Matisse®
C API Reference

January 2017

Matisse C API Reference

PDF generated 7 January 2017

Contents

1 Functions by Themes

1.1 Database Connection

Session

The following list describes the necessary steps required to access a database. It is important to adhere closely to the following sequence of actions when accessing the database:

- Allocate a connection structure
- Set the connection options
- Connect to a database
- Select the connection
- Open a transaction or select a version
- Launch the operations on the database
- Commit or abort the transaction or end the version access
- Deselect the connection
- Disconnect
- Deallocate the connection structure

A single client application may provide several databases. The user must open one connection per database.

Once the database connection is selected, the client has direct access to the data, either within a transaction or within a version access. Within a transaction, the client can modify data. Within a version access, the client can only read data.

You can set an explicit lock on objects within a transaction.

Several databases can be operated simultaneously. For example, you can start a transaction on database A, select a database B then work on B, then return to A.

You must adhere to the following guideline:

- The objects of a given database cannot reference those of another database. If this situation occurs Matisse may generate an error. However, if the referenced object is any object of the current database, no error is generated.

Summary

- Connections to several databases can be opened simultaneously.
- The database connection must be selected in order to have direct access to the data. Access to data occurs either during a transaction or from within a version access.

- Before disconnecting from the database, `the current connection` must be deselected.

List of Functions

```
MtSTS MtAllocateContext (MtContext *connection)
MtSTS MtSetConnectionOption (MtContext connection,
                             MtConnectionOption opt, ...)
MtSTS MtGetConnectionOption (MtContext connection,
                             MtConnectionOption opt, ...)
MtSTS MtConnectDatabase (MtContext connection,
                         MtString host,
                         MtString databaseName,
                         MtString userName,
                         MtString password)
MtSTS MtDisconnectDatabase (MtContext connection)
```

1.2 Database Access

Transaction

A transaction is the smallest granularity operation on a database. It is atomic: all the elements of the transaction either succeed or fail. If they fail, the transaction is aborted. A transaction abort may be initiated by the server or by the user.

Access to the database depends on a wait-time parameter.

This parameter is set on a connection by calling `MtCtxSetConnectionOption`. It can be set at any time.

Within a transaction, access to the database may be blocked for the following reasons:

- If competing transactions mutually prohibit access (deadlock), one of the transactions is aborted (depending on transaction priority) and the cache is flushed.
- If the transaction is blocked for a period longer than the wait-time or if a Matisse error occurs, an error status is initiated.

When exiting a transaction, the cache is flushed: all objects read into client memory during the transaction are deleted and all locks on these objects are released.

A transaction is associated with only a single connection.

The number of locks created is proportional to the number of objects a transaction modifies. Therefore, transactions modifying objects should be as short as possible to avoid affecting other users.

Version Access

The Matisse server manages the successive versions of modified objects.

Matisse allows access to any version previously saved.

Through a saved version, you can work on a consistent view of the database.

Any value read within a version access is deleted from the cache when the access is terminated.

Through this mechanism, the user can read objects or values outside a transaction context without conflicting with another user.

A version access is performed within the function `MtCtxStartVersionAccess`. The access ends with a call to `MtCtxEndVersionAccess`.

Within the scope of these functions, any read operation is relative to the version of the objects corresponding to the time specified in `MtCtxStartVersionAccess`. Any modifications to the version is not allowed and not supported.

For any given database, you can determine all the versions that have been defined at different logical times. You can list these versions by using the functions `MtCtxOpenVersionStream`, `MtCtxNextVersion` and `MtCtxCloseStream`.

Summary

- Historical versions are stamped using a unique string for each version.
- Within the scope of an `MtCtxStartVersionAccess - MtCtxEndVersionAccess`, you have access to a state of the database that has been previously saved.
- Through a version access of the current state, you can read the latest version of objects outside a transaction context.
- Within the scope of `MtCtxStartVersionAccess - MtCtxEndVersionAccess`, you are not allowed to perform any modifications.
- Historical versions give a consistent view of the database at a specific time and are available outside of any transactional context.
- To access a specific version, specify the string returned at the moment of `MtCtxCommitTransaction`, as an argument of `MtCtxStartVersionAccess`.

List of Functions

```
MtSTS MtCtxAbortTransaction (void)
MtSTS MtCtxCommitTransaction(MtString prefix,
                             MtString* timeName)
MtSTS MtCtxEndVersionAccess (void)
MtSTS MtCtxGetWaitTime (MtLockWaitTime* waitTime)
MtSTS MtCtxStartVersionAccess (MtString timeName)
```

```
MtSTS MtCtxSetWaitTime (MtLockWaitTime waitTime)
MtSTS MtCtxStartTransaction (MtTranPriority priority)
```

1.3 Database Reading

Overview

Once an object is loaded or read, the value of the object is stored in the client cache. Using this value or extracting the values of the object properties does not generate server access. In other words, all operations are performed on the client cache and not on the database.

CAUTION: Loading an object does not imply loading the successors of the object.

There are several types of get functions which are further described in the following sections. A brief review of Matisse functions is presented below.

Each Matisse function taking a schema object as an argument has two slightly different forms depending on whether you specify the schema object by its identifier or using a character string. This is why many functions possess two slightly different variants. The `CreateObject` function, for example, has two variants - `MtCtxCreateObject` and `MtCtx_CreateObject`. `MtCtxCreateObject` take a character string to specify the class while `MtCtx_CreateObject` takes an identifier to specify the class.

For some `Get` functions, the number of possible variants increases. A `Get` function, as its name implies, gets a copy of a value from a database. The function may allocate a buffer to store a copy of this value, and then return a pointer to this buffer, or it may store a copy of this value in a buffer allocated by the calling program.

A `Get` function that allocates a buffer, begins with the letters `MtCtxMGet` or `MtCtx_MGet` (depending on whether you specify a character string or an object identifier.) A `Get` function that does not allocate a buffer, begins with the letters `MtCtxGet` or `MtCtx_Get`. Depending on the kind of `Get` function called, the following guidelines must be observed:

`MtCtxGet` or `MtCtx_Get`: When calling a `Get` function that does not allocate space for the value, the programmer must declare a buffer of the appropriate type and pass the address of this buffer as an argument. The `Get` function then copies the retrieved value into this buffer.

`MtCtxMGet` or `MtCtx_MGet`: When calling a `Get` function that allocates memory for a value, the user must declare a pointer to a variable of the appropriate type. The address of this pointer must be passed as an argument to the `Get` function that allocates memory. The address that the function stores in

this pointer can be used by the calling program to read the retrieved value. When manipulation of the data is no longer required, the program should deallocate the data with the `C free` function.

Schema Access

Any schema object can be accessed through entry-points, loops on the class instances, or through navigation among objects. However, Matisse offers quick functions that grant direct access to the objects.

Object Description

You can obtain explicit information on any Matisse object. Specifically, you can determine:

- The object's class
- If the object is part of the original meta-schema
- If the object is, or is not, an instance of a specific class

Value of a Property

The value of an object is made up of a set of associations of the type *`property/property_value`*.

The possible properties of an object are those defined by the object's class (and its superclasses), as well as the inverse properties of the relationships for which the class is a valid successor.

Attribute values are dynamically typed. The type of the attribute value is determined at run time.

Note, however, that in Matisse, when a property is unassigned, it has a default value. The default value for a relationship is an empty array of objects. The default value of an attribute is inherited from `MtDefaultValue`.

The value of the default value (when the `MtDefaultValue` of the attribute has not been specified) is of type `MT_NULL`.

Relations

Matisse manages inverse links.

When a link is added or deleted between two objects (via a relationship) the inverse link is automatically updated between both objects.

The successor, through the inverse relationship of an object, can then be considered as the predecessor of the object through the direct property. These concepts are symmetric.

You can search for predecessors using specific functions defined in Matisse. It may be easier, however, to use functions that search for successors, even when you are searching for predecessors. To do so, you must specify the inverse relationship.

The successors of a relationship can be ordered.

In addition to the functions that allow you to get all successors and predecessors of an object through a relationship, you can use a stream to enumerate all the successors or predecessors (see *section 1.5, Object Streaming*).

Multimedia Streaming

Large attributes of type list can be used to store multimedia data such as audio or video. For instance, if you access a video stored as a list of bytes, you will be able to read the video by blocks directly from the server and send it to a viewer without having the video copied to the client cache. The functions `MtCtxGetListElements` and `MtCtx_GetListElements` implement this interface.

Loading Objects

When accessing an attribute or a relationship in an object, that attribute or relationship is automatically loaded into the client cache. In addition, the functions `MtCtxLoadNumObjects` and `MtCtxLoadObjects` allow you to explicitly load objects into the client cache. Once the objects are loaded, information on these objects is retrieved from the cache rather than from the server.

Access Through Entry Points

An entry point enables you to access an instance using the value of one of its properties.

Attributes are characterized by a `MtMakeEntryFunction` function. For a specific attribute, when `MtMakeEntryFunction` is specified, any instance for which the attribute's value has been assigned can be accessed through one or more keywords (strings) computed from the value of the attribute. This feature lets you search quickly for instances based on specific indexing.

Note that a make entry function may produce empty strings. If this occurs, no keywords are indexed and the object will not be accessible through the entry point.

Note that entry points are not case sensitive.

In addition to the functions that allow you to retrieve objects from their entry points, you can use a stream to enumerate all these objects (see *section 1.5, Object Streaming*).

You can also retrieve schema objects by specifying only the value of their `MtName` attribute.

Access Through Indexes

Indexes allow access to an object stream. An index is defined by a set of one to four *criteria*, in other words, attributes that are attached to the same class. The values of the criteria may be ordered in ascending or in descending order. If an attribute is multiple valued (i.e., a list) for an object, there will be multiple entries for this object in the index.

To scan the index, you must specify a start value and an end value. Start and end values may be between zero and the maximum possible number of criteria for the index. The index may be scanned in direct (the direction in which the values were indexed) or in reverse order.

Information about Modified Successors

Matisse provides functions that return the list of all successors added to or removed from an object through a specific relationship, starting from the beginning of the transaction.

List of Functions

Schema Access

```
MtSTS MtCtxGetAttribute (MtContext ctx, MtOid* attribute,
                         MtString attributeName)
MtSTS MtCtxGetClass (MtContext ctx, MtOid*class,
                     MtString className)
MtSTS MtCtxGetIndex (MtContext ctx, MtOid* index,
                     MtString indexName)
MtSTS MtCtxGetRelationship (MtContext ctx, MtOid*
relationship,
                            MtString relationshipName)
```

Object Description

```
MtSTS MtCtxGetObjectClass (MtContext ctx, MtOid* class,
                           MtOid object)
MtSTS MtCtxOpenAttributeStream (MtContext ctx,
                            MtStream* objectAttStream,
                            MtOid object)
MtSTS MCtxtOpenInverseRelationshipsStream (MtContext ctx,
                           MtStream* objectIRelStream,
                           MtOid object)
MtSTS MtCtxOpenRelationshipsStream (MtContext ctx,
                            MtStream* objectRelStream,
                            MtOid object)
MtSTS MtCtxIsPredefinedObjet (MtContext ctx,
                           MtBoolean* predefinedMSP,
                           MtOid object)
MtSTS MtCtxIsInstanceOf (MtContext ctx, MtBoolean* result,
                         MtOid object,
                          MtString className)
MtSTS MtCtx_IsInstanceOf (MtContext ctx, MtBoolean* result,
                           MtOid object,
                           MtOid class)
```

Attribute Value in an Object

```
MtSTS MtCtxGetDimension (MtContext ctx, MtOid object,
                         MtString attributeName,
                         MtSize rankIndex,
                         MtSize* dimension)
MtSTS MCtxt_GetDimension (MtContext ctx, MtOid object,
                          MtOid attribute,
```

```
                        MtSize rankIndex,
                        MtSize* dimension)
MtSTS MtCtxGetListElements (MtContext ctx, MtOid object,
                        MtString attributeName,
                        MtType type,
                        void* bufList,
                        MtSize* numElts,
                        MtSize firstEltOffset)
MtSTS MtCtx_GetListElements (MtContext ctx, MtOid object,
                        MtOid attribute,
                        MtType type,
                        void* bufList,
                        MtSize* numElts,
                        MtSize firstEltOffset)
MtSTS MtCtxGetValue (MtContext ctx, MtOid object,
                    MtString attributeName,
                    MtType* type,
                    void* value, MtSize* rank,
                    MtSize* size,
                    MtBoolean* defaultValueP)
MtSTS MtCtx_GetValue (MtContext ctx, MtOid object, MtOid
attribute,
                    MtType* type, void* value,
                    MtSize* rank, MtSize* size,
                    MtBoolean* defaultValueP)
MtSTS MtCtxMGetValue (MtContext ctx, MtOid object,
                    MtString attributeName,
                    MtType* type, void** value,
                    MtSize* rank,
                    MtBoolean* defaultValueP)
MtSTS MtCtx_MGetValue (MtContext ctx, MtOid object,
                    MtOid attribute, MtType* type,
                    void** value, MtSize* rank,
                    MtBoolean* defaultValueP)
```

Relationship Value in an Object

```
MtSTS MtCtxGetSuccessors (MtContext ctx, MtSize*
numObjects,
                        MtOid* successors,
                        MtOid object,
                        MtString relationshipName)
MtSTS MtCtx_GetSuccessors (MtContext ctx, MtSize*
numObjects,
                        MtOid* successors,
                        MtOid object,
                        MtOid relationship)
MtSTS MtCtxMGetSuccessors (MtContext ctx, MtSize*
numObjects,
                        MtOid** successors,
                        MtOid object,
                        MtString relationshipName)
```

```
MtSTS MtCtx_MGetSuccessors (MtContext ctx, MtSize*
numObjects,
                           MtOid** successors,
                           MtOid object,
                           MtOid relationship)
MtSTS MtCtxOpenSuccessorsStream (MtContext ctx, MtStream*
relStream,
                         MtOid object,
                         MtString relationshipName)
MtSTS MtCtx_OpenSuccessorsStream
                         (MtContext ctx, MtStream*
relStream,
                          MtOid object,
                          MtOid relationship)
```

Inverse Links in an Object

```
MtSTS MtCtxGetPredecessors (MtContext ctx, MtSize*
numObjects,
                           MtOid* predecessors,
                           MtOid object,
                           MtString relationshipName)
MtSTS MtCtx_GetPredecessors (MtContext ctx, MtSize*
numObjects,
                             MtOid* predecessors,
                             MtOid relationship)
MtSTS MtCtxMGetPredecessors (MtContext ctx, MtSize*
numObjects,
                           MtOid** predecessors,
                           MtOid object,
                           MtString relationshipName)
MtSTS MtCtx_MGetPredecessors (MtContext ctx, MtSize*
numObjects,
                             MtOid** predecessors,
                             MtOid object,
                             MtOid relationship)
MtSTS MtCtxOpenPredecessorsStream (MtContext ctx,
                           MtStream* IRelStream,
                           MtOid object,
                           MtString relationshipName)
MtSTS MCtxt_OpenPredecessorsStream (MtContext ctx,
                            MtStream* IRelStream,
                            MtOid object,
                            MtOid relationship)
```

Loading Object

```
MtSTS MtCtxLoadNumObjects MtContext ctx, (MtSize
numObjects,
                           MtOid* objects)
MtSTS MtCtxLoadObjects (MtContext ctx, MtSize numObjects,
...)
```

Entry Points Access

```
MtSTS MtCtxGetObjectsFromEntryPoint(MtContext ctx,
                             MtSize* numObjects,
```

MtOid* *objects*,
MtString *entryPoint*,
MtString *dictName*,
MtString *className*)

MtSTS **MtCtx_GetObjectsFromEntryPoint** (MtContext *ctx*,
MtSize* *numObjects*,
MtOid* *objects*,
MtString *entryPoint*,
MtOid *dictionary*,
MtOid *class*)

MtSTS **MtCtxMGetObjectsFromEntryPoint** (MtContext *ctx*,
MtSize* *numObjects*,
MtOid** *objects*,
MtString *entryPoint*,
MtChar* *dictName*,
MtChar* *className*)

MtSTS **MtCtx_MGetObjectsFromEntryPoint** (MtContext *ctx*,
MtSize* *numObjects*,
MtOid** *objects*,
MtChar* *entryPoint*,
MtOid *dictionary*,
MtOid *class*)

MtSTS **MtCtxOpenEntryPointStream** (MtContext *ctx*,
MtStream* *entryPointStream*,
MtChar* *entryPoint*,
MtChar* *dictName*,
MtChar* *className*,
MtSize *numObjectsPerBuffer*)

MtSTS **MtCtx_OpenEntryPointStream** (MtContext *ctx*,
MtStream* *entryPointStream*,
MtChar* *entryPoint*,
MtOid *dictionary*,
MtOid *class*,
MtSize *numObjectsPerBuffer*)

