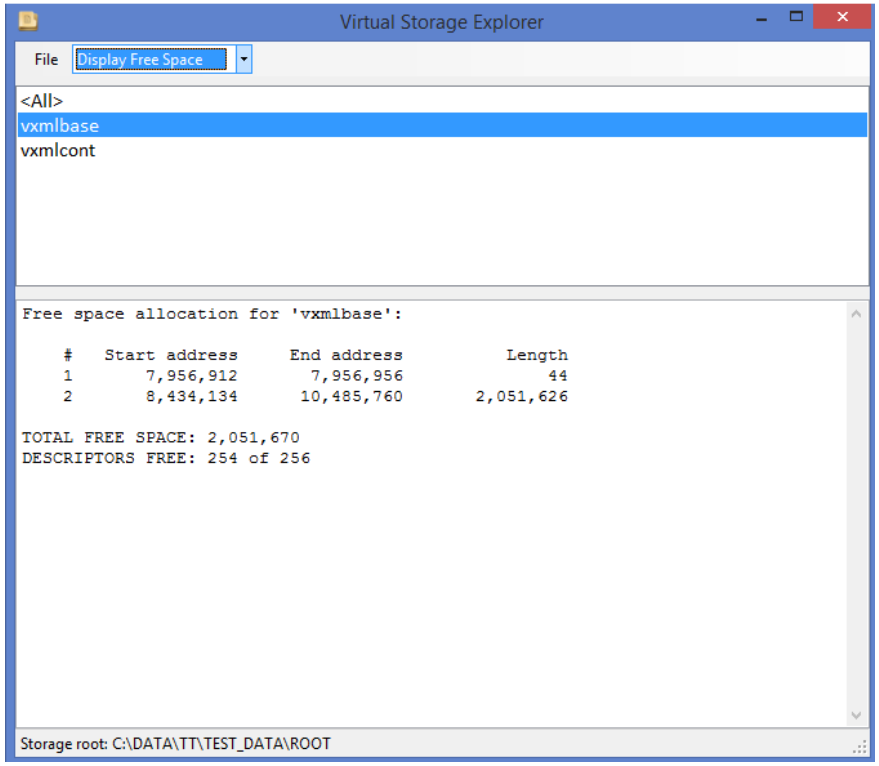
3. Display space information
   Precondition: storage must be opened.
   If you select ‘Display Space Info’ in the menu line combo box and the space name in the list box then this particular space information will be displayed:



4. Display free space

   Precondition: storage must be opened.

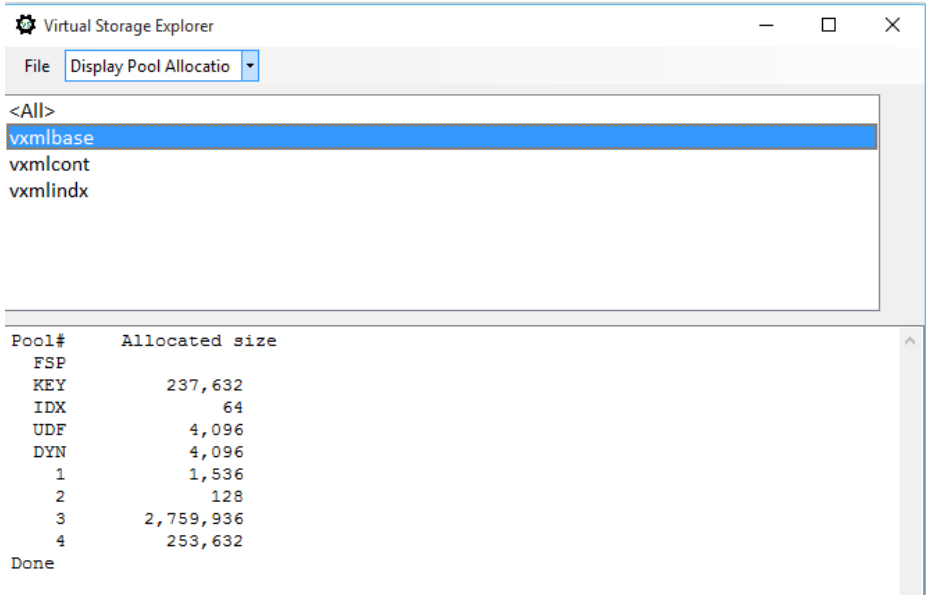
   If you select ‘Display Free Space’ in the menu line combo box and the space name in the list box then this particular free space information will be displayed:



5. Display pool allocation

   Precondition: storage must be opened.

