vdsf API Reference Manual

0.3.0

Generated by Doxygen 1.5.1

Wed Jun 25 21:36:34 2008

Contents

1 vdsf API Module Index

1.1 vdsf API Modules

Here is a list of all modules:

2 vdsf API Directory Hierarchy

2.1 vdsf API Directories

This directory hierarchy is sorted roughly, but not completely, alphabetically:

3 vdsf API Data Structure Index

3.1 vdsf API Data Structures

Here are the data structures with brief descriptions:

4 vdsf API File Index

4.1 vdsf API File List

Here is a list of all files with brief descriptions:

5 vdsf API Module Documentation

5.1 API functions for vdsf folders.

Functions

- VDSF_EXPORT int vdsFolderClose (VDS_HANDLE objectHandle)

  Close a folder.

- VDSF_EXPORT int vdsFolderCreateObject (VDS_HANDLE folderHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjectDefinition ∗pDefinition)

  Create a new object in shared memory as a child of the current folder.

- VDSF_EXPORT int vdsFolderCreateObjectXML (VDS_HANDLE folderHandle, const char ∗xmlBuffer, size_t lengthInBytes)

  Create a new object in shared memory as a child of the current folder.

- VDSF_EXPORT int vdsFolderDestroyObject (VDS_HANDLE folderHandle, const char ∗objectName, size_t nameLengthInBytes)

  Destroy an object, child of the current folder, in shared memory.

- VDSF_EXPORT int vdsFolderGetFirst (VDS_HANDLE objectHandle, vdsFolderEntry ∗pEntry)

  Iterate through the folder - no data items are removed from the folder by this function.

- VDSF_EXPORT int vdsFolderGetNext (VDS_HANDLE objectHandle, vdsFolderEntry ∗pEntry)

  Iterate through the folder.

- VDSF_EXPORT int vdsFolderOpen (VDS_HANDLE sessionHandle, const char ∗folderName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing folder (see vdsCreateObject to create a new folder).

- VDSF_EXPORT int vdsFolderStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the folder.

5.1.1 Function Documentation

5.1.1.1 VDSF_EXPORT int vdsFolderClose (VDS_HANDLE *objectHandle*)

Close a folder.

This function terminates the current access to the folder in shared memory (the folder itself is untouched).

Parameters:

← ***objectHandle*** The handle to the folder (see vdsFolderOpen).

Returns:

0 on success or a vdsErrors on error.

5.1.1.2 VDSF_EXPORT int vdsFolderCreateObject (VDS_HANDLE *folderHandle*, const char ∗ *objectName*, size_t *nameLengthInBytes*, vdsObjectDefinition ∗ *pDefinition*)

Create a new object in shared memory as a child of the current folder.

The creation of the object only becomes permanent after a call to vdsCommit.

This function does not provide a handle to the newly created object. Use vdsQueueOpen and similar functions to get the handle.

Parameters:

← ***folderHandle*** Handle to the current folder.

← ***objectName*** The name of the object.

← ***nameLengthInBytes*** The length of *objectName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

← ***pDefinition*** The type of object to create (folder, queue, etc.) and the "optional" definition.

Returns:

0 on success or a vdsErrors on error.

5.1.1.3 VDSF_EXPORT int vdsFolderCreateObjectXML (VDS_HANDLE *folderHandle*, const char ∗ *xmlBuffer*, size_t *lengthInBytes*)

Create a new object in shared memory as a child of the current folder.

The creation of the object only becomes permanent after a call to vdsCommit.

This function does not provide a handle to the newly created object. Use vdsQueue-Open and similar functions to get the handle.

Parameters:

← ***folderHandle*** Handle to the current folder.

← ***xmlBuffer*** The XML buffer (string) containing all the required information.

← ***lengthInBytes*** The length of *xmlBuffer* (in bytes) not counting the null terminator.

Returns:

0 on success or a vdsErrors on error.

5.1.1.4 VDSF_EXPORT int vdsFolderDestroyObject (VDS_HANDLE *folderHandle*, const char ∗ *objectName*, size_t *nameLengthInBytes*)

Destroy an object, child of the current folder, in shared memory.

The destruction of the object only becomes permanent after a call to vdsCommit.

Parameters:

← ***folderHandle*** Handle to the current folder.

← ***objectName*** The name of the object.

← ***nameLengthInBytes*** The length of *objectName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

Returns:

0 on success or a vdsErrors on error.

5.1.1.5 VDSF_EXPORT int vdsFolderGetFirst (VDS_HANDLE *objectHandle*, vdsFolderEntry ∗ *pEntry*)

Iterate through the folder - no data items are removed from the folder by this function.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Parameters:

← ***objectHandle*** The handle to the folder (see vdsFolderOpen).

→ ***pEntry*** The data structure provided by the user to hold the content of each item in the folder. Memory allocation for this buffer is the responsability of the caller.

Returns:

0 on success or a vdsErrors on error.

5.1.1.6 VDSF_EXPORT int vdsFolderGetNext (VDS_HANDLE *objectHandle*, vdsFolderEntry ∗ *pEntry*)

Iterate through the folder.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Evidently, you must call vdsFolderGetFirst to initialize the iterator.

Parameters:

← ***objectHandle*** The handle to the folder (see vdsFolderOpen).

→ ***pEntry*** The data structure provided by the user to hold the content of each item in the folder. Memory allocation for this buffer is the responsability of the caller.

Returns:

0 on success or a vdsErrors on error.

5.1.1.7 VDSF_EXPORT int vdsFolderOpen (VDS_HANDLE *sessionHandle*, const char ∗ *folderName*, size_t *nameLengthInBytes*, VDS_HANDLE ∗ *objectHandle*)

Open an existing folder (see vdsCreateObject to create a new folder).

Parameters:

← ***sessionHandle*** The handle to the current session.

← ***folderName*** The fully qualified name of the folder.

← ***nameLengthInBytes*** The length of *folderName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

→ ***objectHandle*** The handle to the folder, allowing us access to the folder in shared memory. On error, this handle will be set to zero (NULL) unless the objectHandle pointer itself is NULL.

Returns:

0 on success or a vdsErrors on error.

5.1.1.8 VDSF_EXPORT int vdsFolderStatus (VDS_HANDLE *objectHandle*, vdsObjStatus ∗ *pStatus*)

Return the status of the folder.

Parameters:

← ***objectHandle*** The handle to the folder (see vdsFolderOpen).

→ ***pStatus*** A pointer to the status structure.

Returns:

0 on success or a vdsErrors on error.

5.2 API functions for vdsf hash maps.

5.2.1 Detailed Description

Hash maps use unique keys - the data items are not sorted.

Functions

- VDSF_EXPORT int vdsHashMapClose (VDS_HANDLE objectHandle)

  Close a Hash Map.

- VDSF_EXPORT int vdsHashMapDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

  Retrieve the data definition of the hash map.

- VDSF_EXPORT int vdsHashMapDelete (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength)

  Remove the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsHashMapGet (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗returned-Length)

  Retrieve the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsHashMapGetFirst (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsHashMapGetNext (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsHashMapInsert (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Insert a data element in the hash map.

- VDSF_EXPORT int vdsHashMapOpen (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗object-Handle)

  Open an existing hash map (see vdsCreateObject to create a new object).

- VDSF_EXPORT int vdsHashMapReplace (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Replace a data element in the hash map.

- VDSF_EXPORT int vdsHashMapStatus (VDS_HANDLE objectHandle, vds-ObjStatus ∗pStatus)

  Return the status of the hash map.

5.2.2 Function Documentation

5.2.2.1 VDSF_EXPORT int vdsHashMapClose (VDS_HANDLE *objectHandle*)

Close a Hash Map.

This function terminates the current access to the hash map in shared memory (the hash map itself is untouched).

Warning:

Closing an object does not automatically commit or rollback data items that were inserted or removed. You still must use either vdsCommit or vdsRollback to end the current unit of work.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

Returns:

0 on success or a vdsErrors on error.

5.2.2.2 VDSF_EXPORT int vdsHashMapDefinition (VDS_HANDLE *objectHandle*, vdsObjectDefinition ∗∗ *definition*)

Retrieve the data definition of the hash map.

Warning:

This function allocates a buffer to hold the definition (using malloc()). You must free it (with free()) when you no longer need the definition.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

→ ***definition*** The buffer allocated by the API to hold the content of the object definition. Freeing the memory (with free()) is the responsability of the caller.

Returns:

0 on success or a vdsErrors on error.

5.2.2.3 VDSF_EXPORT int vdsHashMapDelete (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*)

Remove the data item identified by the given key from the hash map.

Data items which were added by another session and are not yet committed will not be seen by this function and cannot be removed. Likewise, destroyed data items (even if not yet committed) are invisible.

The removals only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

← ***key*** The key of the item to be removed.

← ***keyLength*** The length of the *key* buffer (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.2.2.4 VDSF_EXPORT int vdsHashMapGet (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *returnedLength*)

Retrieve the data item identified by the given key from the hash map.

Data items which were added by another session and are not yet committed will not be seen by this function. Likewise, destroyed data items (even if not yet committed) are invisible.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

← ***key*** The key of the item to be retrieved.

← ***keyLength*** The length of the *key* buffer (in bytes).