Index Access

MtSTS **MtCtxGetObjectsFromIndex** (MtContext *ctx*, MtSize *numObjects*,
MtOid **objects;*
void **indexEntry[]*,
MtSize *nbOfCriteria*,
MtString *indexName*,
MtString *className)*

MtSTS **MtCtx_GetObjectsFromIndex** (MtContext *ctx*, MtSize *numObjects*,
MtOid **objects;*
void **indexEntry[]*,
MtSize *nbOfCriteria*,
MtOid *index*,
MtOid *aClass)*

MtSTS **MtCtxMGetObjectsFromIndex** (MtContext *ctx*, MtSize *numObjects*,

```
                        MtOid **objects;
                        void *indexEntry[],
                        MtSize nbOfCriteria,
                        MtString indexName,
                        MtString className)
MtSTS MtCtx_MGetObjectsFromIndex (MtContext ctx, MtSize
numObjects,
                        MtOid **objects;
                        void *indexEntry[],
                        MtSize nbOfCriteria,
                        MtOid index,
                        MtOid aClass)
MtSTS MtCtxOpenIndexEntriesStream (MtContext ctx, MtStream
*stream,
                        MtString indexName,
                        MtString className,
                        MtDirection direction,
                        MtSize nbOfCriteria,
                        void *startValues [],
                        void *endValues,
                        MtSize nbEntriesPerCall)
MtSTS MtCtx_OpenIndexEntriesStream (MtContext ctx, MtStream
*stream,
                         MtOid index,
                         MtOid class,
                         MtDirection direction,
                         MtSize nbOfCriteria,
                         void *startValues [],
                         void *endValues,
                         MtSize nbEntriesPerCall)
MtSTS MtCtxOpenIndexObjectsStream (MtContext ctx, MtStream
*stream,
                        MtString indexName,
                        MtString className,
                        MtDirection direction,
                        MtSize nbOfCriteria,
                        void *startValues [],
                        void *endValues,
                        MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenIndexObjectsStream (MtContext ctx, MtStream
*stream,
                         MtOid index,
                         MtOid class,
                         MtDirection direction,
                         MtSize nbOfCriteria,
                         void *startValues [],
                         void *endValues,
                         MtSize nbObjectsPerCall)
MtSTS MtCtxNextIndexEntry (MtContext ctx, MtStream stream,
                       void *values [],
                       MtOid *object)
```

```
MtSTS MtCtxNextObject (MtContext ctx, MtStream, MtOid
*object)
MtSTS MtCtxNextObjects (MtContext ctx, MtStream stream,
                        MtOid* objects,
                        MtSize* numObjects)
```

Modified Successors

```
MtSTS MtCtxGetAddedSuccessors (MtContext ctx,
                               MtSize* numAddedSuccessors,
                               MtOid* allAddedSuccessors,
                               MtOid object,
                               MtString relationshipName)
MtSTS MtCtx_GetAddedSuccessors (MtContext ctx,
                               MtSize* numAddedSuccessors,
                               MtOid* allAddedSuccessors,
                               MtOid object,
                               MtOid relationship)
MtSTS MtCtxGetRemovedSuccessors (MtContext ctx,
                                 MtSize* numRemSuccessors,
                                 MtOid* allRemSuccessors,
                                 MtOid object,
                                 MtString relationshipName)
MtSTS MtCtx_GetRemovedSuccessors (MtContext ctx,
                                  MtSize* numRemSuccessors,
                                  MtOid* allRemSuccessors,
                                  MtOid object,
                                  MtOid relationship)
MtSTS MtCtxMGetAddedSuccessors (MtContext ctx,
                               MtSize* numAddedSuccessors,
                               MtOid** allAddedSuccessors,
                               MtOid object,
                               MtString relationshipName)
MtSTS MtCtx_MGetAddedSuccessors (MtContext ctx,
                               MtSize* numAddedSuccessors,
                               MtOid** allAddedSuccessors,
                               MtOid object,
                               MtOid relationship)
MtSTS MtCtxMGetRemovedSuccessors (MtContext ctx,
                                 MtSize* numRemSuccessors,
                                 MtOid** allRemSuccessors,
                                 MtOid object,
                                 MtString relationshipName)
MtSTS MtCtx_MGetRemovedSuccessors (MtContext ctx,
                                  MtSize* numRemSuccessors,
                                  MtOid** allRemSuccessors,
                                  MtOid object,
                                  MtOid relationship)
```

1.4 Database Modification

Overview

The only objects that can be modified in the `MT_DATA_MODIFICATION` mode are the database terminal instances. The schema (and therefore the meta-schema) cannot be modified.

In `MT_DATA_DEFINITION` mode, the terminal instances, schema and the meta-schema can be modified.

Modifications can be performed during a transaction only. When a stream is opened, only the modifications that do not corrupt the stream are authorized. A transaction ends with a commit or an abort. An abort may be implemented by Matisse when a deadlock occurs.

Any modification can be divided into two steps:

1. Calling the modification function:

   When object modification is attempted, a check occurs to determine if the object is modifiable. Checks are made to determine if the property to be modified is allowed for the object, if the object is updatable (schema objects are not updatable in `MT_DATA_MODIFICATION` mode), if the object has already been modified, or if the modifications will make it impossible to carry out future checks.

   A check is performed on the values that are stored. The values must conform to the constraints of the database: Storing a number higher than that specified, a type other than that specified, etc. is not permitted.

2. Committing the transaction:

   Object consistency is checked only when the transaction is to be committed. All modified objects and entry-points are then validated and written. The transaction is then definitively committed, and the client cache is flushed.

When an object is validated, for each object property that has been modified, Matisse checks the structural constraints (the value of the attribute `MtType` for an attribute, the values of the attribute `MtCardinality`, and the relationship `MtSuccessors` for a relationship, etc.).

If an error occurs while the transaction is being committed, the transaction is not aborted. The user must either handle the error, abort it, or correct the data. Matisse presents various functions to validate objects individually, before the overall transaction commit.

There are three categories of modification:

- Object creation
- Object deletion
- Object modification

Object Validation

Objects that are modified during a transaction are checked at the end of the transaction only (i.e., when `MtCommitTransaction` is called).

When an object is validated, for each object property that has been modified, Matisse checks the structural constraints (the attribute `MtType` for an attribute, the attribute `MtCardinality`, and the relationship `MtSuccessors` for a relationship).

Multimedia Streaming

Large attributes of type list can be used to store multimedia data such as audio or video. If you use the list interface to store a large attribute, the attribute will be stored directly on the server without caching in the client cache and the attribute will be stored outside of the embedding object. By default the large attribute will not be fetched when you fetch the object. Instead, you can fetch it through the streaming API. The functions `MtSetListElements` and `Mt_SetListElements` implement this interface.

Entry Points

When you modify the value of an attribute that has an entry-point function, the make entry function is called twice. The function is first called with the previous value to delete the previous entry point. The function is then called with the new value to generate a new entry point for the attribute.

Indexes

When you modify an object by changing the value of an attribute that represents an index criterion, the index is updated.

In `MT_DATA_DEFINITION` mode, you may also want to create a new index for a class which already has instances. The entries in the index for each instance of the class are created at commit time. During the transaction, the index is not scannable.

The index may be deleted in `MT_DATA_DEFINITION` mode. There is no other restriction for deleting an index.

Within the same transaction, you may create several indexes. You may also delete an index.

List of Functions

Object Creation

```
MtSTS MtCtxCreateObject (MtContext ctx, MtOid* object,
                         MtString className)
MtSTS MtCtx_CreateObject (MtContext ctx, MtOid* object,
MtOid class)
```

Object Deletion

```
MtSTS MtCtxRemoveObject (MtContext ctx, MtOid object)
```

Object Modification

```
MtSTS MtCtxAddNumSuccessors (MtContext ctx, MtOid object,
                             MtString relationshipName,
                             MtSize numSuccessors,
                             MtOid* successors)
```

```
MtSTS MtCtx_AddNumSuccessors (MtContext ctx, MtOid object,
                              MtOid relationship,
                              MtSize numSuccessors,
                              MtOid* successors)
MtSTS MtCtxAddSuccessor (MtContext ctx, MtOid object,
                         MtString relationshipName,
                         MtOid successor,
                         MtWhere where, ...)
MtSTS MtCtx_AddSuccessor (MtContext ctx, MtOid object,
                          MtOid relationship,
                         MtOid successor,
                         MtWhere where, ...)
MtSTS MtCtxAddSuccessors (MtContext ctx, MtOid object,
                          MtString relationshipName,
                          MtSize numSuccessors, ...)
MtSTS MtCtx_AddSuccessors (MtContext ctx, MtOid object,
                           MtOid relationship,
                           MtSize numSuccessors, ...)
MtSTS MtCtxRemoveAllSuccessors (MtContext ctx, MtOid
object,
                           MtString relationshipName)
MtSTS MtCtx_RemoveAllSuccessors (MtContext ctx, MtOid
object,
                                 MtOid relationship)
MtSTS MtCtxRemoveNumSuccessors (MtContext ctx, MtOid
object,
                           MtString relationshipName,
                           MtSize numSuccessors,
                           MtOid* successors)
MtSTS MtCtx_RemoveNumSuccessors (MtContext ctx, MtOid
object,
                              MtOid relationship,
                              MtSize numSuccessors,
                              MtOid* successors)
MtSTS MtCtxRemoveSuccessors (MtContext ctx, MtOid object,
                           MtString relationshipName,
                           MtSize MtContext ctx,
numSuccessors, ...)
MtSTS MtCtx_RemoveSuccessors (MtOid object,
                              MtOid relationship,
                              MtSize numSuccessors,
                              ...)
MtSTS MtCtxRemoveValue (MtContext ctx, MtOid object,
                        MtString attributeName)
MtSTS MtCtx_RemoveValue (MtContext ctx, MtOid object,
                         MtOid attribute)
MtSTS MtCtxSetListElements (MtContext ctx, MtOid object,
                        MtString attributeName,
                        MtType type,
                        void* buflist,
```

```
                        MtSize* numElts,
                        MtSize firstEltOffset,
                        MtBoolean discardAfter)
MtSTS MtCtx_SetListElements (MtContext ctx, MtOid object,
                         MtOid attribute)
                        MtType type,
                        void* buflist,
                        MtSize* numElts,
                        MtSize firstEltOffset,
                        MtBoolean discardAfter)
MtSTS MtCtxSetValue (MtContext ctx, MtOid object,
                    MtString attributeName,
                    MtType type, void* value,
                    MtSize rank,
                    ...)
MtSTS MtCtx_SetValue (MtContext ctx, MtOid object,
                     MtOid attribute,
                     MtType type,
                     void* value, MtSize rank, ...)
```

Entry Points

```
MtSTS <make entry function> (MtSize numArgs,
                             MtSize* oneMore,
                             MtString buf,
                             void** context,
                             MtType type,
                             void* value,
                             MtSize rank,
                             MtSize* dims)
```

1.5 Object Streaming

Overview

The stream mechanism offers the capability of successively retrieving a set of objects that share a common feature (e.g. they all point to the same entry-point or they all belong to the same class). Object identifiers are transferred to the server when an object is accessed through the stream while the objects themselves are not transferred to the server unless a read operation is applied.

Using streams, you can scan:

- all instances of a class and its subclasses by opening a `classStream` (using a function such as `,MtCtxOpenIntancesStream`, `MtCtxOpenOwnIntancesStream`), and mapping the instances with the `MtCtxNextObject(s)` function.
- all objects pointed to by the same entry-point and depending on a specific class and relationship, This is done by opening an `EPStream` (using a function such as `MtCtxOpenEntryPointStream`) and mapping the objects with the `MtCtxNextObject(s)` function.

- an index from one entry to another. This is done by opening an `IndexStream` (using a function such as `MtCtxOpenIndexStream`) and mapping the objects with the `MtCtxNextObject(s)` function or mapping both the entries and the objects with the `MtCtxNextIndexEntry` function.
- all successors of an object specified by a relationship. This is done by opening a `RelStream` (using a function such as `MtCtxOpenRelationsipsStream`), and mapping the successors with the `MtCtxNextObject(s)` functions.
- all the predecessors of an object specified by a relationship. This is done by opening an `IRelStream` (using a function such as `MtCtxOpenInverseRelationshipsStream`) and mapping the successors with the `MtCtxNextObject(s)` function.
- all the attributes of an object. This is done by opening an `ObjectAttStream` (using a function such as `MtCtxOpenAttributesStream`) and mapping the attributes with the `MtCtxNextProperty` function. This function indicates whether or not a value has been assigned for each object attribute.
- all the relationships of an object. This is done by opening an `ObjectRelStream` (using a function such as `MtCtxOpenReleationshipsStream`), and mapping the relationships with the `MtCtxNextProperty` function. This function specifies whether the relationship has a value in the object.
- all the inverse relationships present in an object. This is done by opening an `ObjIRelStream` (using a function such as `MtCtxOpenInverseRelationshipsStream`) and mapping the inverse relationships with the `MtCtxNextProperty` function.

When a stream is opened, you can modify, create or delete the objects of the stream. The stream will not immediately reflect these changes. Once all the elements of the stream have been retrieved, the function will return the `MATISSE_ENDOFSTREAM` code. Use `MtCtxCloseStream` to close the stream.

List of Functions

```
MtSTS MtCtxCloseStream (MtContext ctx, MtStream stream)
MtSTS MtCtxNextObject (MtContext ctx, MtStream stream,
                       MtOid* object)
MtSTS MtCtxNextObjects (MtContext ctx, MtStream stream,
                        MtOid* objects,
                        MtSize* numObjects)
MtSTS MtCtxNextProperty (MtContext ctx, MtStream stream,
                         MtOid* property,
                         MtBoolean* specifiedP)
MtSTS MtCtxNextIndexEntry (MtContext ctx, MtStream stream,
                           void *values [],
                           MtOid *object)
MtSTS MtCtxOpenInstancesStream (MtContext ctx,
                           MtStream* classStream,
```

```
                        MtString className,
                         MtSize numInstPerBuffer)
MtSTS MtCtx_OpenInstancesStream (MtContext ctx,
                         MtStream* classStream,
                         MtOid class,
                          MtSize numInstPerBuffer)
MtSTS MtCtxOpenEntryPointStream (MtContext ctx,
                       MtStream* entryPointStream,
                       MtString entryPoint,
                       MtString dictName,
                       MtString className,
                       MtSize numInstPerBuffer)
MtSTS MtCtx_OpenEntryPointStream (MtContext ctx,
                        MtStream* entryPointStream,
                        MtString entryPoint,
                        MtOid dictionary,
                        MtOid class,
                        MtSize numInstPerBuffer)
MtSTS MtCtxOpenIndexEntriesStream (MtContext ctx, MtStream
*stream,
                         MtString indexName,
                         MtString className,
                         MtDirection direction,
                         MtSize nbOfCriteria,
                         void *startValues [],
                         void *endValues,
                         MtSize nbEntriesPerCall)
MtSTS MtCtx_OpenIndexEntriesStream (MtContext ctx, MtStream
*stream,
                          MtOid index,
                          MtOid class,
                          MtDirection direction,
                          MtSize nbOfCriteria,
                          void *startValues [],
                          void *endValues,
                          MtSize nbEntriesPerCall)
MtSTS MtCtxOpenIndexObjectsStream (MtContext ctx, MtStream
*stream,
                         MtString indexName,
                         MtString className,
                         MtDirection direction,
                         MtSize nbOfCriteria,
                         void *startValues [],
                         void *endValues,
                         MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenIndexObjectsStream (MtContext ctx, MtStream
*stream,
                          MtOid index,
                          MtOid class,
                          MtDirection direction,
                          MtSize nbOfCriteria,
```

```
                                void *startValues [],
                                void *endValues,
                                MtSize nbObjectsPerCall)
MtSTS MtCtxOpenPredecessorsStream (MtContext ctx,
                           MtStream* iRelStream,
                           MtOid object,
                           MtString relationshipName)
MtSTS MtCtx_OpenPredecessorsStream (MtContext ctx,
                            MtStream* iRelStream,
                            MtOid object,
                            MtOid relationship)
MtSTS MtCtxOpenAttributesStream (MtContext ctx,
                           MtStream* objectAttStream,
                           MtOid object)
MtSTS MtCtxOpenInverseRelationshipsStream MtContext ctx, (
                          MtStream* objectIRelStream,
                          MtOid object)
MtSTS MtCtxOpenRelationshipsStream (MtContext ctx,
                           MtStream* objectRelStream,
                           MtOid object)
MtSTS MtCtxOpenSuccessorsStream (MtContext ctx, MtStream*
relStream,
                          MtOid object,
                          MtString relationshipName)
MtSTS MtCtx_OpenSucessorsStream (MtContext ctx, MtStream*
relStream,
                           MtOid object,
                           MtOid relationship)
```

1.6 Class Description Access

You can obtain all the properties (including both attribute and relationship descriptors) or all the superclasses defined for a specific class, using a function such as `MtGetSuccessors`. If this function is used, however, the properties or the superclasses that are obtained are those defined in the class without taking the inheritance mechanism into account. If you want to obtain all the properties or all the superclasses of the class (defined directly or inherited), you must use one of the functions listed below.