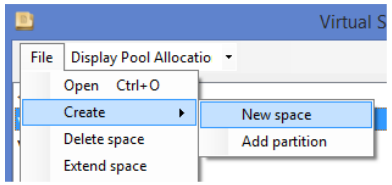
   If you select ‘Display Pool Allocation’ in the menu line combo box and the space name in the list box, then pool allocation for this particular free space will be displayed:



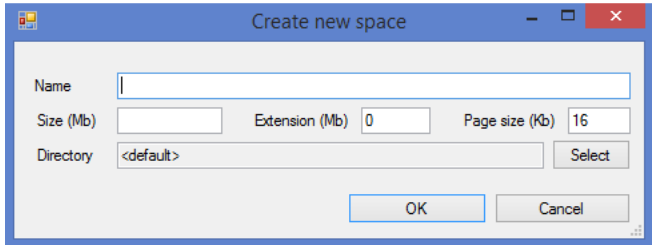
6. Create new space

   Precondition: storage must be opened.

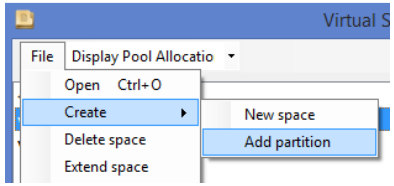
   To create new space in the storage you must select menu ‘File/Create/New space’:



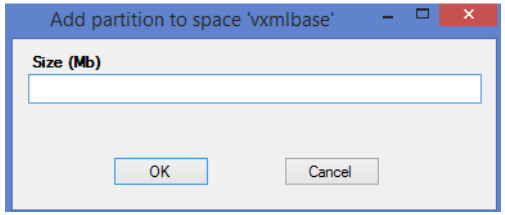
   Pop-up window will be opened to populate space parameters:



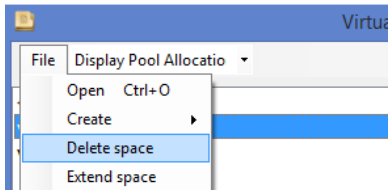
7. Add new partition
   Precondition: storage must be opened.
   To create new partition of the existing space in the storage you must select menu ‘File/Create/Add partition’:



   Pop-up window will be opened to populate new partition size:

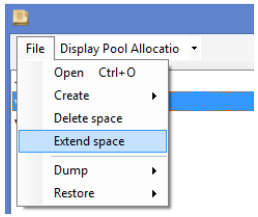


8. Delete space
   Precondition: storage must be opened.
   To delete existing space in the storage you must select menu ‘File/Delete space’:

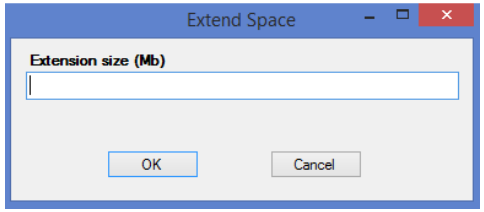


You must confirm this action in the dialog window.

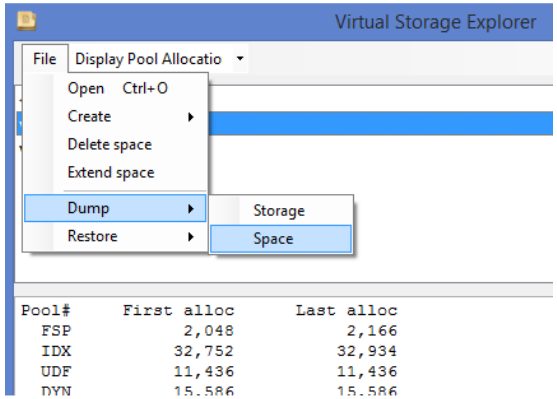
9. Extend space
   Precondition: storage must be opened.
   To extend existing space in the storage you must select menu ‘File/Extend space’:



   Pop-up window will be opened to populate extension size:

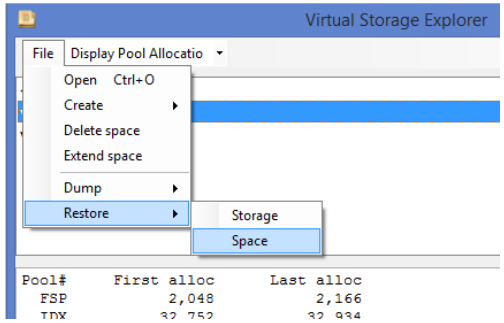


10. Dump storage
    Precondition: storage must be opened.
    You can create reserved copy in the portable format of the whole storage or single space by selecting ‘File/Dump/[Storage|Space]’:



System will show pop-up dialog to specify the directory for dump file(s).

11. Restore storage
    Precondition: storage must be opened.
    You can restore the whole storage or single space from the previously created reserved copy (dump) by selecting ‘File/Restore/[Storage|Space]’:



System will show pop-up dialog to specify the directory where dump file(s) are located.

V. Command-line Utility

VSUtil is command-line tool for storage management.

Command and one or more parameters shall be specified.

All commands and parameters are NOT case-sensitive and could be truncated unambiguously (e.g. ‘-name’, ‘-n’, ‘-na’) .