→ ***buffer*** The buffer provided by the user to hold the content of the data item. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***returnedLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.2.2.5 VDSF_EXPORT int vdsHashMapGetFirst (VDS_HANDLE *objectHandle*, void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *retKeyLength*, size_t ∗ *retDataLength*)

Iterate through the hash map.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Data items retrieved this way will not be sorted.

Parameters:

- ← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).
- → ***key*** The key buffer provided by the user to hold the content of the key associated with the first element. Memory allocation for this buffer is the responsability of the caller.
- ← ***keyLength*** The length of the *key* buffer (in bytes).
- → ***buffer*** The buffer provided by the user to hold the content of the first element. Memory allocation for this buffer is the responsability of the caller.
- ← ***bufferLength*** The length of *buffer* (in bytes).
- → ***retKeyLength*** The actual number of bytes in the key
- → ***retDataLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.2.2.6 VDSF_EXPORT int vdsHashMapGetNext (VDS_HANDLE *objectHandle*, void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *retKeyLength*, size_t ∗ *retDataLength*)

Iterate through the hash map.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Evidently, you must call vdsHashMapGetFirst to initialize the iterator. Not so evident - calling vdsHashMapGet will reset the iteration to the data item retrieved by this function (they use the same internal storage). If this cause a problem, please let us know.

Data items retrieved this way will not be sorted.

Parameters:

- ← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).
- → ***key*** The key buffer provided by the user to hold the content of the key associated with the data element. Memory allocation for this buffer is the responsability of the caller.
- ← ***keyLength*** The length of the *key* buffer (in bytes).
- → ***buffer*** The buffer provided by the user to hold the content of the data element. Memory allocation for this buffer is the responsability of the caller.
- ← ***bufferLength*** The length of *buffer* (in bytes).
- → ***retKeyLength*** The actual number of bytes in the key

$\rightarrow$ ***retDataLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.2.2.7 VDSF_EXPORT int vdsHashMapInsert (VDS_HANDLE *objectHandle*, const void $*$ *key*, size_t *keyLength*, const void $*$ *data*, size_t *dataLength*)

Insert a data element in the hash map.

The additions only become permanent after a call to vdsCommit.

Parameters:

$\leftarrow$ ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

$\leftarrow$ ***key*** The key of the item to be inserted.

$\leftarrow$ ***keyLength*** The length of the *key* buffer (in bytes).

$\leftarrow$ ***data*** The data item to be inserted.

$\leftarrow$ ***dataLength*** The length of *data* (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.2.2.8 VDSF_EXPORT int vdsHashMapOpen (VDS_HANDLE *sessionHandle*, const char $*$ *hashMapName*, size_t *nameLengthInBytes*, VDS_HANDLE $*$ *objectHandle*)

Open an existing hash map (see vdsCreateObject to create a new object).

Parameters:

$\leftarrow$ ***sessionHandle*** The handle to the current session.

$\leftarrow$ ***hashMapName*** The fully qualified name of the hash map.

$\leftarrow$ ***nameLengthInBytes*** The length of *hashMapName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

$\rightarrow$ ***objectHandle*** The handle to the hash map, allowing us access to the map in shared memory. On error, this handle will be set to zero (NULL) unless the objectHandle pointer itself is NULL.

Returns:

0 on success or a vdsErrors on error.

5.2.2.9 VDSF_EXPORT int vdsHashMapReplace (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*, const void ∗ *data*, size_t *dataLength*)

Replace a data element in the hash map.

The replacements only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

← ***key*** The key of the item to be replaced.

← ***keyLength*** The length of the *key* buffer (in bytes).

← ***data*** The new data item that will replace the previous data.

← ***dataLength*** The length of *data* (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.2.2.10 VDSF_EXPORT int vdsHashMapStatus (VDS_HANDLE *objectHandle*, vdsObjStatus ∗ *pStatus*)

Return the status of the hash map.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsHashMapOpen).

→ ***pStatus*** A pointer to the status structure.

Returns:

0 on success or a vdsErrors on error.

5.3 API functions for vdsf read-only hash maps.

5.3.1 Detailed Description

Hash maps use unique keys - the data items are not sorted.

Functions

- VDSF_EXPORT int vdsMapClose (VDS_HANDLE objectHandle)

  Close a Hash Map.

- VDSF_EXPORT int vdsMapDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

  Retrieve the data definition of the hash map.

- VDSF_EXPORT int vdsMapDelete (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength)

  Remove the data item identified by the given key from the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapEdit (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open a temporary copy of an existing hash map for editing.

- VDSF_EXPORT int vdsMapGet (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Retrieve the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsMapGetFirst (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsMapGetNext (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsMapInsert (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Insert a data element in the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapOpen (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing hash map read only (see vdsCreateObject to create a new object).

- VDSF_EXPORT int vdsMapReplace (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Replace a data element in the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the hash map.

5.3.2 Function Documentation

5.3.2.1 VDSF_EXPORT int vdsMapClose (VDS_HANDLE *objectHandle*)

Close a Hash Map.

This function terminates the current access to the hash map in shared memory (the hash map itself is untouched).

Warning:

Closing an object does not automatically commit or rollback data items that were inserted or removed (if the map was open with vdsMapEdit). You still must use either vdsCommit or vdsRollback to end the current unit of work.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).

Returns:

0 on success or a vdsErrors on error.

5.3.2.2 VDSF_EXPORT int vdsMapDefinition (VDS_HANDLE *objectHandle*, vdsObjectDefinition ∗∗ *definition*)

Retrieve the data definition of the hash map.

Warning:

This function allocates a buffer to hold the definition (using malloc()). You must free it (with free()) when you no longer need the definition.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).

→ ***definition*** The buffer allocated by the API to hold the content of the object definition. Freeing the memory (with free()) is the responsability of the caller.

Returns:

0 on success or a vdsErrors on error.

5.3.2.3 VDSF_EXPORT int vdsMapDelete (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*)

Remove the data item identified by the given key from the hash map (you must be in edit mode).

The removals only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapEdit).

← ***key*** The key of the item to be removed.

← ***keyLength*** The length of the *key* buffer (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.3.2.4 VDSF_EXPORT int vdsMapEdit (VDS_HANDLE *sessionHandle*, const char ∗ *hashMapName*, size_t *nameLengthInBytes*, VDS_HANDLE ∗ *objectHandle*)

Open a temporary copy of an existing hash map for editing.

The copy becomes the latest version of the map when a session is committed.

Parameters:

← ***sessionHandle*** The handle to the current session.

← ***hashMapName*** The fully qualified name of the hash map.

← ***nameLengthInBytes*** The length of *hashMapName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

→ ***objectHandle*** The handle to the hash map, allowing us access to the map in shared memory. On error, this handle will be set to zero (NULL) unless the objectHandle pointer itself is NULL.

Returns:

0 on success or a vdsErrors on error.

5.3.2.5 VDSF_EXPORT int vdsMapGet (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *returnedLength*)

Retrieve the data item identified by the given key from the hash map.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).

← ***key*** The key of the item to be retrieved.

← ***keyLength*** The length of the *key* buffer (in bytes).

→ ***buffer*** The buffer provided by the user to hold the content of the data item. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***returnedLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.3.2.6 VDSF_EXPORT int vdsMapGetFirst (VDS_HANDLE *objectHandle*, void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *retKeyLength*, size_t ∗ *retDataLength*)

Iterate through the hash map.

Data items retrieved this way will not be sorted.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).

→ ***key*** The key buffer provided by the user to hold the content of the key associated with the first element. Memory allocation for this buffer is the responsability of the caller.

← ***keyLength*** The length of the *key* buffer (in bytes).

→ ***buffer*** The buffer provided by the user to hold the content of the first element. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***retKeyLength*** The actual number of bytes in the key

→ ***retDataLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.3.2.7 VDSF_EXPORT int vdsMapGetNext (VDS_HANDLE *objectHandle*, void ∗ *key*, size_t *keyLength*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *retKeyLength*, size_t ∗ *retDataLength*)

Iterate through the hash map.

Evidently, you must call vdsMapGetFirst to initialize the iterator. Not so evident - calling vdsMapGet will reset the iteration to the data item retrieved by this function (they use the same internal storage). If this cause a problem, please let us know.

Data items retrieved this way will not be sorted.

Parameters:

- ← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).
- → ***key*** The key buffer provided by the user to hold the content of the key associated with the data element. Memory allocation for this buffer is the responsability of the caller.
- ← ***keyLength*** The length of the *key* buffer (in bytes).
- → ***buffer*** The buffer provided by the user to hold the content of the data element. Memory allocation for this buffer is the responsability of the caller.
- ← ***bufferLength*** The length of *buffer* (in bytes).
- → ***retKeyLength*** The actual number of bytes in the key
- → ***retDataLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.3.2.8 VDSF_EXPORT int vdsMapInsert (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*, const void ∗ *data*, size_t *dataLength*)

Insert a data element in the hash map (you must be in edit mode).

The additions only become permanent after a call to vdsCommit.

Parameters:

- ← ***objectHandle*** The handle to the hash map (see vdsMapEdit).
- ← ***key*** The key of the item to be inserted.
- ← ***keyLength*** The length of the *key* buffer (in bytes).
- ← ***data*** The data item to be inserted.
- ← ***dataLength*** The length of *data* (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.3.2.9 VDSF_EXPORT int vdsMapOpen (VDS_HANDLE *sessionHandle*, const char ∗ *hashMapName*, size_t *nameLengthInBytes*, VDS_HANDLE ∗ *objectHandle*)

Open an existing hash map read only (see vdsCreateObject to create a new object).