Additionally, Matisse presents functions that provide information such as the number of instances and the set of subclasses of a class.

List of Functions

```
MtSTS MtCtxGetAllAttributes (MtContext ctx, MtSize*
numAttributes,
                                MtOid* attributes,
                                MtString className)
MtSTS MtCtx_GetAllAttributes (MtContext ctx, MtSize*
numAttributes,
```

MtOid* *attributes*,
MtOid *class*)

MtSTS **MtCtxGetAllInverseRelationships** (MtContext *ctx*,
MtSize* *numIRelationships*,
MtOid* *iRelationships*,
MtString *className*)

MtSTS **MtCtx_GetAllInverseRelationships** (MtContext *ctx*,
MtSize* *numIRelationships*,
MtOid* *iRelationships*,
MtOid *class*)

MtSTS **MtCtxGetAllRelationships** (MtContext *ctx*,
MtSize* *numRelationships*,
MtOid* *relationships*,
MtString *className*)

MtSTS **MtCtx_GetAllRelationships** (MtContext *ctx*,
MtSize* numRelationships,
MtOid* *relationships*,
MtOid *class*)

MtSTS **MtCtxGetAllSubclasses** (MtContext *ctx*, MtSize*
numSubclasses,
MtOid* *subClasses*,
MtString *className*)

MtSTS **MtCtx_GetAllSubclasses** (MtSize* *numSubclasses*,
MtOid* *subClasses*,
MtOid *class*)

MtSTS **MtCtxGetAllSuperclasses** (MtContext *ctx*,
MtSize* *numSuperclasses*,
MtOid* *superClasses*,
MtString *className*)

MtSTS **MtCtx_GetAllSuperclasses** (MtContext *ctx*,
MtSize* *numSuperclasses*,
MtOid* *superClasses*,
MtOid *class*)

MtSTS **MtCtxGetInstancesNumber** (MtContext *ctx*,
MtSize* *instancesNumber*,
MtString *className*)

MtSTS **MtCtx_GetInstancesNumber** (MtContext *ctx*,
MtSize* *instancesNumber*,
MtOid *class*)

MtSTS **MtCtxMGetAllAttributes** (MtContext *ctx*, MtSize*
numAttributes,
MtOid** *attributes*,
MtString *className*)

MtSTS **MtCtx_MGetAllAttributes** (MtContext *ctx*, MtSize*
numAttributes,
MtOid** *attributes*,
MtOid *class*)

MtSTS **MtCtxMGetAllInverseRelationships** (MtContext *ctx*,
MtSize* *numIRelationships*,

```
                        MtOid** iRelationships,
                        MtString className)
MtSTS MtCtx_MGetAllInverseRelationships (MtContext ctx,
                        MtSize* numIRelationships,
                        MtOid** iRelationships,
                        MtOid class)
MtSTS MtCtxMGetAllRelationships (MtContext ctx,
                         MtSize* numRelationships,
                         MtOid** relationships,
                         MtString className)
MtSTS MtCtx_MGetAllRelationships (MtContext ctx,
                         MtSize* numRelationships,
                         MtOid** relationships,
                         MtOid class)
MtSTS MtCtxMGetAllSubclasses (MtContext ctx, MtSize*
numSubclasses,
                           MtOid** subClasses,
                           MtString className)
MtSTS MtCtx_MGetAllSubclasses(MtContext ctx, MtSize*
numSubclasses,
                           MtOid** subClasses,
                           MtOid class)
MtSTS MtCtxMGetAllSuperclasses (MtContext ctx,
                          MtSize* numSuperclasses,
                          MtOid** superClasses,
                          MtString className)
MtSTS MtCtx_MGetAllSuperclasses (MtContext ctx,
                          MtSize* numSuperclasses,
                          MtOid** superClasses,
                          MtOid class)
MtSTS MtCtxOpenInstancesStream (MtContext ctx,
                         MtStream* classStream,
                         MtString className,
                         MtSize numInstPerBuffer)
MtSTS MtCtx_OpenInstancesStream (MtContext ctx,
                         MtStream* classStream,
                         MtOid class,
                          MtSize numInstPerBuffer)
MtSTS MtCtxOpenOwnInstancesStream (MtContext ctx,
                         MtStream* classStream,
                         MtString className,
                         MtSize numInstPerBuffer)
MtSTS MtCtx_OpenOwnInstancesStream (MtContext ctx,
                         MtStream* classStream,
                         MtOid class,
                          MtSize numInstPerBuffer)
```

1.7 Embedded SQL

The functions discussed in this section execute SQL queries and retrieve results. Please refer to the *MATISSE SQL Programmer's Guide* for a description of the MATISSE SQL syntax.

List of Functions

```
MtSTS MtCtxSQLAllocStmt (MtContext ctx, MtSQLStmt* stmt)
MtSTS MtCtxSQLExecDirect
   (MtContext ctx, MtSQLStmt stmt,
    MtString stmtStr)
MtSTS MtCtxSQLFreeStmt (MtContext ctx, MtSQLStmt stmt)
MtSTS MtCtxSQLGetColumnInfo
   (MtContext ctx, MtSQLStmt stmt,
    MtSize colNum,
    MtType* coltype,
    MtString colname,
    MtSize* sz)
MtSTS MtSCtxQLGetParamDimensions
   (MtContext ctx, MtSQLStmt stmt,
    MtSize  paramNumber,
    MtSize* rank,
    MtSize  dimensions)
MtSTS MtCtxSQLGetParamListElements
   (MtContext ctx, MtSQLStmt stmt,
    MtSize  paramNumber,
    MtType  type,
    void*   buf,
    MtSize* buf_size,
    MtSize  firstEltOffset)
MtSTS MtCtxSQLGetParamValue
   (MtContext ctx, MtSQLStmt stmt,
    MtSize paramNumber,
    MtType* type,
    void* value,
    MtSize* size)

MtSTS MtCtxSQLMGetParamValue
   (MtContext ctx, MtSQLStmt stmt,
    MtSize paramNumber,
    MtType* type,
    void** value,
    MtSize* size)
MtSTS MtCtxSQLGetRowListElements
   (MtContext ctx, MtStream stream,
    MtSize colNum,
    MtType colType,
```

```
    void* bufList,
    MtSize* numElts,
    MtSize firstEltOffset)
MtSTS MtCtxSQLGetRowValue
   (MtContext ctx, MtStream stream,
    MtSize colNum,
    MtType* colType,
    void* value,
    MtSize* size)

MtSTS MtCtxSQLMGetRowValue
   (MtContext ctx, MtStream stream,
    MtSize colNum,
    MtType* colType,
    void** value,
    MtSize* size)
MtSTS MtCtxSQLGetStmtInfo
   (MtContext ctx, MtSQLStmt stmt,
    MtSQLStmtAttr stmtAttr,
    void* value,
    MtSize* size)
MtSTS MtCtxSQLGetStmtType
   (MtContext ctx, MtSQLStmt stmt,
    MtSQLStmtType* stmtType)
MtSTS MtCtxSQLNext
   (MtContext ctx, MtStream stream)
MtSTS MtCtxSQLNumResultCols
   (MtContext ctx, MtSQLStmt stmt,
    MtSize* numcols)
MtSTS MtCtxSQLOpenStream
   (MtContext ctx, MtStream* stream.
    MtSQLStmt stmt)
```

1.8 Error Handling

Status Management

Each Matisse function returns a status (type `MtSTS`). The status MUST be tested whenever a Matisse function is called. The functions `MtFailure` and `MtSuccess` test respectively, the failure or the success of the operation. The functions `MtCtxError` and `MtCtxPError` provide additional information on the error.

Programmers can generate their own errors using the function `MtCtxMakeUserError`. They can, therefore, associate a specific value and a string with the error code which is always `MATISSE_USERERROR`.

Summary

- Each Matisse function returns a status.
- The programmer can generate custom errors.

List of Functions

```
int MtCtxCheckErrorP (MtContext ctx, MtSTS status)
char* MtCtxError (MtContext ctx)
int MtFailure (MtSTS status)
void* MtCtxGetUserError (MtContext ctx)
MtSTS MtCtxMakeUserError (MtContext ctx, void* error,
                          MtString errorString)
void MtCtxPError (MtContext ctx, MtString comment)
int MtSuccess (MtSTS status)
```

Variable

```
MtErrorStr
```

1.9 Miscellaneous

Dates and Times

The C API contains several functions to handle date/time values.

Numeric Types

The C API contains functions to handle fixed precision types with maximum precision 19 and maximum scale 19. Default precision and scale is 19, 2.

Print Function

The function `MtCtxPrint` prints an object, independent of its type.

Locks

Explicit locks let you have a more accurate control over the objects that are manipulated. You can, for example, set a lock on several objects simultaneously.

In addition, you can select a pessimistic strategy explicitly in some situations.

For pessimistic locking, write locks must be requested for any to-be-modified object. When used, there is less risk that the transaction will aborted by deadlocks. If a deadlock is detected during an explicit lock request, the request fails but the transaction is not aborted.

If, however, the modified objects are related to other database objects, the operations on inverse links may generate deadlocks.

Using explicit locks may handicap other users.

The Matisse programmer interface provides the option of setting explicit locks on any object, except on the schema.

Save Time Enumeration

The function `MtCtxCommitTransaction` lets you associate a string with the logical time that results. You can use this string to identify the logical time that you want to consult in a version access. A state that can be consulted is indicated by a string. You can get the list of all the states that can be consulted using the stream on these states (`MtCtxOpenVersionStream`, `MtCtxNextVersion` and `MtCtxCloseStream`.

Memory Management

Within a transaction, the client cache can grow disproportionately if the user is handling numerous objects.

Matisse offers the option of freeing objects from the client cache..

List of Functions

Dates and Times

```
MtSTS MtTimestampAdd (
                MtTimestamp *result,
                MtTimestamp *time,
                MtInterval *interval)
MtSTS MtTimestampBuild (
                MtTimestamp *result,
                MtString buffer,
                MtTimeZone timezone)
MtSTS MtTimestampCompare (
                MtInteger *result,
                MtTimestamp *time1,
                MtTimestamp *time2)
MtSTS MtTimestampDiff (
                MtInteval *result,
                MtTimestamp *time1,
                MtTimestamp *time2)
MtSTS MtTimestampGetCurrent (
                MtTimestamp *currentTime)
MtSTS MtTimestampPrint (
                MtString buffer,
                MtSize bufferSize,
                MtString format,
                MtTimestamp *time,
                MtTimeZone timezone)
MtSTS MtTimestampSubtract (
                MtTimestamp *result,
                MtTimestamp *time,
                MtInterval *interval)
MtSTS MtIntervalAdd (
                MtInterval *result,
                MtInterval *interval1,
                MtInterval *interval2)
```

```
MtSTS MtIntervalBuild (
                MtInterval *result,
                MtString buffer)
MtSTS MtIntervalCompare (
                MtInteger *result,
                MtInterval *interval1,
                MtInterval *interval2)
MtSTS MtIntervalDivide (
                MtInterval *result,
                MtInterval *interval,
                MtInteger n)
MtSTS MtIntervalMultiply (
                MtInterval *result,
                MtInterval *interval,
                MtInteger n)
MtSTS MtIntervalPrint (
                MtString buffer,
                MtSize bufferSize,
                MtString format,
                MtInterval *interval)
MtSTS MtIntervalSubtract (
                MtInterval *result,
                MtInterval *interval1,
                MtInterval *interval2)
```

Numeric Types

```
MtSTS MtNumericAdd (
                MtNumeric *result,
                MtNumeric *value1,
                MtNumeric *value2)
MtSTS MtNumericBuild (
                MtNumeric *result,
                MtString string,
                MtInteger precision,
                MtInteger scale)
MtSTS MtNumericCompare (
                MtInteger *result,
                MtNumeric *value1,
                MtNumeric *value2)
MtSTS MtNumericDivide (
                MtNumeric *result,
                MtNumeric *value1,
                MtNumeric *value2)
MtSTS MtNumericFromDouble (
                MtNumeric *result,
                MtDouble value)
MtSTS MtNumericFromLong (
                MtNumeric *result,
                MtLong value)
```

```
MtSTS MtNumericGetPrecision (
                MtSize *precision,
                MtString value)
MtSTS MtNumericGetScale (
                MtSize *scale,
                MtString value)
MtSTS MtNumericMultiply (
                MtNumeric *result,
                MtNumeric *value1,
                MtNumeric *value2)
MtSTS MtNumericPrint (
                MtString buffer,
                MtSize buffsz,
                MtNumeric *value)
MtSTS MtNumericRound (
                MtNumeric *result,
                MtNumeric *value,
                MtSize roundScale,
                MtRounding roundingMethod)
MtSTS MtNumericSubtract (
                MtNumeric *result,
                MtNumeric *value1,
                MtNumeric *value2)
MtSTS MtNumericToDouble (
                MtDouble *result,
                MtNumeric *value)
MtSTS MtNumericToLong (
                   MtLong  *result,
                   MtNumeric *value)
```

Print Function

```
MtSTS MtCtxPrint (MtContext ctx, MtOid object, FILE*
stream)
```

Locks

```
MtSTS MtCtxLockNumObjects (MtContext ctx, MtSize
numObjects,
                        MtOid* objects,
                        MtLock* locks)
MtSTS MtCtxLockObjects (MtContext ctx, MtSize numObjects,
                    MtOid firstObject,
                    MtLock firstLock, ...)
MtSTS MtCtxLockObjectsFromEntryPoint (MtContext ctx, MtLock
lock,
                            MtString entryPoint,
                            MtString dictName,
                            MtString className)
MtSTS MtCtx_LockObjectsFromEntryPoint (MtContext ctx,
MtLock lock,
```

MtString *entryPoint*,
MtOid *dictionary*,
MtOid *class*)

Save Times

MtSTS **MtCtxNextVersion** (MtContext *ctx*, MtStream *versionStream*,
MtString *buf*,
MtSize *bufSize*)

MtSTS **MCtxtOpenVersionStream** (MtContext *ctx*,
MtStream* *versionStream*)

Memory Management

MtSTS **MtCtxFreeNumObjects** (MtContext *ctx*, MtSize *numObjects*,
MtOid* *Objects*)

MtSTS **MtCtxFreeObjects** (MtContext *ctx*, MtSize *numObjects*, ...)

2 Type Reference

2.1 Matisse Programming Types

When you create a Matisse schema or write a database application, you should only use recommended programming types to manage attribute values. Most programming types correspond to the Matisse data types described in the following section.

The recommended Matisse programming types can be used in your program after including the `matisseCtx.h` file. These types are listed below:

MtBoolean This type is used to signal a condition that is `TRUE` or `FALSE`. There are two values defined to be `MtBoolean: MT_TRUE` and `MT_FALSE`.

MtChar This type is used to manage a character.

MtConfigurationType This type is used to specify the information to be retrieved by the function `MtCtxGetConfigurationInfo`. Only the following values are possible:

```
MT_MAX_BUFFERED_OBJECTS
MT_MAX_INDEX_CRITERIA_NUMBER
MT_MAX_INDEX_OID_LENGTH
```

See *GetConfigurationInfo, on page 74* for further information on this programming type.

MtContext This is an opaque structure resulting from a connection and is used for the functions `MtCtxAllocateContext, MtCtxFreeContext, MtCtxSetConnectionOption, MtCtxGetConnectionOption, MtCtxConnectDatabase, MtCtxDisconnectDatabase`.

MtDirection `MtDirection` indicates the order in which an index is scanned when a stream is opened on this type. There are only two possible values for this type, `MT_DIRECT` and `MT_REVERSE`. `MT_DIRECT` indicates that the index should be scanned from the first entry to the last. `MT_REVERSE` indicates that the index should be scanned from the last entry to the first.

MtDouble This type is used to manage double precision floating point numbers (64-bit).

MtFloat This type is used to manage a floating point numbers (32-bit).

MtIndexCriteriaInfo This type is used to store information retrieved by the functions `MtCtxMGetIndexInfo, MtCtx_MGetIndexInfo, MtCtxGetIndexInfo,` and `MtCtx_GetIndexInfo`.

`MtIndexCriteriaInfo` is a structure. It contains the following fields:

`MtOid indexOid` - the object identifier of the index

`MtSize nbOfCriteria` - the number of criteria

`criteria`- an array dimensioned as the maximum number of criteria. Each element of the array is also a structure describing a criterion:

- `MtOid attributeOid` - the object identifier of the criterion, which may be an attribute
- `MtType type` - the type of the criterion
- `MtInteger size` - the size of the criterion as described in the meta-schema
- `MtOrdering order` - the ordering of the index for the criterion, as described in the meta-schema.