Commands:

create - create new space; parameters:

-name <space-name> [-root <root-path>] [-pagesize <page-size>] [-extension <extension>]

[-directory <space-path>]

extend - extend existing space; parameters:

-name <space-name> [-root <root-path>] [-size <extension-size>]

remove - remove space; parameters:

-name <space-name> [-root <root-path>]

addpartition - add partition to the existing space; parameters:

-name <space-name> [-root <root-path>] [-size <partition-size>]

dump - create backup copy of the storage or single space; parameters:

-name <space-name> [-root <root-path>] [-directory <dump-path>]

restore - restore storage or single space from the backup copy; parameters:

-name <space-name> [-root <root-path>] [-directory <dump-path>]

list - display information about all storage spaces or single space; parameters:

-name <space-name> [-root <root-path>]

Parameters:

-**name** - space name; mandatory parameter; wildcards can be used for ‘dump’, ‘restore’ and ‘list’;

-**root** - storage root directory; if not specified, the current directory is assumed as root;

-**pagesize** - space page size in Kbytes (‘create’ only); optional, default value is 16;

-**size** - space size in Mbytes (‘create’ only); optional, default value is 5;

-**extension** - space auto extension size in Mbytes (‘create’ only); optional, default value is 0;

-**directory** - path to space file if different from the storage root directory (‘create’ only);

Usage:

VSUtil <command> [-<parameter_1> <value>]…[-<parameter_n> <value>]

Examples:

VSUtil create –name myspace –size 120 extension 20

VSUtil extend –root c:\data\mystorage –name myspace –size 50

VSUtil list –root c:\data\mystorage –name *space

VSUtil dump –name * -directory c:\backup

Command and parameters (M – mandatory, O – optional):

Parameter / Command	Name	Root	Pagesize	Size	Extension	Directory
Create	M	O	O	O	O	O
Extend	M	O	-	M	-	-
Remove	M	O	-	-	-	-
Addpartition	M	O	-	M	-	-
Dump	M	O	-	-	-	O

Restore	M	O	-	-	-	O
List	M	O	-	-	-	-

VI. Code sample

1. Command-line utility (VSUtil) source code

```
static void Main(string[] args)
{
    const string DEF_CMD_CREATE = "create";
    const string DEF_CMD_EXTEND = "extend";
    const string DEF_CMD_REMOVE = "remove";
    const string DEF_CMD_ADDPARTITION = "addpartition";
    const string DEF_CMD_DUMP = "dump";
    const string DEF_CMD_RESTORE = "restore";
    const string DEF_CMD_LIST = "list";

    string[] cmds = { DEF_CMD_CREATE, DEF_CMD_EXTEND, DEF_CMD_REMOVE, DEF_CMD_ADDPARTITION, DEF_CMD_DUMP,
    DEF_CMD_RESTORE, DEF_CMD_LIST };

    const string DEF_OP_NAME = "-n";
    const string DEF_OP_ROOT = "-r";
    const string DEF_OP_PAGESIZE = "-p";
    const string DEF_OP_SPACESIZE = "-s";
    const string DEF_OP_EXTEND = "-e";
    const string DEF_OP_DIRECTORY = "-d";



    string errmsg = "Invalid parameter";
    string errexe = "Command execution error";

    string cmd = "";

    string err = "";
    string root = "";
    string dir = "";
```