Parameters:

- ← ***sessionHandle*** The handle to the current session.

← ***hashMapName*** The fully qualified name of the hash map.

← ***nameLengthInBytes*** The length of *hashMapName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

→ ***objectHandle*** The handle to the hash map, allowing us access to the map in shared memory. On error, this handle will be set to zero (NULL) unless the objectHandle pointer itself is NULL.

Returns:

0 on success or a vdsErrors on error.

5.3.2.10 VDSF_EXPORT int vdsMapReplace (VDS_HANDLE *objectHandle*, const void ∗ *key*, size_t *keyLength*, const void ∗ *data*, size_t *dataLength*)

Replace a data element in the hash map (you must be in edit mode).

The replacements only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapEdit).

← ***key*** The key of the item to be replaced.

← ***keyLength*** The length of the *key* buffer (in bytes).

← ***data*** The new data item that will replace the previous data.

← ***dataLength*** The length of *data* (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.3.2.11 VDSF_EXPORT int vdsMapStatus (VDS_HANDLE *objectHandle*, vdsObjStatus ∗ *pStatus*)

Return the status of the hash map.

Parameters:

← ***objectHandle*** The handle to the hash map (see vdsMapOpen or vdsMapEdit).

→ ***pStatus*** A pointer to the status structure.

Returns:

0 on success or a vdsErrors on error.

5.4 API functions for vdsf processes.

Functions

- VDSF_EXPORT void vdsExit ()

  This function terminates all access to the VDS.

- VDSF_EXPORT int vdsInit (const char ∗wdAddress, int protectionNeeded)

  This function initializes access to a VDS.

5.4.1 Function Documentation

5.4.1.1 VDSF_EXPORT void vdsExit ()

This function terminates all access to the VDS.

This function will also close all sessions and terminate all accesses to the different objects.

This function takes no argument and always end successfully (even if called twice or if vdsInit was not called).

5.4.1.2 VDSF_EXPORT int vdsInit (const char ∗ *wdAddress*, int *protectionNeeded*)

This function initializes access to a VDS.

It takes 2 input arguments, the address of the watchdog and an integer (used as a boolean, 0 for false, 1 for true) to indicate if sessions and other objects (Queues, etc) are shared amongst threads (in the current process) and must be protected. Recommendation: always set protectionNeeded to 0 (false) unless you cannot do otherwise. In other words it is recommended to use one session handle for each thread. Also if the same queue needs to be accessed by two threads it is more efficient to have two different handles instead of sharing a single one.

[Additional note: API objects (or C handles) are just proxies for the real objects sitting in shared memory. Proper synchronization is already done in shared memory and it is best to avoid to synchronize these proxy objects.]

Upon successful completion, the process handle is set. Otherwise the error code is returned.

Parameters:

← ***wdAddress*** The address of the watchdog. Currently a string with the port number ("12345").

← ***protectionNeeded*** A boolean value indicating if multi-threaded locks are needed or not.

Returns:

0 on success or a vdsErrors on error.

5.5 API functions for vdsf FIFO queues.

5.5.1 Detailed Description

A reminder: FIFO, First In First Out.

Data items are placed at the end of the queue and retrieved from the beginning of the queue.

Functions

- VDSF_EXPORT int vdsQueueClose (VDS_HANDLE objectHandle)

  Close a FIFO queue.

- VDSF_EXPORT int vdsQueueDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

  Retrieve the data definition of the queue.

- VDSF_EXPORT int vdsQueueGetFirst (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Iterate through the queue - no data items are removed from the queue by this function.

- VDSF_EXPORT int vdsQueueGetNext (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Iterate through the queue - no data items are removed from the queue by this function.

- VDSF_EXPORT int vdsQueueOpen (VDS_HANDLE sessionHandle, const char ∗queueName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing FIFO queue (see vdsCreateObject to create a new queue).

- VDSF_EXPORT int vdsQueuePop (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Remove the first item from the beginning of a FIFO queue and return it to the caller.

- VDSF_EXPORT int vdsQueuePush (VDS_HANDLE objectHandle, const void ∗pItem, size_t length)

Insert a data element at the end of the FIFO queue.

- VDSF_EXPORT int vdsQueueStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the queue.

5.5.2 Function Documentation

5.5.2.1 VDSF_EXPORT int vdsQueueClose (VDS_HANDLE *objectHandle*)

Close a FIFO queue.

This function terminates the current access to the queue in shared memory (the queue itself is untouched).

Warning:

Closing an object does not automatically commit or rollback data items that were inserted or removed. You still must use either vdsCommit or vdsRollback to end the current unit of work.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

Returns:

0 on success or a vdsErrors on error.

5.5.2.2 VDSF_EXPORT int vdsQueueDefinition (VDS_HANDLE *objectHandle*, vdsObjectDefinition ∗∗ *definition*)

Retrieve the data definition of the queue.

Warning:

This function allocates a buffer to hold the definition (using malloc()). You must free it (with free()) when you no longer need the definition.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

→ ***definition*** The buffer allocated by the API to hold the content of the object definition. Freeing the memory (with free()) is the responsability of the caller.

Returns:

0 on success or a vdsErrors on error.

5.5.2.3 VDSF_EXPORT int vdsQueueGetFirst (VDS_HANDLE *objectHandle*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *returnedLength*)

Iterate through the queue - no data items are removed from the queue by this function.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

→ ***buffer*** The buffer provided by the user to hold the content of the first element. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***returnedLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.5.2.4 VDSF_EXPORT int vdsQueueGetNext (VDS_HANDLE *objectHandle*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *returnedLength*)

Iterate through the queue - no data items are removed from the queue by this function.

Data items which were added by another session and are not yet committed will not be seen by the iterator. Likewise, destroyed data items (even if not yet committed) are invisible.

Evidently, you must call vdsQueueGetFirst to initialize the iterator. Not so evident - calling vdsQueuePop will reset the iteration to the last element (they use the same internal storage). If this cause a problem, please let us know.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

→ ***buffer*** The buffer provided by the user to hold the content of the next element. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***returnedLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.5.2.5 VDSF_EXPORT int vdsQueueOpen (VDS_HANDLE *sessionHandle*, const char ∗ *queueName*, size_t *nameLengthInBytes*, VDS_HANDLE ∗ *objectHandle*)

Open an existing FIFO queue (see vdsCreateObject to create a new queue).

Parameters:

← ***sessionHandle*** The handle to the current session.

← ***queueName*** The fully qualified name of the queue.

← ***nameLengthInBytes*** The length of *queueName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

→ ***objectHandle*** The handle to the queue, allowing us access to the queue in shared memory. On error, this handle will be set to zero (NULL) unless the objectHandle pointer itself is NULL.

Returns:

0 on success or a vdsErrors on error.

5.5.2.6 VDSF_EXPORT int vdsQueuePop (VDS_HANDLE *objectHandle*, void ∗ *buffer*, size_t *bufferLength*, size_t ∗ *returnedLength*)

Remove the first item from the beginning of a FIFO queue and return it to the caller.

Data items which were added by another session and are not yet committed will not be seen by this function. Likewise, destroyed data items (even if not yet committed) are invisible.

The removals only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

→ ***buffer*** The buffer provided by the user to hold the content of the data item. Memory allocation for this buffer is the responsability of the caller.

← ***bufferLength*** The length of *buffer* (in bytes).

→ ***returnedLength*** The actual number of bytes in the data item.

Returns:

0 on success or a vdsErrors on error.

5.5.2.7 VDSF_EXPORT int vdsQueuePush (VDS_HANDLE *objectHandle*, const void ∗ *pItem*, size_t *length*)

Insert a data element at the end of the FIFO queue.

The additions only become permanent after a call to vdsCommit.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

← ***pItem*** The data item to be inserted.

← ***length*** The length of *pItem* (in bytes).

Returns:

0 on success or a vdsErrors on error.

5.5.2.8 VDSF_EXPORT int vdsQueueStatus (VDS_HANDLE *objectHandle*, vdsObjStatus ∗ *pStatus*)

Return the status of the queue.

Parameters:

← ***objectHandle*** The handle to the queue (see vdsQueueOpen).

→ ***pStatus*** A pointer to the status structure.

Returns:

0 on success or a vdsErrors on error.

5.6 API functions for vdsf sessions.

Functions

- VDSF_EXPORT int vdsCommit (VDS_HANDLE sessionHandle)

  Commit all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

- VDSF_EXPORT int vdsCreateObject (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjectDefinition ∗pDefinition)

  Create a new object in shared memory.

- VDSF_EXPORT int vdsDestroyObject (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes)

Destroy an existing object in shared memory.

- VDSF_EXPORT int vdsErrorMsg (VDS_HANDLE sessionHandle, char ∗message, size_t msgLengthInBytes)

  Return the error message associated with the last error(s).

- VDSF_EXPORT int vdsExitSession (VDS_HANDLE sessionHandle)

  Terminate the current session.

- VDSF_EXPORT int vdsGetInfo (VDS_HANDLE sessionHandle, vdsInfo ∗pInfo)

  Return information on the current status of the VDS (Virtual Data Space).