MtOid This is a Matisse object identifier.

MtLock This is the type of lock set on a Matisse object or on an entry-point (`MT_READ` and `MT_WRITE`).

MtLockWaitTime This type is used to specify the wait-time for server access conflicts to be resolved. Two constants are defined with special values:

`MT_NO_WAIT:` if the lock cannot immediately be granted, it is released

`MT_WAIT_FOREVER:` wait until there is a deadlock or until the lock is be granted

MtOrdering This type indicates the direction that objects in an index are ordered (`MT_ASCEND` and `MT_DESCEND`).

MtShort This type is used to manage a signed 16 bit integer.

MtInteger This type is used to manage a signed 32-bit integer.

MtLong This type is used to manage a signed 64-bit integer.

MtServerExecutionPriority This type specifies the user priority for access to the database. Two constants are defined for specifying the legal range of values for this priority. These constants are `MT_MIN_SERVER_EXECUTION_PRIORITY` and `MT_MAX_SERVER_EXECUTION_PRIORITY`. Any value between these two values is valid.

MtSize This type is used by the functions `MtGetDataBytesReceived` and `MtGetDataBytesSent` to define a size such as the number of elements in an array or the size of an attribute or an object. `MtSize` is a 32-bit integer that assures compatibility with applications built on earlier versions of Matisse.

MtStream This is the stream used to manipulate objects.

MtString This type is used to manage a string (character array). It is defined as a pointer to `MtChar` (i.e., `typedef MtChar* MtString`).

MtSTS This status is returned by each Matisse function.

MtTimestamp This is a public structure used to handle dates and timestamps. The fields are as follows:

`MtShort year` - in range 1 to 8163

`MtShort month` - in range 1 to 12

```
MtShort day - in range 1 to 31
MtShort hour - in range 0 to 23
MtShort minute - in range 0 to 59
MtShort second - in range 0 to 59
MtInteger microsecs - in range 0 to 999999
```

MtInterval This is a public structure used to handle intervals of dates or timestamps. The fields are as follows:

```
MtShort sign - + or -
MtInteger days - in range 0 to 1491308
MtShort hours - in range 0 to 23
MtShort minutes - in range 0 to 59
MtShort seconds - in range 0 to 59
MtInteger microsecs - in range 0 to 999999
```

MtTimestampType This is a timestamp type enumeration. This type is limited to `MT_LOCAL_TIMESTAMP` and `MT_UNIVERSAL_TIMESTAMP` values.

MtTranPriority This type is used to specify the user priority for access conflict resolution. Two constants are defined to specify the legal range of values for transaction priority. These constants are `MT_MIN_TRAN_PRIORITY`, the minimum value for transaction priority and `MT_MAX_TRAN_PRIORITY`, the maximum value for transaction priority.

MtType This type contains a type of attribute value (enumeration of all types is described in the next chapter).

MtWhere This type is used to specify the location of a new successor. It can be specified with the following values:

`MT_FIRST`: The successor is added at the beginning of the existing list of successors

`MT_APPEND`: The successor is added at the end of the existing list of successors

`MT_AFTER`: The successor is added after the successor whose identifier is specified by the `where` argument.

This data type is used by the `Mt_AddSuccessor` and `MtAddSuccessor` functions.

2.2 Matisse Data Types

A Matisse attribute accepts the basic C language types. When you need to assign a value to a Matisse attribute, one of a predefined set of datatypes must be used.

To use this set, you must include the `matisseCtx.h` file.

Below is the list of all the data types that can be used to store an attribute value in Matisse:

MT_BOOLEAN	Boolean.
MT_BOOLEAN_LIST	Vector of booleans.
MT_CHAR	Extended ASCII character (0 to 255).
MT_DATE	Date (Year-Month-Day).
MT_DATE_LIST	Vector of dates.
MT_DOUBLE	Double precision floating point (IEEE format).
MT_DOUBLE_LIST	Vector of double floating point values (IEEE format)
MT_FLOAT	Single floating point (IEEE format).
MT_FLOAT_LIST	Vector of single floating point values (IEEE format).
MT_NULL	Represents “no value” (an empty list).
MT_SHORT	Signed integer stored on a maximum of 16 bits.
MT_SHORT_LIST	Vector of signed 16-bit integers.
MT_INTEGER	Signed integer stored on a maximum of 32 bits.
MT_INTEGER_LIST	Vector of signed 32-bit integers.
MT_LONG	Signed integer stored on a maximum of 64 bits.
MT_LONG_LIST	Vector of signed 64-bit integers.
MT_STRING, MT_TEXT	String made up of extended ASCII characters.
MT_TIME_INTERVAL	Timestamp interval (days hours:minutes:seconds.microseconds).
MT_TIME_INTERVAL_LIST	Vector of timestamp intervals.
MT_TIMESTAMP	Timestamp (Year-Month-Day Hour:Minute:seconds.microseconds).

MT_TIMESTAMP_LIST	Vector of timestamps.
MT_BYTE	Unsigned integer stored on a maximum of 8 bits.
MT_BYTES, MT_AUDIO, MT_VIDEO, MT_IMAGE	Vector of unsigned 8-bit integers.
MT_NUMERIC	Fixed precision value with maximum precision 19 and maximum scale 19. Default precision and scale is 19, 2.
MT_NUMERIC_LIST	Vector of fixed precision types.

2.3 Type Correspondences

All of the Matisse data types correspond to Matisse programming types. The following table shows the Matisse programming types and the Matisse data types to which they correspond.

Programming Type	Matisse Data Type
MtBoolean	MT_BOOLEAN
MtBoolean*	MT_BOOLEAN_LIST
MtChar	MT_CHAR
MtDouble	MT_DOUBLE
MtDouble*	MT_DOUBLE_LIST
MtFloat	MT_FLOAT
MtFloat*	MT_FLOAT_LIST
MtInterval	MT_INTERVAL
MtInterval*	MT_INTERVAL_LIST
MtShort	MT_SHORT
MtShort*	MT_SHORT_LIST
MtInteger	MT_INTEGER
MtInteger*	MT_INTEGER_LIST
MtLong	MT_LONG
MtLong*	MT_LONG_LIST
MtString, MtChar*	MT_STRING, MT_TEXT
MtString*	MT_STRING_LIST
MtTimestamp	MT_DATE, MT_TIMESTAMP
MtTimestamp*	MT_DATE_LIST, MT_TIMESTAMP_LIST
MtByte	MT_BYTE

Programming Type	Matisse Data Type
MtByte*	MT_BYTE_ARRAY, MT_BYTES, MT_AUDIO, MT_VIDEO, MT_IMAGE
MtNumeric	MT_NUMERIC
MtNumeric*	MT_NUMERIC_LIST

A variable declared as `MtType` may be set to any of the Matisse data types listed above.

All the array types may have up to eight dimensions.

3 Detailed API Reference

All of the C API functions begin with the prefix `MtCtx`. The first argument `ctx` of all function starting with `MtCtx` is of type `MtContex.`

Functions taking an `MtOid` (an object id) append a '_' to `MtCtx` prefix (i.e., `MtCtx_`). Because C doesn't support overloading, functions taking a string have only the `MtCtx` prefix.

For the set of Matisse `Get` functions, there are often four variants that perform almost identical operations with slightly different input or output arguments. These functions have the prefixes `MtCtxGet`, `MtCtx_Get`, `MtCtxMGet`, and `MtCtx_MGet`. The '`M`' following `MtCtx` or `MtCtx_`, signifies that memory is allocated by Matisse, whereas the functions without the '`M`' require that the programmer allocate memory before the function call.

For the set of Matisse functions that operate on several objects, there are often four variants that perform almost identical operations with slightly different input or output arguments. These functions have names which contain '`Num`'. The '`Num`' signifies that the function uses arrays instead of a variable number of arguments represented by the ellipsis punctuator ('...') in C language.

To use the functions described in the following text, you will need to include the `matisseCtx.h` file in your program.

AbortTransaction

Syntax

```
MtSTS MtCtxAbortTransaction (MtContext ctx)
```

Purpose

This function aborts the current transaction without committing any modifications.

Arguments

This function takes no arguments.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANS
MATISSE_TRANABORTED
```

Description

When this function is called, the transaction is aborted and the client cache is flushed.

See also

CommitTransaction (p. 48)
StartTransaction (p. 168)

AddSuccessor

Syntax

```
MtSTS MtCtxAddSuccessor
  (MtContext ctx, MtOid object,
   MtString relationshipName,
   MtOid successor,
   MtWhere where,
   ...)
MtSTS MtCtx_AddSuccessor
  (MtContext ctx, MtOid object,
   MtOid relationship,
   MtOid successor,
   MtWhere where,
   ...)
```

Purpose

This function adds a new successor to the relationship.

Arguments

object INPUT

An object.

relationshipName INPUT

A relationship name (in the form of a string).

*relationship*INPUT

A relationship object.

*successor*INPUT

The successor to be added.

where INPUT

The location where the new successor is to be added.

where can be specified with the following values:

MT_FIRST (the successor is added at the beginning of the existing list of successors)

MT_APPEND (the successor is added at the end of the existing list of successors)

MT_AFTER (the successor is added after the successor that is specified following the *where* argument).

Other INPUT arguments:

When the argument where is set to MT_AFTER, it must be followed by the successor after which the new successor is to be added.

Result

MATISSE_SUCCESS
MATISSE_ALREADYSUCC
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALCLASSMODIF9
MATISSE_INVALINDEXMODIF2
MATISSE_INVALINDEXMODIF4
MATISSE_INVALMODIF
MATISSE_INVALOP
MATISSE_INVALREL

```
MATISSE_INVALSTRINGSIZE
MATISSE_INVALSUPCLASS
MATISSE_INVALWHERE
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHFUNC
MATISSE_NOSUCHREL
MATISSE_NOSUCHSUCC
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_OVERRIDENVIOLATION
MATISSE_RELEXPECTED
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description

The location of the new successor depends on the value of the argument *where*:

- If *where* is set to MT_FIRST, the successor is added at the beginning of the existing list of successors.
- If *where* is set to MT_APPEND, the successor is added at the end of the existing list of successors.
- if *where* is set to MT_AFTER, the successor is added after the successor that is specified following the *where* argument.

Matisse preserves the order of the successors in a relationship. Functions such as MtCtxGetSuccessors retrieve the successors in the same order as they were stored.

Only the successors of a relationship defined in the data schema can be modified.

For each successor added to the relationship, the inverse relationship in the successor is added.

Modifications are validated and saved on the server during MtCtxCommitTransaction.

The name of relationships is not case sensitive.

These functions can be called only from within a transaction.

CAUTION: The objects of a database cannot reference those of another database through a Matisse relationship. If this situation occurs, Matisse generates an error.

See also *AddSuccessors* (p. 45)
GetAddedSuccessors (p. 58)

AddSuccessors

Syntax

```
MtSTS MtCtxAddNumSuccessors
  (MtContext ctx, MtOid object,
   MtString relationshipName,
   MtSize numSuccessors,
   MtOid* successors)
MtSTS MtCtx_AddNumSuccessors
  (MtContext ctx, MtOid object, MtOid relationship,
   MtSize numSuccessors,
   MtOid* successors)
MtSTS MtCtxAddSuccessors
  (MtContext ctx, MtOid object,
   MtString relationshipName,
   MtSize numSuccessors, ...)
MtSTS MtCtx_AddSuccessors
  (MtContext ctx, MtOid object, MtOid relationship,
   MtSize numSuccessors, ...)
```

Purpose These functions add new successors to the relationship. The new successors follow the successors already present in the object.

Arguments

object INPUT
An object.

*relationshipName*INPUT
A relationship name (a string).

*relationship*INPUT
A relationship object.

*numSuccessors*INPUT
The number of successors to be added.

*successors*INPUT
The array of the successors to be added.

Other INPUT arguments:
The argument *numSuccessors* must be followed by the successors (type MtOid) to be added.

Result

MATISSE_SUCCESS
MATISSE_ALREADYSUCC
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALCLASSMODIF9
MATISSE_INVALINDEXMODIF2
MATISSE_INVALINDEXMODIF4

```
MATISSE_INVALMODIF
MATISSE_INVALNB
MATISSE_INVALOP
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_INVALSUPCLASS
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHFUNC
MATISSE_NOSUCHREL
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_OVERRIDENVIOLATION
MATISSE_RELEXPECTED
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_UNEXPECTEDDUPLICATES
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description

Matisse preserves the order of the successors in a relationship. Functions such as `MtCtxGetSuccessors` retrieve the successors in the same order as they were stored.

Only the successors of a relationship defined in the data schema can be added.

For each successor added to the relationship, the inverse relationship in the successor is added.

Modifications are validated and saved on the server during `MtCtxCommitTransaction`.

The name of relationships is not case sensitive.

These functions can be called only from within a transaction.

CAUTION: The objects of a database cannot reference those of another database through a Matisse relationship. `MtOid` values always refer to objects of the currently selected database even when they have been retrieved during previous transactions with another database.

See also

AddSuccessor (p. 43)
GetAddedSuccessors (p. 58)

AllocateContext

Syntax
```
MtSTS MtCtxAllocateContext
  (MtContext* connection)
```

Purpose
This function allocates a connection with default options.

Arguments
*connection*OUTPUT
The structure that will contain all the information about the database connection.

Result
MATISSE_SUCCESS
MATISSE_MEMORYFAULT

Description
This function allocates a connection with default options. The options can be changed or retrieved by using `MtCtxSetConnectionOption` and `MtCtxGetConnectionOption` respectively.
The following sequence of actions must be implemented when accessing a database:

allocate a connection structure

establish the connection to the database

set the connection as current

execute operations on the database

deselect the connection

close the connection

free the connection structure

See also
ConnectDatabase (p. 50)
CurrentDate (p. 53)
DisconnectDatabase (p. 53)
FreeContext (p. 57)
GetConnectionOption (p. 74)
SetConnectionOption (p. 147)
SetListElements (p. 149)

CloseStream

Syntax
```
MtSTS MtCtxCloseStream (MtContext ctx, MtStream stream)
```

Purpose
This function closes the stream that is pointed to by *stream*.

Arguments
stream INPUT

An entry-point stream, a class stream, a relationship stream, an object attribute stream, an object relationship stream, or an object inverse relationship stream.

Result

```
MATISSE_SUCCESS
MATISSE_INVALSTREAM
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_STREAMCLOSED
```

Description

These functions can be called from within a transaction or during a version access.

See also

OpenInstancesStream (p. 132)
OpenEntryPointStream (p. 125)
OpenIndexEntriesStream (p. 126)
OpenPredecessorsStream (p. 137)
OpenAttributesStream (p. 124)
OpenInverseRelationshipsStream (p. 134)
OpenRelationshipsStream (p. 138)
OpenSuccessorsStream (p. 139)

CommitTransaction

Syntax

```
MtSTS MtCtxCommitTransaction
  (MtContext ctx, MtString prefix,
   MtString* versionName)
```

Purpose

This function terminates a transaction by committing any modification. Moreover, it allows you to save an instance view of the database for future accesses in version mode (refer to the function `MtCtxStartVersionAccess`).

Arguments

prefix INPUT

If you want to maintain a version of the database (for a future access in version mode), this argument must point to a string of no more than 20 characters. This string will facilitate the creation of a database version identifier at the end of the transaction.

If you do not want to maintain the current version of the database, the argument must be `NULL`.

*versionName*OUTPUT

If *prefix* is not `NULL`, this argument receives the database version identifier that is saved. It is made up of up to the first 20 characters (maximum) of *prefix* followed by a hexadecimal number. This string will reference the version in `MtCtxStartVersionAccess`. Note that Matisse allocates memory for this string automatically then returns a pointer to the allocated memory.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INCOMPCRITERIANUMBER
MATISSE_INCOMPCRITERIASIZE
MATISSE_INVALARG
MATISSE_INVALATTMODIF1
MATISSE_INVALATTMODIF2
MATISSE_INVALATTMODIF4
MATISSE_INVALATTMODIF5
MATISSE_INVALATTREMOVE
MATISSE_INVALATTTYPE
MATISSE_INVALCARDINALITY
MATISSE_INVALCLASSMODIF1
MATISSE_INVALCLASSMODIF2
MATISSE_INVALCLASSMODIF4
MATISSE_INVALCLASSMODIF5
MATISSE_INVALCLASSMODIF6
MATISSE_INVALCLASSMODIF7
MATISSE_INVALCLASSMODIF
MATISSE_INVALCLASSMODIF11
MATISSE_INVALCRITERIACLASS
MATISSE_INVALCRITERIAORDER
MATISSE_INVALCRITERIASIZE
MATISSE_INVALCRITERION
MATISSE_INVALNAMESIZE
MATISSE_INVALOP
MATISSE_INVALRELDELETE
MATISSE_INVALRELMODIF1
MATISSE_INVALRELMODIF2
MATISSE_INVALRELMODIF3
MATISSE_INVALRELMODIF4
MATISSE_INVALRELMODIF5
MATISSE_INVALRELREMOVE
MATISSE_INVALSTRINGSIZE
MATISSE_INVALSUCCESSOR
MATISSE_INVALSUCCREMOVE
MATISSE_INVALSUCCSNB
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHFUNC
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
MATISSE_WRITEWAITTIME
```

Description When an object is validated, for each object property that has been modified, Matisse checks the structural constraints (the attribute `MtType` for an attribute, the attribute `MtCardinality`, and the relationship `MtCtxSuccessors` for a relationship, etc.).