mailto: ivvvsx@gmail.com

```
string name = "";
string size = "";
string ext = "";
string page = "";
VSEngine vs;

if (args.Length == 0)
    err = errmsg + " - command is not specified";
else
{
    for (int i = 0; i < cmds.Length; i++)
    {
        if (cmds[i].IndexOf(args[0].ToLower()) == 0)
        {
            if (cmd != "")
            {
                err = errmsg + " - umbiguous command - '" + args[0] + "'";
                break;
            }
            else
                cmd = cmds[i];
        }
    }

    if (cmd == "")
        err = errmsg + " - command is not recognized";
    else
    {
        if (err == "")
        {
            root = getParameterValue(args, DEF_OP_ROOT);
            if (root.Substring(0, 1) == ":")
                err = errmsg + " - rooth path is not specified or incorrect";
            else
            {
                name = getParameterValue(args, DEF_OP_NAME);
                dir = getParameterValue(args, DEF_OP_DIRECTORY);        // Space directory
                size = getParameterValue(args, DEF_OP_SPACESIZE);
                ext = getParameterValue(args, DEF_OP_EXTEND);
                page = getParameterValue(args, DEF_OP_PAGESIZE);

                vs = new VSEngine(root);
```

```
                    if (cmd == DEF_CMD_CREATE)
                    {
                        Console.WriteLine(msg100 + ", command='CREATE'");
                        if (name.Substring(0, 1) == ":")
                            err = name;
                        else if (size.Substring(0, 1) == ":")
                            err = size;
                        else if (ext.Substring(0, 1) == ":")
                            err = ext;
                        else if (page.Substring(0, 1) == ":")
                            err = page;
                        else if (dir.Length > 0)
                        {
                            if (dir.Substring(0, 1) == ":")
                                err = dir;
                        }

                        if (err == "")
                        {
                            try
                            {
                                vs.Create(name, Convert.ToInt32(page), Convert.ToInt32(size), Convert.ToInt32(ext),
dir);
                            }
                            catch (VSException e)
                            {
                                Console.WriteLine(errexe);
                                err = e.Message;
                            }
                        }
                        else
                            err = errmsg + err;
                    }

                    else if (cmd == DEF_CMD_EXTEND)
                    {
                        Console.WriteLine(msg100 + ", command='EXTEND'");
                        ext = getParameterValue(args, "-e");

                        if (name.Substring(0, 1) == ":")
                            err = name;
```

```
        else if (ext.Substring(0, 1) == ":")
            err = ext;

        if (err == "")
        {
            try
            {
                vs.Extend(name, Convert.ToInt32(ext));
            }
            catch (VSException e)
            {
                Console.WriteLine(errexe);
                err = e.Message;
            }
        }
        else
            err = errmsg + err;
    }

    else if (cmd == DEF_CMD_REMOVE)
    {
        Console.WriteLine(msg100 + ", command='REMOVE'");

        if (name.Substring(0, 1) == ":")
            err = name;

        if (err == "")
        {

            try
            {
                vs.Remove(name);
            }
            catch (VSException e)
            {
                Console.WriteLine(errexe);
                err = e.Message;
            }
        }
        else
            err = errmsg + err;
    }
```

```
else if (cmd == DEF_CMD_ADDPARTITION)
{
    Console.WriteLine(msg100 + ", command='ADD PARTITION'");

    if (name.Substring(0, 1) == ":")
        err = name;
    else if (size.Substring(0, 1) == ":")
        err = size;

    if (err == "")
    {
        try
        {
            vs.AddPartition(name, Convert.ToInt32(size));
        }
        catch (VSException e)
        {
            Console.WriteLine(errexe);
            err = e.Message;
        }
    }
    else
        err = errmsg + err;
}

else if (cmd == DEF_CMD_DUMP)
{
    Console.WriteLine(msg100 + ", command='DUMP'");

    if (name.Substring(0, 1) == ":")
        err = name;
    if (dir.Length > 0)
    {
        if (dir.Substring(0, 1) == ":")
            err = dir;
    }

    if (err == "")
    {
        string rc = vs.Dump(dir, name);
        if (rc != "")
```

```
                err = "Dump error - " + rc;
            }
        }

        else if (cmd == DEF_CMD_RESTORE)
        {
            Console.WriteLine(msg100 + ", command='RESTORE'");

            if (name.Substring(0, 1) == ":")
                err = name;

            if (dir.Length > 0)
            {
                if (dir.Substring(0, 1) == ":")
                    err = dir;
            }

            if (err.Length == 0)
            {
                string rc = vs.Restore(dir, name);
                if (rc != "")
                    err = "Restore error - " + rc;
            }
        }

        else if (cmd == DEF_CMD_LIST)
        {
            Console.WriteLine(msg100 + ", command='LIST'");

            if (name.Substring(0, 1) == ":")
                err = name;
            else
            {
                string[] rc = vs.List(name);
                for (int i = 0; i < rc.Length; i++)
                    Console.WriteLine(rc[i]);
            }

        }
        else
            err = "Invalid command - " + cmd;
    }
```

```
            }
        }
        int r = 0;
        if (err != "")
        {
            Console.WriteLine(err);
            r = 8;
        }
        Console.WriteLine("Ended, Rc = " + r.ToString() + ", " + DateTime.Now.ToString("s"));
    }
}
```

2. Create new object and set initial attributes

```
protected void Create(short type, string name, string value = "", long ownerid = 0)
 {

     long sz = name.Length + value.Length + 64;

     this.OBJ = NodeSpace.Allocate(sz, type);

     if (ownerid == 0)
         this.OwnerId = this.OBJ.Id;
     else
         this.OwnerId = ownerid;

     OBJ.Set(VXmlDefs.F_NAME, name);

     if (value != "")
         OBJ.Set(VXmlDefs.F_VALUE, value);
}
```

3. Set field value

```
protected long RefId
 {
     get { return OBJ.GetLong(VXmlDefs.F_REF_ID); }
     set
     {
         if (value == 0)
```

```
                OBJ.Delete(VXmlDefs.F_REF_ID);
            else
                OBJ.Set(VXmlDefs.F_REF_ID, value);
        }
    }
```

VII. Contact Us

mailto:ivvvsx@gmail.com