- VDSF_EXPORT int vdsGetStatus (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjStatus ∗pStatus)

  Return the status of the named object.

- VDSF_EXPORT int vdsInitSession (VDS_HANDLE ∗sessionHandle)

  This function initializes a session.

- VDSF_EXPORT int vdsLastError (VDS_HANDLE sessionHandle)

  Return the last error seen in previous calls (of the current session).

- VDSF_EXPORT int vdsRollback (VDS_HANDLE sessionHandle)

  Rollback all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

5.6.1 Function Documentation

5.6.1.1 VDSF_EXPORT int vdsCommit (VDS_HANDLE *sessionHandle*)

Commit all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

Insertions and deletions subjected to this call include both data items inserted and deleted from data containers (maps, etc.) and objects themselves created with vdsCreateObj and/or destroyed with vdsDestroyObj.

Note: the internal calls executed by the engine to satisfy this request cannot fail. As such, you cannot find yourself with an ugly situation where some operations were committed and others not. If an error is returned by this function, nothing was committed.

Parameters:

← ***sessionHandle*** Handle to the current session.

Returns:

0 on success or a vdsErrors on error.

5.6.1.2 VDSF_EXPORT int vdsCreateObject (VDS_HANDLE *sessionHandle*, const char ∗ *objectName*, size_t *nameLengthInBytes*, vdsObjectDefinition ∗ *pDefinition*)

Create a new object in shared memory.

The creation of the object only becomes permanent after a call to vdsCommit.

This function does not provide a handle to the newly created object. Use vdsQueueOpen and similar functions to get the handle.

Parameters:

← ***sessionHandle*** Handle to the current session.

← ***objectName*** The fully qualified name of the object.

← ***nameLengthInBytes*** The length of *objectName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

← ***pDefinition*** The type of object to create (folder, queue, etc.) and the optional definitions (as needed).

Returns:

0 on success or a vdsErrors on error.

5.6.1.3 VDSF_EXPORT int vdsDestroyObject (VDS_HANDLE *sessionHandle*, const char ∗ *objectName*, size_t *nameLengthInBytes*)

Destroy an existing object in shared memory.

The destruction of the object only becomes permanent after a call to vdsCommit.

Parameters:

← ***sessionHandle*** Handle to the current session.

← ***objectName*** The fully qualified name of the object.

← ***nameLengthInBytes*** The length of *objectName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

Returns:

0 on success or a vdsErrors on error.

5.6.1.4 VDSF_EXPORT int vdsErrorMsg (VDS_HANDLE *sessionHandle*, char ∗ *message*, size_t *msgLengthInBytes*)

Return the error message associated with the last error(s).

If the length of the error message is greater than the length of the provided buffer, the error message will be truncated to fit in the provided buffer.

Caveat, some basic errors cannot be captured, if the provided handles (session handles or object handles) are incorrect (NULL, for example). Without a proper handle, the code cannot know where to store the error...

Parameters:

← ***sessionHandle*** Handle to the current session.

→ ***message*** Buffer for the error message. Memory allocation for this buffer is the responsability of the caller.

← ***msgLengthInBytes*** The length of *message* (in bytes). Must be at least 32 bytes.

Returns:

0 on success or a vdsErrors on error.

5.6.1.5 VDSF_EXPORT int vdsExitSession (VDS_HANDLE *sessionHandle*)

Terminate the current session.

An implicit call to vdsRollback is executed by this function.

Once this function is executed, attempts to use the session handle might lead to memory violation (and, possibly, crashes).

Parameters:

← ***sessionHandle*** Handle to the current session.

Returns:

0 on success or a vdsErrors on error.

5.6.1.6 VDSF_EXPORT int vdsGetInfo (VDS_HANDLE *sessionHandle*, vdsInfo ∗ *pInfo*)

Return information on the current status of the VDS (Virtual Data Space).

The fetched information is mainly about the current status of the memory allocator.

Parameters:

← ***sessionHandle*** Handle to the current session.

→ ***pInfo*** A pointer to the vdsInfo structure.

Returns:

0 on success or a vdsErrors on error.

5.6.1.7 VDSF_EXPORT int vdsGetStatus (VDS_HANDLE *sessionHandle*, const char ∗ *objectName*, size_t *nameLengthInBytes*, vdsObjStatus ∗ *pStatus*)

Return the status of the named object.

Parameters:

← ***sessionHandle*** Handle to the current session.

← ***objectName*** The fully qualified name of the object.

← ***nameLengthInBytes*** The length of *objectName* (in bytes) not counting the null terminator (null-terminators are not used by the vdsf engine).

→ ***pStatus*** A pointer to the vdsObjStatus structure.

Returns:

0 on success or a vdsErrors on error.

5.6.1.8 VDSF_EXPORT int vdsInitSession (VDS_HANDLE ∗ *sessionHandle*)

This function initializes a session.

It takes one output argument, the session handle.

Upon successful completion, the session handle is set and the function returns zero. Otherwise the error code is returned and the handle is set to NULL.

This function will also initiate a new transaction.

Upon normal termination, the current transaction is rolled back. You MUST explicitly call vdseCommit to save your changes.

Parameters:

→ ***sessionHandle*** The handle to the newly created session.

Returns:

0 on success or a vdsErrors on error.

5.6.1.9 VDSF_EXPORT int vdsLastError (VDS_HANDLE *sessionHandle*)

Return the last error seen in previous calls (of the current session).

Caveat, some basic errors cannot be captured, if the provided handles (session handles or object handles) are incorrect (NULL, for example). Without a proper handle, the code cannot know where to store the error...

Parameters:

← ***sessionHandle*** Handle to the current session.

Returns:

The last error.

5.6.1.10 VDSF_EXPORT int vdsRollback (VDS_HANDLE *sessionHandle*)

Rollback all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

Insertions and deletions subjected to this call include both data items inserted and deleted from data containers (maps, etc.) and objects themselves created with vds-CreateObj and/or destroyed with vdsDestroyObj.

Note: the internal calls executed by the engine to satisfy this request cannot fail. As such, you cannot find yourself with an ugly situation where some operations were roll-backed and others not. If an error is returned by this function, nothing was rollbacked.

Parameters:

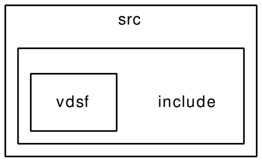
← ***sessionHandle*** Handle to the current session.

Returns:

0 on success or a vdsErrors on error.

6 vdsf API Directory Documentation

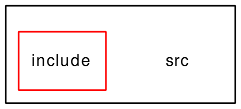
6.1 /home/project/VDSF/vdsf/trunk/src/include/ Directory Reference



Directories

- directory vdsf

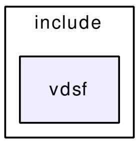
6.2 /home/project/VDSF/vdsf/trunk/src/ Directory Reference



Directories

- directory include

6.3 /home/project/VDSF/vdsf/trunk/src/include/vdsf/ Directory Reference



Files

- file vds.h
- file vdsCommon.h
- file vdsErrors.h
- file vdsFolder.h

  This file provides the API needed to access a VDSF folder.

- file vdsHashMap.h

  This file provides the API needed to access a VDSF hash map.

- file vdsMap.h

  This file provides the API needed to access read-only VDSF hash maps.

- file vdsProcess.h

  This file provides the API functions for vdsf processes.

- file vdsQueue.h

  This file provides the API needed to access a VDSF FIFO queue.

- file vdsSession.h

  This file provides the API needed to create and use a session.

7 vdsf API Data Structure Documentation

7.1 vdsFieldDefinition Struct Reference

```
#include <vdsCommon.h>
```

7.1.1 Detailed Description

Description of the structure of the data (if any).

This structure is aligned in such a way that you can do:

malloc(offsetof(vdsObjectDefinition, fields) + numFields $*$ sizeof(vdsFieldDefinition));

Data Fields

- char name [VDS_MAX_FIELD_LENGTH]

- enum vdsFieldType type
- size_t length
- size_t minLength
- size_t maxLength
- size_t precision
- size_t scale

7.1.2 Field Documentation

7.1.2.1 char vdsFieldDefinition::name[VDS_MAX_FIELD_LENGTH]

7.1.2.2 enum vdsFieldType vdsFieldDefinition::type

7.1.2.3 size_t vdsFieldDefinition::length

7.1.2.4 size_t vdsFieldDefinition::minLength

7.1.2.5 size_t vdsFieldDefinition::maxLength

7.1.2.6 size_t vdsFieldDefinition::precision

7.1.2.7 size_t vdsFieldDefinition::scale

The documentation for this struct was generated from the following file:

- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

7.2 vdsFolderEntry Struct Reference

```
#include <vdsCommon.h>
```

7.2.1 Detailed Description

This data structure is used to iterate throught all objects in a folder.

Note: the actual name of an object (and the length of this name) might vary if you are using different locales (internally, names are stored as wide characters (4 bytes)).

Data Fields

- vdsObjectType type

  The object type.

- int status

  Status (created but not committed, etc.

- size_t nameLengthInBytes

  The actual length of the name of the object.

- char name [VDS_MAX_NAME_LENGTH]

  The name of the object.