If an error occurs while the object is being committed, the variable `mtInvalidObject` is set to the object that causes the error.

`MATISSE_WAITTIME` occurs only if there is a read lock when the objects are being checked. If write locks cannot be acquired while the objects are being written, the `MATISSE_WRITEWAITTIME` error occurs. No additional modifications (i.e. create, update, or delete operations) are allowed even if the transaction is not committed or aborted. All modification functions will return `MATISSE_INVALOP` until the end of the transaction (when either `MtCtxCommitTransaction` or `MtCtxAbortTransaction` returns `MATISSE_SUCCESS`).

When the transaction is aborted, the client cache is flushed.

This function can be called only from within a transaction.

See also *AbortTransaction* (p. 42)
IntervalAdd (p. 102)
StartTransaction (p. 168)
StartVersionAccess (p. 169)

ConnectDatabase

Syntax

```
MtSTS MtCtxConnectDatabase
   (MtContext connection,
    MtString host,
    MtString databaseName,
    MtString userName,
    MtString password)
```

Purpose

This function opens a database connection.

Arguments

*connection*OUTPUT
: A previously allocated structure that will contain all the information on the database connection.

host INPUT
: The location of the database host.

*databaseName*INPUT
: The name of the database to connect to.

*userName*INPUT
: The name of the database user which may be set to NULL. If this is the case, the login name of the user is used.

*password*INPUT
: The user password. Can be set to NULL only if the user name is also NULL.

Result

```
MATISSE_SUCCESS
MATISSE_INVALCONNECTION
MATISSE_OPDENIED
MATISSE_INVALUSERNAMELEN
MATISSE_INVALPASSWDLEND
MATISSE_INVALPASSWD
MATISSE_CONNECTREJECT
MATISSE_CONNLOST
MATISSE_CONNTIMEOUT
MATISSE_DBNAMETOOLONG
MATISSE_DBNOTINIT
MATISSE_INCOMPVERSION
MATISSE_INVALOP
MATISSE_INVTRANSPORT
MATISSE_NOFREETOKEN
MATISSE_NOPMADDR
MATISSE_NOSUCHDB
MATISSE_NOSUCHHOST
MATISSE_PMCONFAILED
MATISSE_STREAMCLOSED
MATISSE_TRANSDISABLED
```

Description

The following sequence of actions must occur when accessing a database:

allocate a connection structure

establish the connection to the database

set the connection as current

execute operations on the database

deselect the connection

close the connection

free the connection structure

.As previously stated, a single client application may provide access to several databases. In this case the user will open one connection per database.

Connections to several different databases can be opened simultaneously.

Once the database is selected, the client has direct access to the data, either from within a transaction or within version access.

Example

```
MtContext connection;
MtCtxAllocateContext(&connection);
MtCtxConnectDatabase
   (&connection, "myhost","mydb",
    NULL, NULL) ;
```

This example shows a connection to the database called `"mydb"` on the machine `"myhost"`. The database user name is set to the value of the current login name (due to the two NULL parameters).

See also
AllocateContext (p. 47)
DisconnectDatabase (p. 53)
FreeContext (p. 57)
SetConnectionOption (p. 147)

CreateObject

Syntax

```
MtSTS MtCtxCreateNumObjects
  (MtContext ctx, MtSize numObjects, MtOid* objects,
   MtString className)
MtSTS MtCtx_CreateNumObjects
  (MtContext ctx, MtSize numObjects, MtOid* objects,
   MtOid class)
MtSTS MtCtxCreateObject
  (MtContext ctx, MtOid* object, MtString className)
MtSTS MtCtx_CreateObject
  (MtContext ctx, MtOid* object, MtOid class)
```

Purpose

These functions create one or more Matisse objects of the class *className* (or *class*, depending on the function being used).

Arguments

*numObjects*INPUT
The number of objects to create.

*objects*OUTPUT
Table of objects allocated by the user.

object OUTPUT
The created object.

*className*INPUT
A class name.

class INPUT
A class identifier.

Result

```
MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_EXCEEDSLIMIT
MATISSE_DEADLOCKABORT
MATISSE_INVALCREATION
MATISSE_INVALMODIF
MATISSE_INVALNB
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
```

```
MATISSE_OBJECTNOTFOUND
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description

The value of the *numObjects* argument must not exceed the limit specified by the function `MtCtxGetConfigurationInfo` applied to the argument `MT_MAX_BUFFERED_OBJECTS`.

The name of the class is not case sensitive.

These functions can be called only from within a transaction.

CurrentDate

Syntax

```
MtTimestamp MtCurrentDate()
```

Purpose

This functions returns the current date.

Arguments

`None.`

Result

An `MtTimestamp` structure with the `hour`, `minute`, `second` and `microsecs` fields set to 0.

See also *DisconnectDatabase* (p. 53)

DisconnectDatabase

Syntax

```
MtSTS MtCtxDisconnectDatabase (MtContext connection)
```

Purpose

This function closes the connection. Any data read from the database is flushed from the client cache.

Arguments

connection`INPUT`

The structure that contains information specific to the database (previously initialized using the function `MtCtxConnectDatabase`).

Result

```
MATISSE_SUCCESS
MATISSE_INVALCONNECTIONSTATE
MATISSE_INVALCONNECTION
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
```

The following sequence of actions must be implemented when accessing a database:

allocate a connection structure

establish the connection to the database

set the connection as current

execute some operations on the database

deselect the connection

close the connection

free the connection structure

See also *AllocateContext* (p. 47)
ConnectDatabase (p. 50)
FreeContext (p. 57)

EndVersionAccess

Syntax
```
MtSTS MtCtxEndVersionAccess (MtContext ctx)
```

Purpose This function ends a version mode access on the database. Once the version mode access is terminated, you can start another version access or a transaction access.

Result
```
MATISSE_SUCCESS
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOVERSIONACCESS
```

See also *StartVersionAccess* (p. 169)

Error

Syntax
```
MtString  MtCtxError (MtContext ctx)
```

Purpose This function returns the string associated with the latest Matisse error.

Result None.

See also *MakeUserError* (p. 111)

EventNotify

Syntax
```
MtSTS MtCtxEventNotify(MtContext ctx, MtEvent event)
```

Purpose This function triggers an event.

Arguments *event*INPUT

The event to be triggered.

Result MATISSE_SUCCESS

See also *EventWait* (p. 55), *EventSubsribe* (p. 55)

EventSubsribe

Syntax
```
MtSTS MtCtxEventSubscribe(MtContext ctx, MtEvent
postedEvents)
```

Purpose This function subscribes to a list of events. Once the subscription is done, all the events that occurred are logged for this subscriber. You are notified that an event occurs by using the MtEventWait function.

Arguments *postedEvents*INPUT

A list of events (MT_EVENT1 | MT_EVENT8).

Result MATISSE_SUCCESS
MATISSE_EVENTSUBSCRIBEFAIL

See also *EventUnsubscribe* (p. 55)

EventUnsubscribe

Syntax
```
MtSTS MtCtxEventUnsubscribe(MtContext ctx)
```

Purpose This function un-subscribes all events that you have subscribed to.

Arguments

Result MATISSE_SUCCESS
MATISSE_NOEVENTACTIVE

See also *EventSubsribe* (p. 55)

EventWait

Syntax
```
MtSTS MtCtxEventWait
  (MtContext ctx, MtLockWaitTime timeout, MtEvent
*triggeredEvents)
```

Purpose This functions remove objects from the client cache and reclaim memory space.

Arguments

*timeout*INPUT

A wait-time in milli-seconds or MTWAIT_FOREVER.

*triggeredEvents*OUTPUT

The triggered events.

Result

MATISSE_SUCCESS
MATISSE_NOEVENTACTIVE
MATISSE_TIMEOUT

See also *EventNotify* (p. 54)

Failure

Syntax

```
int  MtFailure (MtSTS status)
```

Purpose

This macro indicates whether a Matisse function has completed successfully.

Arguments

status INPUT

The status returned by a Matisse function.

Result

0 if the status corresponds to a success; a nonnull integer otherwise.

See also *Success* (p. 170)

Free

Syntax

```
MtSTS MtMFree(void* value)
```

Purpose

This function frees the memory allocated by the functions MtMGet*XXX* and Mt_MGet*XXX*.

Arguments

value INPUT

A value allocated by one of the following functions: MtMGet*XXX* or Mt_MGet*XXX*.

Result

MATISSE_SUCCESS

Description

When a program calls one of the Matisse functions beginning with the letters Mt_MGet or MtMGet, Matisse allocates memory to store the value. When the value is no longer needed, the program must free the value using the MtMFree function.

FreeContext

Syntax

```
MtSTS MtCtxFreeContext
  (MtContext connection)
```

Purpose

This function frees a previously allocated connection structure.

Arguments

*connection*OUTPUT

A connection structure previously allocated by `MtCtxAllocateContext`.

Result

```
MATISSE_SUCCESS
MATISSE_INVALCONNECTION
MATISSE_INVALOP
MATISSE_INVALCONNECTIONSTATE
```

Description

This function frees the connection structure previously allocated by `MtCtxAllocateContext`. This function cannot be called if a database connection is currently opened.
The following sequence of actions must be implemented when accessing a database:

- allocate a connection structure
- establish the connection to the database
- set the connection as current
- execute the required operations on the database
- deselect the connection
- close the connection
- free the connection structure

See also

AllocateContext (p. 47)
ConnectDatabase (p. 50)
CurrentDate (p. 53)
DisconnectDatabase (p. 53)
GetConnectionOption (p. 74)
SetConnectionOption (p. 147)
SetListElements (p. 149)

FreeObjects

Syntax

```
MtSTS MtCtxFreeNumObjects
  (MtContext ctx, MtSize numObjects, MtOid* objects)
MtSTS MtCtxFreeObjects
  (MtContext ctx, MtSize numObjects, ...)
```

Purpose

These functions remove objects from the client cache and reclaim memory space.

Arguments

*numObjects*INPUT

The number of objects to be freed.

*objects*INPUT

An array that contains the objects to be freed. The programmer is responsible for the memory space associated with the array.

Other INPUT arguments:

For MtCtxFreeObjects, the argument *numObjects* is followed by the objects to be freed. The object identifiers should be of type MtOid*.

Result

MATISSE_SUCCESS
MATISSE_INVALNB
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_UNLOADABLEOBJECT

Description

As the objects are loaded in cache, the local objects table enlarges and the available memory space decreases. The objects are freed from the objects table at the end of the transaction only. Depending on user's needs, however, it may prove useful to free objects that are no longer used during the transaction to make room for other objects.

Schema objects or objects that have been modified during the transaction cannot be removed from the cache.

If an object specified as an argument is not loaded or does not exist, no error is generated.

Freeing objects is an atomic operation: if MATISSE_SUCCESS is returned, all the objects have been freed. If an error is returned, no objects have been freed.

MtCtxFreeNumObjects can be called from within a transaction or during a version access.

GetAddedSuccessors

Syntax

```
MtSTS MtCtxGetAddedSuccessors
  (MtContext ctx, MtSize* numAddedSuccessors,
   MtOid* allAddedSuccessors,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_GetAddedSuccessors
  (MtContext ctx, MtSize* numAddedSuccessors,
   MtOid* allAddedSuccessors,
   MtOid object, MtOid relationship)
```

```
MtSTS MtCtxMGetAddedSuccessors
  (MtContext ctx, MtSize* numAddedSuccessors,
   MtOid** allAddedSuccessors,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_MGetAddedSuccessors
  (MtContext ctx, MtSize* numAddedSuccessors,
   MtOid** allAddedSuccessors,
   MtOid object, MtOid relationship)
```

Purpose These functions act through a relationship to retrieve the successors of an object that have been added during the current transaction.

Arguments

*numAddedSuccessors*INPUT/OUTPUT

In input, this parameter determines the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of objects (i.e. `MtCtxGetAddedSuccessors` and `MtCtx_GetAddedSuccessors`).

In output, this parameter gives the number of successors that have been added during the transaction.

*allAddedSuccessors*OUTPUT/INPUT

For the functions `MtCtxGetAddedSuccessors` and `MtCtx_GetAddedSuccessors` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the successors of *object* added during the current transaction through the relationship specified by *RelationshipName* or *relationship*.

For the functions `MtCtxMGetAddedSuccessors` and `MtCtx_MGetAddedSuccessors`, which allocate memory, this argument is a pointer to an array allocated by Matisse. For these functions, the program must declare a pointer to `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the buffer that lists the successors of *object* added during the current transaction.

allAddedSuccessors can be set to `NULL`, in which case the function simply returns the number of successors added during the current transaction.

object INPUT

An object.

*relationshipName*INPUT

A relationship name.

relationship INPUT

The object relationship.

Result

`MATISSE_SUCCESS`
`MATISSE_ARRAYTOOSMALL`
`MATISSE_CONNLOST`
`MATISSE_DEADLOCKABORT`

```
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_WAITTIME
```

Description

The names of relationships are not case sensitive. These functions can be called either from within a transaction or during a version access. During version access, however, these functions are not useful since they deal with the addition of successors inside the current transaction and always return *numAddedSuccessors* with a value of 0.

The functions MtCtxGetAddedSuccessors and MtCtx_GetAddedSuccessors do not allocate an array to store the object successors added during the current transaction through a relationship. The calling program can allocate an array of type MtOid and then pass this array as the *allAddedSuccessors* argument.

The functions MtCtxMGetAddedSuccessors and MtCtx_MGetAddedSuccessors allocate an array to store all the identifiers found. When calling these functions, a program must pass as the *allAddedSuccessors* argument, the address of a pointer to MtOid. In output, this argument will point to an array that contains the objects. To free the memory space allocated for the array, the program can call the standard C function: free.

See also

AddSuccessor (p. 43)
AddSuccessors (p. 45)

GetAllAttributes

Syntax

```
MtSTS MtCtxGetAllAttributes
  (MtContext ctx, MtSize* numAttributes,
   MtOid* attributes,
   MtString className)
MtSTS MtCtx_GetAllAttributes
  (MtContext ctx, MtSize* numAttributes,
   MtOid* attributes,
   MtOid class)
MtSTS MtCtxMGetAllAttributes
  (MtContext ctx, MtSize* numAttributes,
   MtOid** attributes,
   MtString className)
```

```
MtSTS MtCtx_MGetAllAttributes
   (MtContext ctx, MtSize* numAttributes,
    MtOid** attributes,
    MtOid class)
```

Purpose These functions retrieve all the attributes of the class including those attributes defined in the superclasses of the class.

Arguments

numAttributes INPUT/OUTPUT

In input, this parameter contains the size of the array (specified by the user). This parameter can be used as an input argument only by those functions that do not allocate memory for the array of objects (i.e. `MtCtxGetAllAttributes` and `MtCtx_GetAllAttributes`.)

In output, this parameter contains the number of attributes returned by the function.

attributes OUTPUT/INPUT

For the functions `MtCtxGetAllAttributes` and `MtCtx_GetAllAttributes` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the attributes of the *class* and its superclasses.

For the functions `MtCtxMGetAllAttributes` and `MtCtx_MGetAttributes` which allocate memory, this argument is a pointer to a buffer allocated by Matisse. The calling program must declare a pointer to `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the buffer that lists the attributes of *class* and its superclasses recursively.

This parameter can be set to `NULL`, in which case the function returns the number of attributes of *class* and its superclasses.

className INPUT

A class name.

class INPUT

A class object.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The names of classes are not case sensitive. These functions can be called either from within a transaction or during a version access.

The functions `MtCtxGetAllAttributes` and `MtCtx_GetAllAttributes` do not allocate an array to store the attributes. The calling program must allocate an array of type `MtOid` and then pass this array as the *`attributes`* argument.

The functions `MtCtxMGetAllAttributes` and `MtCtx_MGetAllAttributes` allocate an array to store all the identifiers that are found. When calling these functions, a program must pass as the *`attributes`* argument the address of a pointer to `MtOid`. In output, this argument will point to an array that contains the object identifiers. To free the memory space allocated for the array, the program can call the standard C function: `free`.

See also *OpenAttributesStream* (p. 124)

GetAllInverseRelationships

Syntax

```
MtSTS MtCtxGetAllInverseRelationships
  (MtContext ctx, MtSize* numIRelationships,
   MtOid* iRelationships,
   MtString className)
MtSTS MtCtx_GetAllInverseRelationships
  (MtContext ctx, MtSize* numIRelationships,
   MtOid* iRelationships,
   MtOid class)
MtSTS MtCtxMGetAllInverseRelationships
  (MtContext ctx, MtSize* numIRelationships,
   MtOid** iRelationships,
   MtString className)
MtSTS MtCtx_MGetAllInverseRelationships
  (MtContext ctx, MtSize* numIRelationships,
   MtOid** iRelationships,
   MtOid class)
```

Purpose These functions retrieve all the possible inverse relationships of the class specified by *`class`* and its subclasses.

Arguments

`numIRelationships``INPUT/OUTPUT`

In input, this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of identifiers (i.e. `MtCtxGetAllInverseRelationships` and `MtCtx_GetAllInverseRelationships`.

In output, this parameter contains the number of inverse relationships that have been retrieved by the functions.

`iRelationships``OUTPUT/INPUT`

For the functions `MtCtxGetAllInverseRelationships` and `MtCtx_GetAllInverseRelationships` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the possible inverse relationships of *`class`* and of any subclass of *`class`*.

For the functions `MtCtxMGetAllInverseRelationships` and `MtCtx_MGetAllInverseRelationships` which allocate memory, this argument is a pointer to an array allocated by Matisse. The program must allocate a pointer to `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to these functions. In output, this pointer contains the address of the array that lists the possible inverse relationships of *`class`* and of any subclass of *`class`*.

This parameter can be set to `NULL`, in which case the function simply returns the number of possible inverse relationships of *`class`* and its subclasses.

`className``INPUT`

A class name.

`class` `INPUT`

A class object.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

Note that when a class has a possible inverse relationship:

The relationship is not defined for the class.

The relationship has an inverse relationship for which the class is a possible successor.

Every relationship also has an inverse relationship. For each relationship, you must define a value for the relationship `MtCtxInverseRelationship`. The value assigned to the `MtCtxInverseRelationship` is the relationship's *`inverse relationship`*.

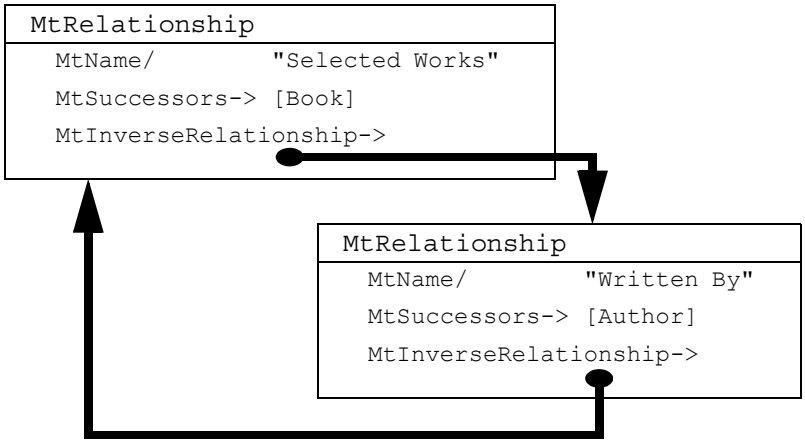
Example Suppose in a schema the following two classes are defined:

```
MtClass
  MtName/                "Author"
  MtAttributes->         Last Name/
  MtRelationships->
```

```
MtClass
  MtName/                "Book"
  MtAttributes->         Title/
  MtRelationships-> Written By->
```

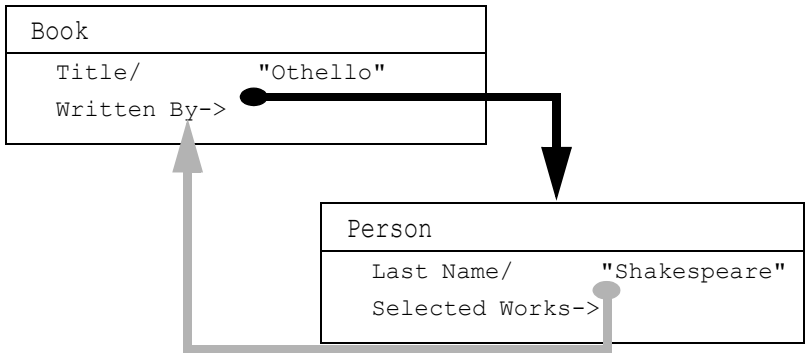
Note that class `Book` defines a relationship `Written By`. This relationship, in turn, defines an inverse relationship.

The following diagram illustrates the definitions of the relationship `Written By` and its inverse relationship, `Selected Works`:



Consider that one instance of `Author` and one instance of `Book` are created, and that for the instance of `Book`, the value of `Written By` is assigned to the instance of `Author`.

The following diagram illustrates the resulting link established between an instance of class `Book` and an instance of class `Author` through the relationship `Written By`:



For the instance `Othello`, the relationship `Written By` is assigned to the instance `Shakespeare`.

As you can see, the inverse relationship `Selected Works` is also implied for the instance of `Shakespeare`.

A stream opened by the function `MtCtxOpenInverseRelationshipsStream` retrieves only those inverse relationships that exist for an object. An object inverse relationship stream opened on the instance `Irving`, for example, will retrieve the inverse relationship `Selected Works`.

Listing Possible Inverse Relationships

It can sometimes prove useful to determine all of the inverse relationships that instances of a particular class can have. You can retrieve this information with the `GetAllInverseRelationships` functions. These functions return a list of all the possible inverse relationships of a class.

The name of classes is not case sensitive. These functions can be called either from within a transaction or during a version access.

The `MtCtxGetAllInverseRelationships` and `MtCtx_GetAllInverseRelationships` functions do not allocate an array to store the possible inverse relationships of *`class`*. The calling program must allocate an array of type `MtOid`, then pass the address of this array as its *`iRelationships`* argument.

The `MtCtxMGetAllInverseRelationships` and `MtCtx_MGetAllInverseRelationships` functions allocate an array to store all the possible inverse relationships that are found. When calling these functions, a program must pass as its *`iRelationships`* argument the address of a pointer to `MtOid`. In output, this argument will point to an array that contains the object identifiers. To free the memory space allocated for the array, the program must call the `free` standard C function.

See also *OpenInverseRelationshipsStream* (p. 134)

GetAllRelationships

Syntax

```
MtSTS MtCtxGetAllRelationships
  (MtSize* numRelationships,
   MtOid* relationships,
   MtString className)
MtSTS MtCtx_GetAllRelationships
  (MtSize* numRelationships,
   MtOid* relationships,
   MtOid class)
MtSTS MtCtxMGetAllRelationships
  (MtSize* numRelationships,
   MtOid** relationships,
   MtString className)
MtSTS MtCtx_MGetAllRelationships
  (MtSize* numRelationships,
   MtOid** relationships,
   MtOid class)
```

Purpose

These functions return all the relationships defined in the class and its superclasses.

Arguments

*numRelationships*INPUT/OUTPUT

In input, this argument contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array (i.e. `MtCtxGetAllInverseRelationships` and `MtCtx_GetAllInverseRelationships`.)

In output, this argument contains the number of relationships of the class and its superclasses.

*relationships*OUTPUT/INPUT

For the functions `MtCtxGetAllRelationships` and `MtCtx_GetAllRelationships` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the relationships of *`className`* or *`class`* and the relationships of any superclass of *`className`* or *`class`* recursively.

For the functions `MtCtxMGetAllRelationships` and `MtCtx_MGetAllRelationships` which allocate memory, this argument is a pointer to an array allocated by Matisse. The program must declare a pointer to `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to these functions. In output, this pointer contains the address of the array that lists the relationships of *`class`* or *`className`* and of any superclass of *`class`* or *`className`*.

This parameter can be set to `NULL`, in which case, the function simply returns the number of relationships of *`class`* and its superclasses.

*className*INPUT

A class name.

class INPUT

A class object.

Result

MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

The names of the classes are not case sensitive. These functions can be called either from within a transaction or during a version access.

The functions `MtCtxGetAllRelationships` and `MtCtx_GetAllRelationships` do not allocate an array to store the relationships. The calling program must allocate an array of type `MtOid`, then pass this array as its *`relationships`* argument.

The functions `MtCtxMGetAllRelationships` and `MtCtx_MGetAllRelationships` allocate an array to store the identifiers of all the relationships found. When calling these functions, a program must pass as its *`relationships`* argument the address of a pointer to `MtOid`. In output, this argument will point to an array that contains the relationships. To free the memory space allocated for the array, the program must call the standard C function: `free`.

See also *OpenRelationshipsStream* (p. 138)

GetAllSubclasses

Syntax

```
MtSTS MtCtxGetAllSubclasses
  (MtContext ctx, MtSize* numSubclasses,
   MtOid* subclasses,
   MtString className)
MtSTS MtCtx_GetAllSubclasses
  (MtContext ctx, MtSize* numSubclasses,
   MtOid* subclasses,
   MtOid class)
```

```
MtSTS MtCtxMGetAllSubclasses
  (MtContext ctx, MtSize* numSubclasses,
   MtOid** subclasses,
   MtString className)
MtSTS MtCtx_MGetAllSubclasses
  (MtContext ctx, MtSize* numSubclasses,
   MtOid** subclasses,
   MtOid class)
```

Purpose These functions retrieve the subclasses of *class* (those defined in the class and in its subclasses).

Arguments

numSubclasses INPUT/OUTPUT

In input, this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of identifiers (i.e `MtCtxGetAllSubclasses` and `MtCtx_GetAllSubclasses`).

In output, this parameter contains the number of subclasses returned by the function.

subclasses OUTPUT/INPUT

For the functions `MtCtxGetAllSubclasses` and `MtCtx_GetAllSubclasses` which do not allocate memory, this argument is an array declared in the calling program. After the function is called, this array will contain the subclasses of *class* or *className* and their subclasses recursively.

For the functions `MtCtxMGetAllSubclasses` and `MtCtx_MGetAllSubclasses` which allocate memory, this argument is a pointer to an array allocated by Matisse. The program must declare a pointer to an `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the buffer that lists the subclasses of *class* and their subclasses recursively.

This parameter can be set to `NULL`, in which case the function returns the number of subclasses of *class* and its subclasses.

className INPUT

A class name.

class INPUT

A class identifier.

Result

MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER

```
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of the class is not case sensitive. These functions can be called either from within a transaction or during a version access.

The functions `MtCtxGetAllSubclasses` and `MtCtx_GetAllSubclasses` do not allocate an array to store the subclasses of *`class`*. The calling program can allocate an array of type `MtOid` and pass the address of this array as its *`subclasses`* argument.

The functions `MtCtxMGetAllSubclasses` and `MtCtx_MGetAllSubclasses` allocate an array to store all the objects found. When calling these functions, a program must pass as its *`subclasses`* argument, the address of a pointer to an `MtOid`. In output, this argument will point to an array that contains the subclasses. To free the memory space allocated for the array, the program can call the standard C function: `free`.

GetAllSuperclasses

Syntax

```
MtSTS MtCtxGetAllSuperclasses
  (MtContext ctx, MtSize* numSuperclasses,
   MtOid* superclasses,
   MtString className)
MtSTS MtCtx_GetAllSuperclasses
  (MtContext ctx, MtSize* numSuperclasses,
   MtOid* superclasses,
   MtOid class)
MtSTS MtCtxMGetAllSuperclasses
  (MtContext ctx, MtSize* numSuperclasses,
   MtOid** superclasses,
   MtString className)
MtSTS MtCtx_MGetAllSuperclasses
  (MtContext ctx, MtSize* numSuperclasses,
   MtOid** superclasses,
   MtOid class)
```

Purpose

These functions retrieve the superclasses of *`class`*—both those defined in the class and those defined in the superclasses of *`class`*.

Arguments

`numSuperclasses``INPUT/OUTPUT`

In input, this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of identifiers (i.e. `MtCtxGetAllSuperclasses` and `MtCtx_GetAllSuperclasses`)

In output, this parameter contains the number of superclasses returned by the function.

superclasses OUTPUT/INPUT

For the functions `MtCtxGetAllSuperclasses` and `MtCtx_GetAllSuperclasses` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the superclasses of *class* or *className* and their superclasses recursively.

For the functions `MtCtxMGetAllSuperclasses` and `MtCtx_MGetAllSuperclasses` which allocate memory, this argument is a pointer to an array allocated by Matisse. The program must declare a pointer to an `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the array that lists the superclasses of *class* and their superclasses recursively.

This parameter can be set to `NULL`, in which case the function returns the number of superclasses of *class* and its superclasses.

className INPUT

A class name.

class INPUT

A class identifier.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of the class is not case sensitive. These functions can be called either from within a transaction or during a version access.

The functions `MtCtxGetAllSuperclasses` and `MtCtx_GetAllSuperclasses` do not allocate an array to store the superclasses of a class. The calling program must allocate an array of type `MtOid`, then pass this array as its *superclasses* argument.

The functions `MtCtxMGetAllSuperclasses` and `MtCtx_MGetAllSuperclasses` allocate an array to store all the objects found. When calling these functions, a program must pass as its *superclasses*

argument the address of a pointer to an `MtOid`. In output, this argument will point to an array that contains the superclasses. To free the memory space allocated for the array, the program must call the standard C function: `free`.

GetAttribute

Syntax

```
MtSTS MtCtxGetAttribute
  (MtContext ctx, MtOid* attribute,
   MtString attributeName)
```

Purpose

This function returns the schema descriptor for the attribute identified by*attributeName*.

Arguments

*attribute*OUTPUT

The attribute whose name is *attributeName*.

*attributeName*INPUT

An attribute name.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_MULTIPLYDEFINED
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The names of attributes are not case sensitive. This function can be called either from within a transaction or during a version access.

GetClass

Syntax

```
MtSTS MtCtxGetClass
  (MtContext ctx, MtOid* class, MtString className)
```

Purpose

This function returns the schema descriptor for the class identified by *className*.

Arguments

class OUTPUT

The class whose name is *className*.

*className*INPUT

A class name.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The names of the classes are not case sensitive. This function can be called either from within a transaction or during a version access.

See also *GetAttribute* (p. 71)
GetClassAttribute (p. 72)
GetClassRelationship (p. 73)
GetRelationship (p. 90)

GetClassAttribute

Syntax

```
MtSTS MtCtxGetClassAttribute
  (MtContext ctx, MtOid* attribute,
   MtString className,
   MtString attributeName)
MtSTS MtCtx_GetClassAttribute
  (MtContext ctx, MtOid* attribute,
   MtOid classOid,
   MtString attributeName)
```

Purpose This function returns the schema descriptor for the attribute identified by*attributeName and defined for the class className*.

Arguments

*attribute*OUTPUT

The attribute whose name is *attributeName*.

*className*INPUT

A class name.

*classOid*INPUT

A class object.

*attributeName*INPUT

An attribute name.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
```

```
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The names of attributes are not case sensitive. This function can be called either from within a transaction or during a version access.

GetClassRelationship

Syntax

```
MtSTS MtCtxGetClassRelationship
  (MtContext ctx, MtOid* relationship,
   MtString className,
   MtString relationshipName)
MtSTS MtCtx_GetClassRelationship
  (MtContext ctx, MtOid* relationship,
   MtOid classOid,
   MtString relationshipName)
```

Purpose This function returns the relationship identified by *relationshipName and defined for the class className*.

Arguments

*relationship*OUTPUT

The relationship whose name is *relationshipName*.

*className*INPUT

A class name.

*classOid*INPUT

A class object.

*relationshipName*INPUT

A relationship name.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The names of the relationships are not case sensitive. This function can be called either from within a transaction or during a version access.

GetConfigurationInfo

Syntax

```
MtSize  MtCtxGetConfigurationInfo
  (MtContext ctx, MtConfigurationType type)
```

Purpose

This function provides information on the configuration of Matisse.

Arguments

type INPUT

An Oid that indicates the kind of information to be retrieved. The following two keys are accepted:

MT_MAX_BUFFERED_OBJECTS, MT_MAX_INDEX_KEY_LENGTH

Description

The following table lists the information returned for each key. Any other input value is invalid. If an invalid value is entered, the function returns an invalid value.

Key	Purpose
MT_MAX_BUFFERED_OBJECTS	Returns the maximum number of objects that can be passed as a parameter to the functions: MtCtxCreateNumObjects, MtCtxLoadNumObjects, MtCtxLoadObjects, MtCtxLockNumObjects and MtCtxLockObjects.
MT_MAX_INDEX_KEY_LENGTH	Returns the maximum size of an index key to be returned.

GetConnectionOption

Syntax

```
MtSTS MtCtxGetConnectionOption
  (MtContext connection,
   MtConnectionOption option, ...)
```

Purpose

This function retrieves the value associated with a connection option.