7.2.2 Field Documentation

7.2.2.1 vdsObjectType vdsFolderEntry::type

The object type.

7.2.2.2 int vdsFolderEntry::status

Status (created but not committed, etc.

) - not used in version 0.1

7.2.2.3 size_t vdsFolderEntry::nameLengthInBytes

The actual length of the name of the object.

7.2.2.4 char vdsFolderEntry::name[VDS_MAX_NAME_LENGTH]

The name of the object.

The documentation for this struct was generated from the following file:

- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

7.3 vdsInfo Struct Reference

```
#include <vdsCommon.h>
```

7.3.1 Detailed Description

This data structure is used to retrieve the status of the virtual data space.

Data Fields

- size_t totalSizeInBytes

  Total size of the virtual data space.

- size_t allocatedSizeInBytes

  Total size of the allocated blocks.

- size_t numObjects

  Number of API objects in the vds (internal objects are not counted).

- size_t numGroups

  Total number of groups of blocks.

- size_t numMallocs

  Number of calls to allocate groups of blocks.

- size_t numFrees

  Number of calls to free groups of blocks.

- size_t largestFreeInBytes

  Largest contiguous group of free blocks.

7.3.2 Field Documentation

7.3.2.1 size_t vdsInfo::totalSizeInBytes

Total size of the virtual data space.

7.3.2.2 size_t vdsInfo::allocatedSizeInBytes

Total size of the allocated blocks.

7.3.2.3 size_t vdsInfo::numObjects

Number of API objects in the vds (internal objects are not counted).

7.3.2.4 size_t vdsInfo::numGroups

Total number of groups of blocks.

7.3.2.5 size_t vdsInfo::numMallocs

Number of calls to allocate groups of blocks.

7.3.2.6 size_t vdsInfo::numFrees

Number of calls to free groups of blocks.

7.3.2.7 size_t vdsInfo::largestFreeInBytes

Largest contiguous group of free blocks.

The documentation for this struct was generated from the following file:

- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

7.4 vdsKeyDefinition Struct Reference

```
#include <vdsCommon.h>
```

7.4.1 Detailed Description

Description of the structure of the hash map key.

Data Fields

- enum vdsKeyType type
- size_t length
- size_t minLength
- size_t maxLength

7.4.2 Field Documentation

7.4.2.1 enum vdsKeyType vdsKeyDefinition::type

7.4.2.2 size_t vdsKeyDefinition::length

7.4.2.3 size_t vdsKeyDefinition::minLength

7.4.2.4 size_t vdsKeyDefinition::maxLength

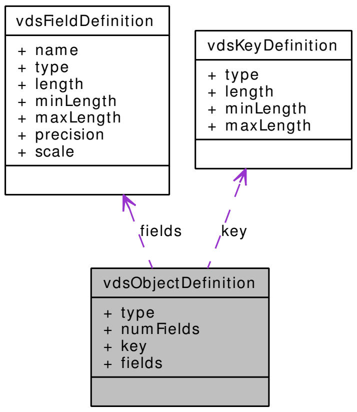
The documentation for this struct was generated from the following file:

- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

7.5 vdsObjectDefinition Struct Reference

```
#include <vdsCommon.h>
```

Collaboration diagram for vdsObjectDefinition:



7.5.1 Detailed Description

This struct has a variable length.

Data Fields

- enum vdsObjectType type
- unsigned int numFields
- vdsKeyDefinition key

  The data definition of the key (hash map only).

- vdsFieldDefinition fields [1]

  The data definition of the fields.

7.5.2 Field Documentation

7.5.2.1 enum vdsObjectType vdsObjectDefinition::type

7.5.2.2 unsigned int vdsObjectDefinition::numFields

7.5.2.3 vdsKeyDefinition vdsObjectDefinition::key

The data definition of the key (hash map only).

7.5.2.4 vdsFieldDefinition vdsObjectDefinition::fields[1]

The data definition of the fields.

The documentation for this struct was generated from the following file:

- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

7.6 vdsObjStatus Struct Reference

```
#include <vdsCommon.h>
```

7.6.1 Detailed Description

This data structure is used to retrieve the status of objects.

Data Fields

- vdsObjectType type

  The object type.

- int status

  Status (created but not committed, etc.

- size_t numBlocks

  The number of blocks allocated to this object.

- size_t numBlockGroup

  The number of groups of blocks allocated to this object.

- size_t numDataItem

  The number of data items in thisa object.

- size_t freeBytes

  The amount of free space available in the blocks allocated to this object.

- size_t maxDataLength

  Maximum data length (in bytes).

- size_t maxKeyLength

  Maximum key length (in bytes) if keys are supported - zero otherwise.

7.6.2 Field Documentation

7.6.2.1 vdsObjectType vdsObjStatus::type

The object type.

7.6.2.2 int vdsObjStatus::status

Status (created but not committed, etc.

) - not used in version 0.1

7.6.2.3 size_t vdsObjStatus::numBlocks

The number of blocks allocated to this object.

7.6.2.4 size_t vdsObjStatus::numBlockGroup

The number of groups of blocks allocated to this object.

7.6.2.5 size_t vdsObjStatus::numDataItem

The number of data items in thisa object.

7.6.2.6 size_t vdsObjStatus::freeBytes

The amount of free space available in the blocks allocated to this object.

7.6.2.7 size_t vdsObjStatus::maxDataLength

Maximum data length (in bytes).

7.6.2.8 size_t vdsObjStatus::maxKeyLength

Maximum key length (in bytes) if keys are supported - zero otherwise.

The documentation for this struct was generated from the following file:

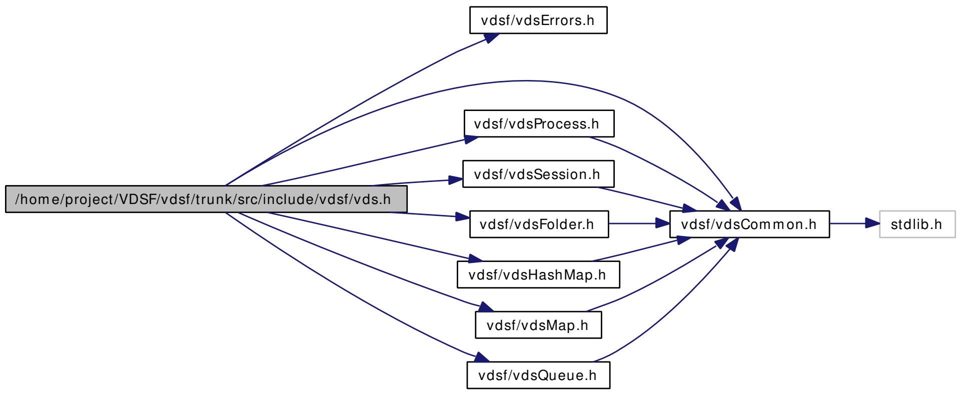
- /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h

8 vdsf API File Documentation

8.1 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h File Reference

```
#include <vdsf/vdsErrors.h>
#include <vdsf/vdsCommon.h>
#include <vdsf/vdsProcess.h>
#include <vdsf/vdsSession.h>
#include <vdsf/vdsFolder.h>
#include <vdsf/vdsHashMap.h>
#include <vdsf/vdsMap.h>
#include <vdsf/vdsQueue.h>
```

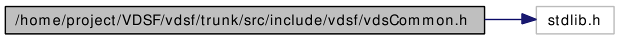
Include dependency graph for vds.h:



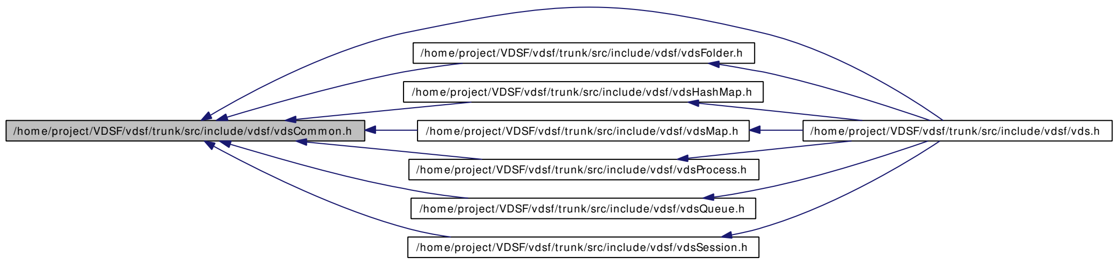
8.2 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vds-Common.h File Reference

`#include <stdlib.h>`

Include dependency graph for vdsCommon.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsFolder.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsHashMap.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsCommon.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsMap.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsProcess.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsQueue.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsSession.h

Data Structures

- struct vdsKeyDefinition

  Description of the structure of the hash map key.

- struct vdsFieldDefinition

  Description of the structure of the data (if any).

- struct vdsObjectDefinition

  This struct has a variable length.

- struct vdsFolderEntry

  This data structure is used to iterate throught all objects in a folder.

- struct vdsObjStatus

  This data structure is used to retrieve the status of objects.

- struct vdsInfo

  This data structure is used to retrieve the status of the virtual data space.

Defines

- #define VDSF_EXPORT

  Uses to tell the VC++ compiler to export/import a function or variable on Windows (the macro is empty on other platforms).

- #define VDS_MAX_NAME_LENGTH 256

  Maximum number of bytes of the name of a vds object (not counting the name of the parent folder(s)).