Arguments

*connection*OUTPUT

A previously allocated structure that contains information about the database connection.

*option*INPUT

The connection option. Possible values are:

MT_SERVER_EXECUTION_PRIORITY, MT_LOCK_WAIT_TIME, MT_DATA_ACCESS_MODE. INPUT

The other input arguments are option specific. For a full description, see below.

Result

MATISSE_SUCCESS
MATISSE_INVALCONNECTOPTION
MATISSE_INVALCONNECTION
MATISSE_NOCURRENTCONNECTION

Description

Connection options affect the way you can interact with the database. You can retrieve the values for the following options:

- MT_DATA_ACCESS_MODE. The associated value indicates the type of access that is required on the database. You need to specify a pointer to a MtDataAccessMode value to retrieve the value currently associated. The options for the value are:
  - MT_DATA_READONLY indicating restricted read only access to the data.
  - MT_DATA_MODIFICATION indicating that read/write access is allowed for the data objects and read only access is allowed for schema and meta-schema objects.
  - MT_DATA_DEFINITION indicating that read/write access is allowed for data objects, schema and meta-schema objects.

    The first two access modes optimize access to the schema. The DATA_DEFINITION access mode should be used only when schema or meta-schema updates are necessary.
    This option cannot be changed when the connection to the database is open.
- MT_LOCK_WAIT_TIME. The associated value indicates the amount of time (in milliseconds) the server will wait for access conflicts to be resolved; if the wait time is exceeded, the explicit or implicit lock request is rejected. You need to specify a pointer to a MtLockWaitTime value to retrieve the currently associated value.
- MT_SERVER_EXECUTION_PRIORITY indicates the priority of the requests the connection sends to the database server. The higher the priority, the faster the requests are executed. You must specify a pointer to a MtServerExecutionPriority value to retrieve the currently associated value. The possible values for MtServerExecutionPriority are: MT_MIN_SERVER_EXECUTION_PRIORITY, MT_NORMAL_SERVER_EXECUTION_PRIORITY, MT_ABOVE_NORMAL_SERVER_EXECUTION_PRIORITY or MT_MAX_SERVER_EXECUTION_PRIORITY.
- MT_MEMORY_TRANSPORT. This option allows use of the shared memory transport rather than tcp or ticots for local access. The connection is first opened using tcp or ticots, then if shared memory resources are available on the machine, the connection is reopened in shared memory. The possible values are:
  - MT_OFF (default): Does not allow shared memory transport for local connection. This option cannot be changed when the connection to the database is open.

- MT_ON: Allows shared memory transport for local connection. The database's confifguration file MEMORYTRANS parameter must be set to 1 (the default is 0) or MT_ON will have no effect.

- MT_NETWORKTRANS_BUFSZ: Sets the size of a network connection buffer. The values are expressed in kilobytes. Allowed values are 32, 64, 128, and 256. The default value is 64.
- MT_MEMORYTRANS_BUFSZ: Sets the size of a memory transport connection buffer. The values are expressed in kilobytes. Allowed values are 32, 64, 128, and 256. The default value is 64

See also *CurrentDate* (p. 53)
SetListElements (p. 149)

GetDimension

Syntax

```
MtSTS MtCtxGetDimension
  (MtContext ctx, MtOid object, MtString attributeName,
   MtSize rank, MtSize* dimension)
MtSTS MtCtx_GetDimension
  (MtContext ctx, MtOid object, MtOid attribute,
   MtSize rank, MtSize* dimension)
```

Purpose

These functions are used to get the dimension of each rank of an array or the size of a list. They return the size of the array for a specific dimension (*rank*, starting at 0) or the length of the list (*rank* must be set to 0).

Arguments

object INPUT

An object.

*attributeName*INPUT

An attribute name.

*attribute*INPUT

An attribute object.

rank INPUT

A dimension.

*dimension*OUTPUT

When the attribute is an array, *dimension* contains the size of the array for the dimension *rank*. If the attribute is a list, *rank* must be equal to 0, and *dimension* gives the number of elements in the list.

Result

MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INCOMPOP
MATISSE_INVALRANKINDEX
MATISSE_INVALSTRINGSIZE

```
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The names of the attributes are not case sensitive. These functions can be called either from within a transaction or during a version access.

For a multidimensional array, the total number of dimensions is available to the user through the function `MtCtxGetValue`. In order to determine the size for any dimension, the user must call one of these functions for each dimension of the array (with an array of *n* dimensions, the user must call one of these functions *n* times using *`rank`* from `0` to *n*-`1`). An array may have up to 8 dimensions ($n \leq 8$).

See also

GetValue (p. 95)

GetIndex

Syntax

```
MtSTS MtCtxGetIndex
  (MtContext ctx, MtOid* index, MtString indexName)
```

Purpose

This function returns the identifier of the index associated with the name specified as an argument.

Arguments

`index` OUTPUT

The identifier of the index.

*`indexName`*INPUT

The name of the index.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOSUCHINDEX
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

See also

GetIndexInfo (p. 78)
OpenIndexEntriesStream (p. 126)

GetIndexInfo

Syntax

```
MtSTS MtCtxGetIndexInfo
  (MtContext ctx, MtString indexName,
   MtSize* nbOfEntries,
   MtIndexCriteriaInfo* indexInfo,
   MtSize* nbOfClasses, MtOid* classes)
MtSTS MtCtx_GetIndexInfo
  (MtContext ctx, MtOid indexOid,
   MtSize* nbOfEntries,
   MtIndexCriteriaInfo* indexInfo,
   MtSize* nbOfClasses, MtOid* classes)
MtSTS MtCtxMGetIndexInfo
  (MtContext ctx, MtString indexName,
   MtSize* nbOfEntries,
   MtIndexCriteriaInfo** indexInfo,
   MtSize* nbOfClasses, MtOid** classes)
MtSTS MtCtx_MGetIndexInfo
  (MtContext ctx, MtOid indexOid,
   MtSize* nbOfEntries,
   MtIndexCriteriaInfo** indexInfo,
   MtSize* nbOfClasses, MtOid** classes)
```

Purpose

This function returns information on the index whose name or identifier is specified as an argument.

Arguments

*indexOid*INPUT

The identifier of the index.

*indexName*INPUT

The name of the index.

*nbOfEntries*OUTPUT

The number of entries in the index.

Can be set to NULL, in which case the function does not return the number of entries.

*indexInfo*OUTPUT/INPUT

For the functions MtCtxGetIndexInfo and MtCtx_GetIndexInfo which do not allocate memory, this argument is a pointer to a structure of type MtCtxIndexCriteriaInfo allocated in the calling program. After the function is called, this structure will contain information on the index.

For the functions MtCtxMGetIndexInfo and MtCtx_MGetIndexInfo which allocate memory, this argument is a pointer to a structure allocated by Matisse. The calling program must declare a pointer to a structure of type MtCtxIndexCriteriaInfo. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the structure that contains index information.

This parameter can be set to NULL, in which case the function does not return the address of this structure.

*nbOfClasses*OUTPUT

The number of classes linked to the index.

This parameter can be set to NULL, in which case the function does not return the number of classes.

*classes*OUTPUT

A list of the classes in the index.

For the functions MtCtxGetIndexInfo and MtCtx_GetIndexInfo which do not allocate memory, this argument is a pointer to an array of type MtOid allocated in the calling program. After the function is called, this array will contain the identifiers of the different classes linked to the index.

For the functions MtCtxMGetIndexInfo and MtCtx_MGetIndexInfo which allocate memory, this argument is a pointer to an array allocated by Matisse. The calling program must declare a pointer to an array of type MtOid. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the array that lists the classes.

This parameter can be set to NULL, in which case the function does not return the classes.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INDEXEXPECTED
MATISSE_INDEXINCREATION
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHINDEX
MATISSE_NOTENOUGHSPACE
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

See also *GetIndex* (p. 77)
OpenIndexEntriesStream (p. 126)

GetInstancesNumber

Syntax

```
MtSTS MtCtxGetInstancesNumber
  (MtContext ctx, MtSize* instancesNumber,
   MtString className)
MtSTS MtCtx_GetInstancesNumber
  (MtContext ctx, MtSize* instancesNumber, MtOid class)
```

Purpose These functions return the number of instances of the class specified as an argument. Support for inheritance is considered: *instancesNumber* corresponds to all the instances specific to the class and to its subclasses.

Arguments

*instancesNumber*OUTPUT

The number of instances of the class that is specified as an argument. Inheritance is considered.

*className*INPUT

A class name.

class INPUT

A class object.

Result

```
MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOSUCHCLASS
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The name of the classes returned is not case sensitive. These functions can be called either from within a transaction or during a version access.

GetListElements

Syntax

```
MtSTS MtCtxGetListElements
  (MtContext ctx, MtOid object, MtString attributeName,
   MtType type,
   void* bufList,
   MtSize* numElts,
   MtSize firstEltOffset)
MtSTS MtCtx_GetListElements
  (MtContext ctx, MtOid object, MtOid attribute,
   MtType type,
   void* bufList,
   MtSize* numElts,
   MtSize firstEltOffset)
```

Purpose These functions retrieve a subset of the list value of the attribute for the specified object. The subset begins at *firstEltOffset* and its size is at most *numElts* long.

Arguments

object INPUT

An object.

*attributeName*INPUT

An attribute name.

*attribute*INPUT

An attribute.

type INPUT

The expected type of the attribute.

Possible types are: `MT_BYTES, MT_AUDIO, MT_IMAGE, MT_VIDEO, MT_NUMERIC_LIST, MT_SHORT_LIST, MT_INTEGER_LIST, MT_DOUBLE_LIST, MT_FLOAT_LIST.`

*bufList*OUTPUT

This argument is the address of a variable allocated by the calling program. After these functions are called, the subset (of the list) retrieved is copied into the variable allocated in the calling program.

*numElts*INPUT/OUTPUT

In input, this parameter indicates the maximum number of elements to be read for the subset. In output it indicates the exact number of elements of the subset.

*firstEltOffset*INPUT

This parameter indicates the offset (or position) of the first element of the subset to be retrieved. The first element of the stored list has a 0 offset.

Two specific values are allowed for *firstEltOffset*:

- `MT_BEGIN_OFFSET`
- `MT_CURRENT_OFFSET`

The first value indicates the first element of the list.

The second value indicates the position of the next element immediately after the last accessed element.

Result

```
MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALLISTOFFSET
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOTENOUGHSPACE
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_TYPEMISMATCH
MATISSE_TYPENOTALLOWED
MATISSE_WAITTIME
```

Description The names of the attributes are not case sensitive. These functions can be called either from within a transaction or during a version access.

When a program calls `MtCtxGetListElements` or `MtCtx_GetListElements`, Matisse does *not* allocate any memory space. These functions copy the subset (of the list), according to *`numElts`*, into a buffer allocated by the calling program.

Matisse internally manages an offset for each list. This offset is set to *`firstEltOffset`* + *`numElts`* after every call to the `MtCtx*GetListElements` or `MtCtx*SetListElements` functions. It can be used for further access by specifying `MT_CURRENT_OFFSET` as the value for the *`firstEltOffset`* argument. There is no default offset so `MT_CURRENT_OFFSET` cannot be specified at the first call. The offset management remains coherent only within the same transaction or version access.

See also *GetValue* (p. 95)
SetListElements (p. 149)
SetValue (p. 151)

GetNumDataBytesReceived

Syntax
```
MtSTS MtCtxGetNumDataBytesReceived
  (MtContext ctx, MtSize* num)
```

Purpose This function returns the total number of bytes corresponding to the actual transfer size of the Matisse objects, that have been read from the beginning of the connection.

Arguments
`num` `OUTPUT`
The number of bytes corresponding to the transfer size of the Matisse objects that have been read from the beginning of the connection.

Result
`MATISSE_SUCCESS`
`MATISSE_NOCURRENTCONNECTION`
`MATISSE_NULLPOINTER`

GetNumDataBytesSent

Syntax
```
MtSTS MtCtxGetNumDataBytesSent
  (MtContext ctx, MtSize* num)
```

Purpose This function returns the total number of bytes corresponding to the total size of the Matisse objects transferred, written from the beginning of the connection.

Arguments

num OUTPUT

The number of bytes corresponding to the total size of the Matisse objects transferred, written from the beginning of the connection.

Result

```
MATISSE_SUCCESS
MATISSE_NOCURRENTCONNECTION
MATISSE_NULLPOINTER
```

GetObjectClass

Syntax

```
MtSTS MtCtxGetObjectClass
  (MtContext ctx, MtOid* class, MtOid object)
```

Purpose

This function returns the class of the object. The object *object* is loaded into memory if it has not already been loaded.

Arguments

class OUTPUT

The class of *object*.

object INPUT

An object.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

This function can be called either from within a transaction or during a version access.

GetObjectsFromEntryPoint

Syntax

```
MtSTS MtCtxGetObjectsFromEntryPoint
  (MtContext ctx, MtSize* numObjects, MtOid* objects,
   MtString entryPoint,
   MtString dictName,
   MtString className)
MtSTS MtCtx_GetObjectsFromEntryPoint
  (MtContext ctx, MtSize* numObjects, MtOid* objects,
   MtString entryPoint,
   MtOid dictionary,
   MtOid class)
```

```
MtSTS MtCtxMGetObjectsFromEntryPoint
  (MtContext ctx, MtSize* numObjects, MtOid** objects,
   MtString entryPoint,
   MtString dictName,
   MtString className)
MtSTS MtCtx_MGetObjectsFromEntryPoint
  (MtContext ctx, MtSize* numObjects, MtOid** objects,
   MtString entryPoint,
   MtOid dictionary,
   MtOid class)
```

Purpose These functions retrieve the objects of the specified class (if specified) and the specified attribute accessed through *entryPoint*.

Arguments

*numObjects*INPUT/OUTPUT

In input, this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of objects (i.e. `MtCtxGetObjectsFromEntryPoint` and `MtCtx_GetObjectsFromEntryPoint`.)

In output, this parameter gives the number of objects that are instances of class *class*, define attribute *attribute* and use entry point *entryPoint*.

*objects*OUTPUT/INPUT

For the functions `MtCtxGetObjectsFromEntryPoint` and `MtCtx_GetObjectsFromEntryPoint` which do not allocate memory, this argument is an array declared in the calling program. After the function is called, this array will contain the retrieved objects.

For the functions `MtCtxMGetObjectsFromEntryPoint` and `MtCtx_MGetObjectsFromEntryPoint` which allocate memory, this argument is a pointer to an array allocated by Matisse. The calling program must declare a pointer to an `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the array listing the objects that are instances of class *class*, define attribute *attribute*, and use entry point *entryPoint*.

This parameter can be set to `NULL`, in which case the functions return only the number of objects that are instances of class *class*, define attribute *attribute*, and use entry point *entryPoint*.

*entryPoint*INPUT

The name of an entry-point object.

*dictName*INPUT

An entry-point dictionary name.

*attribute*INPUT

The identifier of an entry-point dictionary.

*className*INPUT

A class name. May be set to `NULL`.

class INPUT

The identifier of a class. May be set to 0.

Result

MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_ATTEXPECTED
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASS
MATISSE_NOSUCHCLASSATT
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

Entry points and the name of schema objects are not case sensitive. These functions can be called either from within a transaction or during a version access.

CAUTION: The MtCtx_GetObjectsFromEntryPoint and MtCtx_MGetObjectsFromEntryPoint functions return the error code MATISSE_OBJECTNOTFOUND when the attribute object or the class object is not found. When a GetObjectsFromEntryPoint function is executed successfully and no object corresponding to the request has been found, the MATISSE_SUCCESS code is returned and the *objects* argument contains no objects.

The functions MtCtxGetObjectsFromEntryPoint and MtCtx_GetObjectsFromEntryPoint do not allocate an array to store the objects that are accessed. The calling program must allocate an array of type MtOid, then pass this array as its *objects* argument.

The functions MtCtxMGetObjectsFromEntryPoint and MtCtx_MGetObjectsFromEntryPoint allocate an array to store all the identifiers found. When calling these functions, a program must pass as its *objects* argument, the address of a pointer to an MtOid. In output, this argument will point to an array that contains the objects. To free the memory space allocated for the array, the program must call the standard C function: free.

The program may also set the argument *objects* to NULL, in which case the functions simply return the number of objects accessed through *entryPoint*.