- #define VDS_MAX_FULL_NAME_LENGTH 1024

  Maximum number of bytes of the fully qualified name of a vds object (including the name(s) of its parent folder(s)).

- #define VDS_MAX_FIELD_LENGTH 32

  Maximum number of bytes of the name of a field of a vds object.

- #define VDS_MAX_FIELDS 65535

  Maximum number of fields (including the last one).

- #define VDS_FIELD_MAX_PRECISION 30

Typedefs

- typedef void ∗ VDS_HANDLE

  VDS_HANDLE is an opaque data type used by the C API to reference objects created in the API module.

- typedef enum vdsObjectType vdsObjectType
- typedef enum vdsIteratorType vdsIteratorType
- typedef vdsKeyDefinition vdsKeyDefinition
- typedef vdsFieldDefinition vdsFieldDefinition
- typedef vdsObjectDefinition vdsObjectDefinition
- typedef vdsFolderEntry vdsFolderEntry
- typedef vdsObjStatus vdsObjStatus
- typedef vdsInfo vdsInfo

Enumerations

- enum vdsObjectType {
  VDS_FOLDER = 1, VDS_QUEUE = 2, VDS_HASH_MAP = 3, VDS_MAP = 4,
  VDS_LAST_OBJECT_TYPE }

The object type as seen from the API.

- enum vdsIteratorType { VDS_FIRST = 1, VDS_NEXT = 2 }
- enum vdsFieldType {

  VDS_INTEGER = 1, VDS_BINARY, VDS_STRING, VDS_DECIMAL,

  VDS_BOOLEAN, VDS_VAR_BINARY, VDS_VAR_STRING }

  VDSF supported data types.

- enum vdsKeyType {

  VDS_KEY_INTEGER = 101, VDS_KEY_BINARY, VDS_KEY_STRING, VDS_KEY_VAR_BINARY,

  VDS_KEY_VAR_STRING }

  VDSF supported data types for keys.

8.2.1 Define Documentation

8.2.1.1 #define VDS_FIELD_MAX_PRECISION 30

8.2.1.2 #define VDS_MAX_FIELD_LENGTH 32

Maximum number of bytes of the name of a field of a vds object.

8.2.1.3 #define VDS_MAX_FIELDS 65535

Maximum number of fields (including the last one).

8.2.1.4 #define VDS_MAX_FULL_NAME_LENGTH 1024

Maximum number of bytes of the fully qualified name of a vds object (including the name(s) of its parent folder(s)).

Note: setting this value eliminates a possible loophole since some heap memory must be allocated to hold the wide characters string for the duration of the operation (open, close, create or destroy).

8.2.1.5 #define VDS_MAX_NAME_LENGTH 256

Maximum number of bytes of the name of a vds object (not counting the name of the parent folder(s)).

8.2.1.6 #define VDSF_EXPORT

Uses to tell the VC++ compiler to export/import a function or variable on Windows (the macro is empty on other platforms).

8.2.2 Typedef Documentation

8.2.2.1 typedef void* VDS_HANDLE

VDS_HANDLE is an opaque data type used by the C API to reference objects created in the API module.

8.2.2.2 typedef struct vdsFieldDefinition vdsFieldDefinition

8.2.2.3 typedef struct vdsFolderEntry vdsFolderEntry

8.2.2.4 typedef struct vdsInfo vdsInfo

8.2.2.5 typedef enum vdsIteratorType vdsIteratorType

8.2.2.6 typedef struct vdsKeyDefinition vdsKeyDefinition

8.2.2.7 typedef struct vdsObjectDefinition vdsObjectDefinition

8.2.2.8 typedef enum vdsObjectType vdsObjectType

8.2.2.9 typedef struct vdsObjStatus vdsObjStatus

8.2.3 Enumeration Type Documentation

8.2.3.1 enum vdsFieldType

VDSF supported data types.

Enumerator:

VDS_INTEGER

VDS_BINARY

VDS_STRING

VDS_DECIMAL

VDS_BOOLEAN	
VDS_VAR_BINARY	Only valid for the last field of the data definition.
VDS_VAR_STRING	Only valid for the last field of the data definition.

8.2.3.2 enum vdsIteratorType

Enumerator:

VDS_FIRST
VDS_NEXT

8.2.3.3 enum vdsKeyType

VDSF supported data types for keys.

Enumerator:

VDS_KEY_INTEGER	
VDS_KEY_BINARY	
VDS_KEY_STRING	
VDS_KEY_VAR_BINARY	Only valid for the last field of the data definition.
VDS_KEY_VAR_STRING	Only valid for the last field of the data definition.

8.2.3.4 enum vdsObjectType

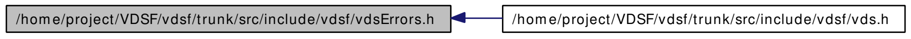
The object type as seen from the API.

Enumerator:

VDS_FOLDER
VDS_QUEUE
VDS_HASH_MAP
VDS_MAP
VDS_LAST_OBJECT_TYPE

8.3 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsErrors.h File Reference

This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsErrors.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Typedefs

- typedef enum vdsErrors vdsErrors

Enumerations

- enum vdsErrors {

  VDS_OK = 0, VDS_INTERNAL_ERROR = 666, VDS_ENGINE_BUSY = 1, VDS_NOT_ENOUGH_VDS_MEMORY = 2,

  VDS_NOT_ENOUGH_HEAP_MEMORY = 3, VDS_NOT_ENOUGH_-RESOURCES = 4, VDS_WRONG_TYPE_HANDLE = 5, VDS_NULL_-HANDLE = 6,

  VDS_NULL_POINTER = 7, VDS_INVALID_LENGTH = 8, VDS_-PROCESS_ALREADY_INITIALIZED = 21, VDS_PROCESS_NOT_-INITIALIZED = 22,

  VDS_INVALID_WATCHDOG_ADDRESS = 23, VDS_INCOMPATIBLE_-VERSIONS = 24, VDS_SOCKET_ERROR = 25, VDS_CONNECT_ERROR = 26,

  VDS_SEND_ERROR = 27, VDS_RECEIVE_ERROR = 28, VDS_-BACKSTORE_FILE_MISSING = 29, VDS_ERROR_OPENING_VDS = 30,

  VDS_LOGFILE_ERROR = 41, VDS_SESSION_CANNOT_GET_LOCK = 42, VDS_SESSION_IS_TERMINATED = 43, VDS_INVALID_OBJECT_NAME = 51,

  VDS_NO_SUCH_OBJECT = 52, VDS_NO_SUCH_FOLDER = 53, VDS_-OBJECT_ALREADY_PRESENT = 54, VDS_IS_EMPTY = 55,

  VDS_WRONG_OBJECT_TYPE = 56, VDS_OBJECT_CANNOT_GET_-LOCK = 57, VDS_REACHED_THE_END = 58, VDS_INVALID_ITERATOR = 59,

  VDS_OBJECT_NAME_TOO_LONG = 60, VDS_FOLDER_IS_NOT_EMPTY = 61, VDS_ITEM_ALREADY_PRESENT = 62, VDS_NO_SUCH_ITEM = 63,

  VDS_OBJECT_IS_DELETED = 64, VDS_OBJECT_NOT_INITIALIZED = 65, VDS_ITEM_IS_IN_USE = 66, VDS_ITEM_IS_DELETED = 67,

  VDS_OBJECT_IS_IN_USE = 69, VDS_OBJECT_IS_READ_ONLY = 70, VDS_INVALID_NUM_FIELDS = 101, VDS_INVALID_FIELD_TYPE = 102,

  VDS_INVALID_FIELD_LENGTH_INT = 103, VDS_INVALID_FIELD_-LENGTH = 104, VDS_INVALID_FIELD_NAME = 105, VDS_DUPLICATE_-FIELD_NAME = 106,

  VDS_INVALID_PRECISION = 107, VDS_INVALID_SCALE = 108, VDS_-INVALID_KEY_DEF = 109, VDS_XML_READ_ERROR = 201, VDS_XML_INVALID_ROOT = 202, VDS_XML_NO_SCHEMA_-LOCATION = 203, VDS_XML_PARSER_CONTEXT_FAILED = 204, VDS_XML_PARSE_SCHEMA_FAILED = 205,

VDS_XML_VALID_CONTEXT_FAILED = 206, VDS_XML_-VALIDATION_FAILED = 207 }

8.3.1 Typedef Documentation

8.3.1.1 typedef enum vdsErrors vdsErrors

8.3.2 Enumeration Type Documentation

8.3.2.1 enum vdsErrors

Enumerator:

VDS_OK No error.

..

VDS_INTERNAL_ERROR Abnormal internal error.

It should not happen!

VDS_ENGINE_BUSY Cannot get a lock on a system object, the engine is "busy".

This might be the result of either a very busy system where unused cpu cycles are rare or a lock might be held by a crashed process.

VDS_NOT_ENOUGH_VDS_MEMORY Not enough memory in the VDS.

VDS_NOT_ENOUGH_HEAP_MEMORY Not enough heap memory (non-VDS memory).

VDS_NOT_ENOUGH_RESOURCES There are not enough resources to correctly process the call.

There are not enough resources to correctly process the call. This might be due to a lack of POSIX semaphores on systems where locks are implemented that way or a failure in initializing a pthread_mutex (or on Windows, a critical section).

VDS_WRONG_TYPE_HANDLE The provided handle is of the wrong type (C API).

This could happen if you provide a queue handle to access a hash map or something similar. It can also occur if you try to access an object after closing it.

If you are seeing this error for the C++ API (or some other object-oriented interface), you've just found an internal error... (the handle is encapsulated and cannot be modified using the public interface).

VDS_NULL_HANDLE The provided handle is NULL.

VDS_NULL_POINTER One of the arguments of an API function is an invalid NULL pointer.

VDS_INVALID_LENGTH An invalid length was provided as an argument to an API function.

This invalid length will usually indicate that the length value is set to zero.

VDS_PROCESS_ALREADY_INITIALIZED The process was already initialized.

One possibility: was vdsInit() called for a second time?

VDS_PROCESS_NOT_INITIALIZED The process was not properly initialized.

One possibility: was vdsInit() called?

VDS_INVALID_WATCHDOG_ADDRESS The watchdog address is invalid (empty string, NULL pointer, etc.

).

VDS_INCOMPATIBLE_VERSIONS API - memory-file version mismatch.

VDS_SOCKET_ERROR Generic socket error.

VDS_CONNECT_ERROR Socket error when trying to connect to the watchdog.

VDS_SEND_ERROR Socket error when trying to send a request to the watchdog.

VDS_RECEIVE_ERROR Socket error when trying to receive a reply from the watchdog.

VDS_BACKSTORE_FILE_MISSING The vds backstore file is missing.

The name of this file is provided by the watchdog - if it is missing, something really weird is going on.

VDS_ERROR_OPENING_VDS Generic i/o error when attempting to open the vds.

VDS_LOGFILE_ERROR Error accessing the directory for the log files or error opening the log file itself.

VDS_SESSION_CANNOT_GET_LOCK Cannot get a lock on the session (a pthread_mutex or a critical section on Windows).

VDS_SESSION_IS_TERMINATED An attempt was made to use a session object (a session handle) after this session was terminated.

VDS_INVALID_OBJECT_NAME Permitted characters for names are alphanumerics, spaces (' '), dashes ('-') and underlines ('_').

The first character must be alphanumeric.

VDS_NO_SUCH_OBJECT The object was not found (but its folder does exist).

VDS_NO_SUCH_FOLDER One of the parent folder of an object does not exist.

VDS_OBJECT_ALREADY_PRESENT Attempt to create an object which already exists.

VDS_IS_EMPTY The object (data container) is empty.

VDS_WRONG_OBJECT_TYPE Attempt to create an object of an unknown object type or to open an object of the wrong type.

VDS_OBJECT_CANNOT_GET_LOCK Cannot get lock on the object.

This might be the result of either a very busy system where unused cpu cycles are rare or a lock might be held by a crashed process.

VDS_REACHED_THE_END The search/iteration reached the end without finding a new item/record.

VDS_INVALID_ITERATOR An invalid value was used for a vdsIteratorType parameter.

VDS_OBJECT_NAME_TOO_LONG The name of the object is too long.

The maximum length of a name cannot be more than VDS_MAX_NAME_-LENGTH (or VDS_MAX_FULL_NAME_LENGTH for the fully qualified name).

VDS_FOLDER_IS_NOT_EMPTY You cannot delete a folder if there are still undeleted objects in it.

Technical: a folder does not need to be empty to be deleted but all objects in it must be "marked as deleted" by the current session. This enables writing recursive deletions

VDS_ITEM_ALREADY_PRESENT An item with the same key was found.

VDS_NO_SUCH_ITEM The item was not found in the hash map.

VDS_OBJECT_IS_DELETED The object is scheduled to be deleted soon.

Operations on this data container are not permitted at this time.

VDS_OBJECT_NOT_INITIALIZED Object must be open first before you can access them.

VDS_ITEM_IS_IN_USE The data item is scheduled to be deleted soon or was just created and is not committed.

Operations on this data item are not permitted at this time.

VDS_ITEM_IS_DELETED The data item is scheduled to be deleted soon.

Operations on this data container are not permitted at this time.

VDS_OBJECT_IS_IN_USE The object is scheduled to be deleted soon or was just created and is not committed.

Operations on this object are not permitted at this time.

VDS_OBJECT_IS_READ_ONLY The object is read-only and update operations (delete/insert/replace) on it are not permitted.

at this time.

VDS_INVALID_NUM_FIELDS The number of fields in the data definition is invalid - either zero or greater than VDS_MAX_FIELDS (defined in vdsf/vdsCommon.h).

VDS_INVALID_FIELD_TYPE The data type of the field definition does not correspond to one of the data type defined in the enum vdsFieldType (vdsf/vdsCommon.h).

or you've used VDS_VAR_STRING or VDS_VAR_BINARY at the wrong place.

Do not forget that VDS_VAR_STRING and VDS_VAR_BINAR can only be used for the last field of your data definition.

VDS_INVALID_FIELD_LENGTH_INT The length of an integer field (VDS_-INTEGER) is invalid.

Valid values are 1, 2, 4 and 8.

VDS_INVALID_FIELD_LENGTH The length of a field (string or binary) is invalid.

Valid values are all numbers greater than zero and less than 4294967296 (4 Giga).

VDS_INVALID_FIELD_NAME The name of the field contains invalid characters.

Valid characters are the standard ASCII alphanumerics ([a-zA-Z0-9]) and the underscore ('_'). The first character of the name must be letter.

VDS_DUPLICATE_FIELD_NAME The name of the field is already used by another field in the current definition.

Note: at the moment field names are case sensitive (for example "account_id" and "Account_Id" are considered different). This might be changed eventually so this practice should be avoided.

VDS_INVALID_PRECISION The precision of a VDS_DECIMAL field is either zero or over the limit for this type (set at 30 currently).

Note: precision is the number of digits in a number.

VDS_INVALID_SCALE The scale of a VDS_DECIMAL field is invalid (greater than the value of precision.

Note: scale is the number of digits to the right of the decimal separator in a number.

VDS_INVALID_KEY_DEF The key definition for a hash map is either invalid or missing.

VDS_XML_READ_ERROR Error reading the XML buffer stream.

No validation is done at this point. Therefore the error is likely something like a missing end-tag or some other non-conformance to the XML's syntax rules.

A simple Google search for "well-formed xml" returns many web sites that describe the syntax rules for XML. You can also use the program xmllint (included in the distribution of libxm2) to pinpoint the issue.

VDS_XML_INVALID_ROOT The root element is not the expected root, <folder> and similar.

VDS_XML_NO_SCHEMA_LOCATION The root element must have an attribute named schemaLocation (in the namespace "http://www.w3.org/2001/XMLSchema-instance") to point to the schema use for the xml buffer stream.

This attribute is in two parts separated by a space. The code expects the file name of the schema in the second element of this attribute.

VDS_XML_PARSER_CONTEXT_FAILED The creation of a new schema parser context failed.

There might be multiple reasons for this, for example, a memory-allocation failure in libxml2. However, the most likely reason is that the schema file is not at the location indicated by the attribute schemaLocation of the root element of the buffer stream.

VDS_XML_PARSE_SCHEMA_FAILED The parse operation of the schema failed.

Most likely, there is an error in the schema. To debug this you can use xmllint (part of the libxml2 package).

VDS_XML_VALID_CONTEXT_FAILED The creation of a new schema validation context failed.

There might be multiple reasons for this, for example, a memory-allocation failure in libxml2.

VDS_XML_VALIDATION_FAILED Document validation for the xml buffer failed.

To debug this problem you can use xmllint (part of the libxml2 package).

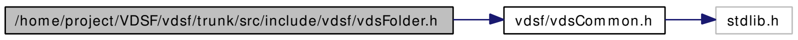
8.4 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsFolder.h File Reference

8.4.1 Detailed Description

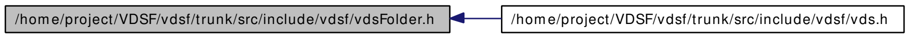
This file provides the API needed to access a VDSF folder.

`#include <vdsf/vdsCommon.h>`

Include dependency graph for vdsFolder.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsFolder.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT int vdsFolderClose (VDS_HANDLE objectHandle)

  Close a folder.

- VDSF_EXPORT int vdsFolderCreateObject (VDS_HANDLE folderHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjectDefinition ∗pDefinition)

  Create a new object in shared memory as a child of the current folder.

- VDSF_EXPORT int vdsFolderCreateObjectXML (VDS_HANDLE folderHandle, const char ∗xmlBuffer, size_t lengthInBytes)

  Create a new object in shared memory as a child of the current folder.

- VDSF_EXPORT int vdsFolderDestroyObject (VDS_HANDLE folderHandle, const char ∗objectName, size_t nameLengthInBytes)

  Destroy an object, child of the current folder, in shared memory.

- VDSF_EXPORT int vdsFolderGetFirst (VDS_HANDLE objectHandle, vdsFolderEntry ∗pEntry)

  Iterate through the folder - no data items are removed from the folder by this function.

- VDSF_EXPORT int vdsFolderGetNext (VDS_HANDLE objectHandle, vdsFolderEntry ∗pEntry)

  Iterate through the folder.

- VDSF_EXPORT int vdsFolderOpen (VDS_HANDLE sessionHandle, const char ∗folderName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing folder (see vdsCreateObject to create a new folder).

- VDSF_EXPORT int vdsFolderStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the folder.

8.5 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsHashMap.h File Reference

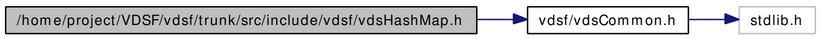
8.5.1 Detailed Description

This file provides the API needed to access a VDSF hash map.

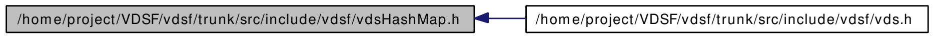
Hash maps use unique keys - the data items are not sorted.

```
#include <vdsf/vdsCommon.h>
```

Include dependency graph for vdsHashMap.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsHashMap.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT int vdsHashMapClose (VDS_HANDLE objectHandle)

  Close a Hash Map.

- VDSF_EXPORT int vdsHashMapDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

  Retrieve the data definition of the hash map.

- VDSF_EXPORT int vdsHashMapDelete (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength)

  Remove the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsHashMapGet (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗returned-Length)

  Retrieve the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsHashMapGetFirst (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsHashMapGetNext (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

  Iterate through the hash map.

- VDSF_EXPORT int vdsHashMapInsert (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Insert a data element in the hash map.

- VDSF_EXPORT int vdsHashMapOpen (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing hash map (see vdsCreateObject to create a new object).

- VDSF_EXPORT int vdsHashMapReplace (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Replace a data element in the hash map.

- VDSF_EXPORT int vdsHashMapStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the hash map.

8.6 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsMap.h File Reference

8.6.1 Detailed Description

This file provides the API needed to access read-only VDSF hash maps.

The features are very similar to the ordinary hash maps except that no locks are require to access the data and special procedures are implemented for the occasional updates:

1) when a map is open in read-only mode (vdsMapOpen(), the end-of-this-unit-of-work calls (vdsCommit/vdsRollback) will check if a new version of the map exits and if indeed this is the case, the new version will be use instead of the old one.

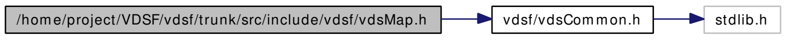
2) when a map is open for editing a working copy of the map is created in shared memory and the map can be updated (no locks again since only the updater can access the working copy). When the session is committed, the working version becomes the latest version and can be open/accessed by readers. And, of course, the same procedure applies if you have a set of maps that must be changed together.

If vdsRollback is called, all changes done to the working copy are erased.

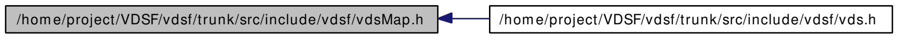
Note: the old versions are removed from memory when all readers have updated their versions. Even if a program is only doing read access to the VDS data, it is important to add vdsCommit() once in a while to refresh the "handles" if the program is running for a while.

```
#include <vdsf/vdsCommon.h>
```

Include dependency graph for vdsMap.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsMap.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT int vdsMapClose (VDS_HANDLE objectHandle)

    Close a Hash Map.

- VDSF_EXPORT int vdsMapDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

    Retrieve the data definition of the hash map.

- VDSF_EXPORT int vdsMapDelete (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength)

    Remove the data item identified by the given key from the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapEdit (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

    Open a temporary copy of an existing hash map for editing.

- VDSF_EXPORT int vdsMapGet (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

    Retrieve the data item identified by the given key from the hash map.

- VDSF_EXPORT int vdsMapGetFirst (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

    Iterate through the hash map.

- VDSF_EXPORT int vdsMapGetNext (VDS_HANDLE objectHandle, void ∗key, size_t keyLength, void ∗buffer, size_t bufferLength, size_t ∗retKeyLength, size_t ∗retDataLength)

    Iterate through the hash map.

- VDSF_EXPORT int vdsMapInsert (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

    Insert a data element in the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapOpen (VDS_HANDLE sessionHandle, const char ∗hashMapName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing hash map read only (see vdsCreateObject to create a new object).

- VDSF_EXPORT int vdsMapReplace (VDS_HANDLE objectHandle, const void ∗key, size_t keyLength, const void ∗data, size_t dataLength)

  Replace a data element in the hash map (you must be in edit mode).

- VDSF_EXPORT int vdsMapStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the hash map.

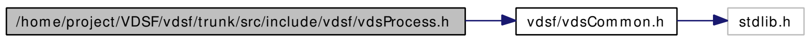
8.7 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsProcess.h File Reference

8.7.1 Detailed Description

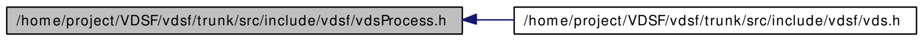
This file provides the API functions for vdsf processes.

```
#include <vdsf/vdsCommon.h>
```

Include dependency graph for vdsProcess.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsProcess.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT void vdsExit ()

  This function terminates all access to the VDS.

- VDSF_EXPORT int vdsInit (const char ∗wdAddress, int protectionNeeded)

  This function initializes access to a VDS.

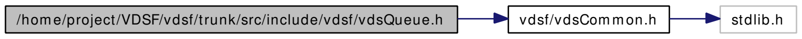
8.8 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsQueue.h File Reference

8.8.1 Detailed Description

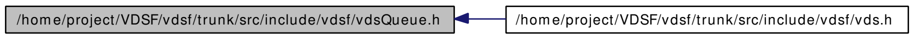
This file provides the API needed to access a VDSF FIFO queue.

`#include <vdsf/vdsCommon.h>`

Include dependency graph for vdsQueue.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsQueue.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT int vdsQueueClose (VDS_HANDLE objectHandle)

  Close a FIFO queue.

- VDSF_EXPORT int vdsQueueDefinition (VDS_HANDLE objectHandle, vdsObjectDefinition ∗∗definition)

  Retrieve the data definition of the queue.

- VDSF_EXPORT int vdsQueueGetFirst (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Iterate through the queue - no data items are removed from the queue by this function.

- VDSF_EXPORT int vdsQueueGetNext (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Iterate through the queue - no data items are removed from the queue by this function.

- VDSF_EXPORT int vdsQueueOpen (VDS_HANDLE sessionHandle, const char ∗queueName, size_t nameLengthInBytes, VDS_HANDLE ∗objectHandle)

  Open an existing FIFO queue (see vdsCreateObject to create a new queue).

- VDSF_EXPORT int vdsQueuePop (VDS_HANDLE objectHandle, void ∗buffer, size_t bufferLength, size_t ∗returnedLength)

  Remove the first item from the beginning of a FIFO queue and return it to the caller.

- VDSF_EXPORT int vdsQueuePush (VDS_HANDLE objectHandle, const void ∗pItem, size_t length)

  Insert a data element at the end of the FIFO queue.

- VDSF_EXPORT int vdsQueueStatus (VDS_HANDLE objectHandle, vdsObjStatus ∗pStatus)

  Return the status of the queue.

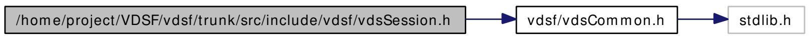
8.9 /home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsSession.h File Reference

8.9.1 Detailed Description

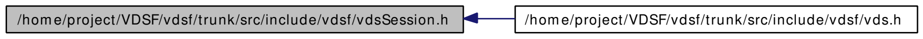
This file provides the API needed to create and use a session.

```
#include <vdsf/vdsCommon.h>
```

Include dependency graph for vdsSession.h:



This graph shows which files directly or indirectly include this file:


/home/project/VDSF/vdsf/trunk/src/include/vdsf/vdsSession.h
/home/project/VDSF/vdsf/trunk/src/include/vdsf/vds.h

Functions

- VDSF_EXPORT int vdsCommit (VDS_HANDLE sessionHandle)

  Commit all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

- VDSF_EXPORT int vdsCreateObject (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjectDefinition ∗pDefinition)

  Create a new object in shared memory.

- VDSF_EXPORT int vdsDestroyObject (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes)

  Destroy an existing object in shared memory.

- VDSF_EXPORT int vdsErrorMsg (VDS_HANDLE sessionHandle, char ∗message, size_t msgLengthInBytes)

  Return the error message associated with the last error(s).

- VDSF_EXPORT int vdsExitSession (VDS_HANDLE sessionHandle)

  Terminate the current session.

- VDSF_EXPORT int vdsGetInfo (VDS_HANDLE sessionHandle, vdsInfo ∗pInfo)

  Return information on the current status of the VDS (Virtual Data Space).

- VDSF_EXPORT int vdsGetStatus (VDS_HANDLE sessionHandle, const char ∗objectName, size_t nameLengthInBytes, vdsObjStatus ∗pStatus)

  Return the status of the named object.

- VDSF_EXPORT int vdsInitSession (VDS_HANDLE ∗sessionHandle)

  This function initializes a session.

- VDSF_EXPORT int vdsLastError (VDS_HANDLE sessionHandle)

  Return the last error seen in previous calls (of the current session).

- VDSF_EXPORT int vdsRollback (VDS_HANDLE sessionHandle)

  Rollback all insertions and deletions (of the current session) executed since the previous call to vdsCommit or vdsRollback.

Index