See also *OpenEntryPointStream* (p. 125)
SetValue (p. 151)

GetObjectsFromIndex

Syntax

```
MtSTS MtCtxGetObjectsFromIndex
  (MtContext ctx, MtSize numObjects,
   MtOid* objects;
   void* indexEntry[],
   MtSize nbOfCriteria,
   MtString indexName,
   MtString className)
MtSTS MtCtx_GetObjectsFromIndex
  (MtContext ctx, MtSize numObjects,
   MtOid* objects;
   void* indexEntry[],
   MtSize nbOfCriteria,
   MtOid index,
   MtOid aClass)
MtSTS MtCtxMGetObjectsFromIndex
  (MtContext ctx, MtSize numObjects,
   MtOid** objects;
   void* indexEntry[],
   MtSize nbOfCriteria,
   MtString indexName,
   MtString className)
MtSTS MtCtx_MGetObjectsFromIndex
  (MtContext ctx, MtSize numObjects,
   MtOid** objects;
   void* indexEntry[],
   MtSize nbOfCriteria,
   MtOid index,
   MtOid aClass)
```

Purpose These functions retrieve the objects of the specified class (if given) and the specified attribute from the index given a set of criteria given in `indexEntry[]`.

Arguments

`numObjects` `INPUT/OUTPUT`

As input this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of objects (i.e., `MtCtxGetObjectsFromIndex` and `MtCtx_GetObjectsFromIndex`).

As output, this parameter gives the number of objects that are instances of the class defined and that meet the criteria given in `indexEntry`.

objects OUTPUT/INPUT

For the functions `MtCtxGetObjectsFromIndex` and `MtCtx_GetObjectsFromIndex` which do not allocate memory, this argument is an array declared in the calling program. After the function is called this array will contain the retrieved objects.

For functions `MtCtxMGetObjectsFromIndex` and `MtCtx_MGetObjectsFromIndex` which allocate memory, this argument is a pointer to an array allocated by Matisse. The calling program must declare a pointer to an `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the array listing the objects that are instances of the class, and meet the criteria given in `indexEntry[]`.

This parameter can be set to NULL, in which case the functions return only the number of objects that are instances of the defined class and attribute and that meet the given criteria.

indexEntry INPUT

The given criteria lookup values of the index request.

nbOfCriteria INPUT

The number of criteria to be considered during the index object lookup. Matisse supports a maximum of four lookup criteria for any indexed object.

indexName INPUT

Name of an index.

index INPUT

Identifier of an index

className INPUT

A class name. May be set to NULL.

AClass

The identifier of a class. May be set to NULL.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSINDEX
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
MATISSE_NOSUCHINDEX
MATISSE_NOSCANNABLEINDEX
MATISSE_INDEXEXPECTED
```

Description The class argument is optional. You can specify a class if you want to put an additional constraint on the index stream. For example, if the index groups together instances of two or more classes, you can specify that instances of only one class be returned by the function. Alternatively, you can set the argument to NULL. Whether or not you specify a class, the instances that are returned are those whose attributes possess values that were specified by the criteria given by the `indexEntry` argument.

The argument `nbOfCriteria` designates the number of criteria taken into account when instances from an object are returned. This argument designates how many elements of the array indexEntry are taken into account.

CAUTION: When a `GetObjectsFromIndex` function is executed successfully an no object corresponding to the request has been found, the `MATISSE_SUCCESS` code is returned, and the object's argument contains no objects.

The functions `MtCtxGetObjectsFromIndex` and `MtCtx_GetObjectsFromIndex` do not allocate an array to store the objects that are accessed. The calling program must allocate an array of type `MtOid`, then pass this array as its objects argument.

The functions `MtCtxMGetObjectsFromIndex` and `MtCtx_MgetObjectsFromIndex` allocate an array to store all the identifiers found. When calling these functions, a program must pass as its objects argument, the address of a pointer to an `MtOid`. In output, this argument will point to an array that contains the objects. To free the memory space allocated for the array, the program must call the standard C function: free.

The program may also set the argument objects to NULL, in which case the functions simply return the number of objects.

See also *OpenIndexEntriesStream* (p. 126)
OpenIndexObjectsStream (p. 129)
SetValue (p. 151)

GetPredecessors

Syntax

```
MtSTS MtCtxGetPredecessors
  (MtContext ctx, MtSize* numPredecessors,
   MtOid* predecessors,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_GetPredecessors
  (MtContext ctx, MtSize* numPredecessors,
   MtOid* predecessors,
```

```
    MtOid object,
    MtOid relationship)
MtSTS MtCtxMGetPredecessors
   (MtContext ctx, MtSize* numPredecessors,
    MtOid** predecessors,
    MtOid object,
    MtString relationshipName)
MtSTS MtCtx_MGetPredecessors
   (MtContext ctx, MtSize* numPredecessors,
    MtOid** predecessors,
    MtOid object,
    MtOid relationship)
```

Purpose These functions return an array that contains the object predecessors through *relationship* or *relationshipName*.

Arguments

*numPredecessors*INPUT/OUTPUT

In input, this parameter specifies the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array of objects (i.e. MtCtxGetPredecessors and MtCtx_GetPredecessors).

In output, this parameter gives the number of object identifiers of predecessors returned by the function.

*predecessors*OUTPUT/INPUT

For the functions MtCtxGetPredecessors and MtCtx_GetPredecessors which do not allocate memory, this argument is the address of an array allocated in the calling program. After the function is called, this array will contain the predecessors of *object* through *relationship* or *relationshipName*.

For the functions MtCtxMGetPredecessors and MtCtx_MGetPredecessors which allocate memory, this argument is a pointer to an array allocated by Matisse. The calling program must declare a pointer to an MtOid. After declaring this pointer, the program must pass the address of this pointer as the argument to these functions. In output, this pointer contains the address of the array that lists the predecessors of *object* through *relationship* or *relationshipName*.

This parameter can be set to NULL, in which case the function returns the number of predecessors of *object* through *relationship* or *relationshipName*.

object INPUT

An object.

*relationshipName*INPUT

A relationship name.

*relationship*INPUT

A relationship object.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALIREL
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of the relationships is not case sensitive. These functions can be called either from within a transaction or during a version access.

The argument *numPredecessors* specifies the number of objects in the array.

The functions `MtCtxGetPredecessors` and `MtCtx_GetPredecessors` do not allocate an array to store the predecessors to *object* through a relationship *relationship* or *relationshipName*. The calling program must allocate an array of type `MtOid` and then pass this array as its *predecessors* argument.

The functions `MtCtxMGetPredecessors` and `MtCtx_MGetPredecessors` allocate an array to store all the identifiers found. When calling these functions, a program must pass as its *predecessors* argument the address of a pointer to `MtOid`. In output, this argument will point to an array that contains the objects. To free the memory space allocated for the array, the program can call the standard C function: `free`.

See also *OpenPredecessorsStream* (p. 137)

GetRelationship

Syntax

```
MtSTS MtCtxGetRelationship
  (MtContext ctx, MtOid* relationship,
   MtString relationshipName)
```

Purpose

This function returns the relationship whose name is *relationshipName*.

Arguments

*relationship*OUTPUT

The relationship whose name is *relationshipName*.

*relationshipName*INPUT

A relationship name.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_MULTIPLYDEFINED
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The name of the relationship is not case sensitive. This function can be called either from within a transaction or during a version access.

GetRemovedSuccessors

Syntax

```
MtSTS MtCtxGetRemovedSuccessors
  (MtContext ctx, MtSize* numRemSuccessors,
   MtOid* allRemSuccessors,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_GetRemovedSuccessors
  (MtContext ctx, MtSize* numRemSuccessors,
   MtOid* allRemSuccessors,
   MtOid object,
   MtOid relationship)
MtSTS MtCtxMGetRemovedSuccessors
  (MtContext ctx, MtSize* numRemSuccessors,
   MtOid** allRemSuccessors,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_MGetRemovedSuccessors
  (MtContext ctx, MtSize* numRemSuccessors,
   MtOid** allRemSuccessors,
   MtOid object,
   MtOid relationship)
```

Purpose These functions act through a relationship to retrieve the successors of an object that have been removed during the current transaction.

Arguments

*numRemSuccessors*INPUT/OUTPUT

In input, this parameter contains the size of the array specified by the user. This parameter can be used as an input argument only by those functions that do not allocate memory for the array (i.e. `MtCtxGetRemovedSuccessors` and `MtCtx_GetRemovedSuccessors`.)

In output, gives the number of successors that have been removed during the current transaction.

*allRemSuccessors*OUTPUT/INPUT

For the functions MtCtxGetRemovedSuccessors and MtCtx_GetRemovedSuccessors which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the successors of *object* (through the relationship *RelationshipName* or *relationship*) that have been removed during the current transaction.

For the functions MtCtxMGetRemovedSuccessors and MtCtx_MGetRemovedSuccessors which allocate memory, this argument is a pointer to an array allocated by Matisse, therefore the calling program must declare a pointer to MtOid. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the address of the array that lists the successors of *object* removed during the current transaction.

This parameter can be set to NULL, in which case the function simply returns the number of successors removed during the current transaction.

object INPUT

An object identifier.

*relationshipName*INPUT

A relationship name.

*relationship*INPUT

A relationship object.

Result

MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED

Description

The name of relationship is not case sensitive. The functions can be called either from within a transaction or during a version access. If they are called during a version access, however, these functions are not useful since they provide only the list of successors removed from the relationship and always return *numRemSuccessors* set to 0.

The functions MtCtxGetRemovedSuccessors and MtCtx_GetRemovedSuccessors do not allocate an array to store the successors to an object (through a relationship) that have been removed during the current transaction. The calling program must allocate an array of type MtOid then pass this array as its *allRemSuccessors* argument.

The functions `MtCtxMGetRemovedSuccessors` and `MtCtx_MGetRemovedSuccessors` allocate an array to store all the identifiers that are found. When calling these functions, a program must pass as its *`allRemSuccessors`* argument, the address of a pointer to `MtOid`. In output, this argument will point to an array that contains the objects. To free the memory space allocated for the array, the program must call the standard C function: `free`.

See also *RemoveAllSuccessors* (p. 141)
RemoveSuccessors (p. 143)

GetSuccessors

Syntax

```
MtSTS MtCtxGetSuccessors
   (MtContext ctx, MtSize* numSuccessors,
    MtOid* successors,
    MtOid object,
    MtString relationshipName)
MtSTS MtCtx_GetSuccessors
   (MtContext ctx, MtSize* numSuccessors,
    MtOid* successors,
    MtOid object,
    MtOid relationship)
MtSTS MtCtxMGetSuccessors
   (MtContext ctx, MtSize* numSuccessors,
    MtOid** successors,
    MtOid object,
    MtString relationshipName)
MtSTS MtCtx_MGetSuccessors
   (MtContext ctx, MtSize* numSuccessors,
    MtOid** successors,
    MtOid object,
    MtOid relationship)
```

Purpose

If the relationship is a relationship defined for *`object`*, or an inverse relationship of a relationship of which *`object`* can be a successor, these functions return an array that contains all the objects that are successors through the specified relationship.

Arguments

`numSuccessors``INPUT/OUTPUT`

In input, this parameter contains the size of the array specified by the user. It must be used as an input argument only by those functions that do not allocate memory for the array of identifiers (i.e. `MtCtxGetSuccessors` and `MtCtx_GetSuccessors`).

In output, this parameter gives the number of successors of the object through *`relationship`* or *`relationshipName`*.

`successors``OUTPUT/INPUT`

For the functions `MtCtxGetSuccessors` and `MtCtx_GetSuccessors` which do not allocate memory, this argument is an array allocated in the calling program. After the function is called, this array will contain the successors of *`object`* through the relationship specified by *`relationshipName`* or *`relationship`*.

For the functions `MtCtxMGetSuccessors` and `MtCtx_MGetSuccessors` which allocate memory, this argument is a pointer to an array allocated by Matisse. The calling program must declare a pointer to an `MtOid`. After declaring this pointer, the program must pass the address of this pointer as the argument to the function. In output, this pointer contains the array that lists the successors of *`object`* through the relationship specified by *`relationshipName`* or *`relationship`*.

This parameter can be set to `NULL`, in which case the function simply returns the number of successors of *`object`* through relationship *`relationship`* or *`relationshipName`*.

`object` `INPUT`

An object.

`relationshipName``INPUT`

A relationship name.

`relationship``INPUT`

A relationship.

Result

```
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of the relationship is not case sensitive. These functions can be called either from within a transaction or during a version access.

The argument *`numSuccessors`* specifies the number of objects in the array.

The functions `MtCtxGetSuccessors` and `MtCtx_GetSuccessors` do not allocate an array to store the object successors to *`object`* through the relationship *`relationship`* or *`relationshipName`*. The calling program must allocate an array of type `MtOid`, then pass this array as its *`successors`* argument.

The functions `MtCtxMGetSuccessors` and `MtCtx_MGetSuccessors` allocate an array to store the identifiers that are found. When calling these functions, a program must pass as its *`successors`* argument the address of a pointer to an `MtOid`. In output, this argument will point to an array that contains the successors to *`object`*. To free the memory space allocated for the array, the program can call the standard C function: `free`.

See also *OpenSuccessorsStream* (p. 139)

GetUserError

Syntax

```
void*  MtCtxGetUserError ()
```

Purpose This function returns the last user error that was generated.

Result The last user error.

Description The error identifier is set by the last call of the function `MtCtxMakeUserError`.

See also *MakeUserError* (p. 111)

GetValue

Syntax

```
MtSTS MtCtxGetValue
  (MtContext ctx, MtOid object, MtString attributeName,
   MtType* type,
   void* value,
   MtSize* rank,
   MtSize* size,
   MtBoolean* defaultValueP)
MtSTS MtCtx_GetValue
  (MtContext ctx, MtOid object, MtOid attribute,
   MtType* type,
   void* value,
   MtSize* rank,
   MtSize* size,
   MtBoolean* defaultValueP)
MtSTS MtCtxMGetValue
  (MtContext ctx, MtOid object, MtString attributeName,
   MtType* type,
```

```
    void** value,
    MtSize* rank,
    MtBoolean* defaultValueP)
MtSTS MtCtx_MGetValue
  (MtContext ctx, MtOid object, MtOid attribute,
    MtType* type,
    void** value,
    MtSize* rank,
    MtBoolean* defaultValueP)
```

Purpose

These functions return the value of an attribute for the object specified as an argument. The value corresponds to the default attribute value when the attribute in the object has no value. If this is the case, *defaultValueP* is set to MT_TRUE.

Arguments

object INPUT

An object.

*attributeName*INPUT

An attribute name.

*attribute*INPUT

An attribute.

type OUTPUT

The type of the attribute. Possible types are: MT_BOOLEAN, MT_BOOLEAN_LIST, MT_CHAR, MT_DATE, MT_DATE_LIST, MT_DOUBLE, MT_DOUBLE_LIST, MT_FLOAT, MT_FLOAT_LIST, MT_INTERVAL, MT_INTERVAL_LIST, MT_NUMERIC, MT_NUMERIC_LIST, MT_NULL, MT_SHORT, MT_SHORT_LIST, MT_INTEGER, MT_INTEGER_LIST, MT_LONG, MT_LONG_LIST, MT_STRING, MT_STRING_LIST, MT_TIMESTAMP, MT_TIMESTAMP_LIST, MT_BYTE, MT_BYTES, MT_TEXT, MT_AUDIO, MT_VIDEO.

This parameter can be set to NULL, in which case the function does not return the type of the attribute.

value OUTPUT

For the functions MtCtxGetValue and MtCtx_GetValue which do not allocate memory, this argument is the address of a variable allocated in the calling program. After these functions are called, the retrieved value is copied to the variable allocated in the calling program.

When the type is not MT_NULL, Matisse creates a copy of the attribute in the address indicated by the user. When *value* is of type MT_STRING_LIST, it contains an array of pointers, followed by the corresponding strings.

For the functions MtCtxMGetValue and MtCtx_MGetValue which allocate memory, this argument is the address of a variable pointer declared in the calling program. After these functions are called, the pointer contains the address of the variable containing the value retrieved by the function.

This parameter can be set to NULL, in which case the function does not return the value of the attribute. This is useful when the user is interested in the type and the dimension of the attribute value, or the size of this property.

rank OUTPUT

The number of dimensions of the value. This parameter can be set to NULL, in which case the function does not return any information.

The number of dimensions of a value is equal to 0 for the following types: MT_BOOLEAN, MT_CHAR, MT_DATE, MT_DOUBLE, MT_FLOAT, MT_INTERVAL, MT_NULL, MT_SHORT, MT_INTEGER, MT_LONG, MT_NUMERIC, MT_STRING, MT_TIMESTAMP, and MT_BYTE. The number of dimensions is equal to 1 for the MT_*_LIST, MT_BYTES, MT_AUDIO, MT_VIDEO and MT_IMAGE types when the stored value is not NULL and equal to 0 otherwise.

size INPUT/OUTPUT

In input, for the functions MtCtxGetValue and MtCtx_GetValue only, *size* corresponds to the size in bytes of the buffer specified by the user. In output, for all the functions, *size* corresponds to the size of the buffer that contains the value that is returned.

This parameter can be set to NULL (which requires that value is also set to null). If both the *size* and value parameters are set to NULL, the function does not return the size. This can be useful if the user is interested in the type or the dimension of the attribute value.

In output, for all of the functions, *size* corresponds to the size of the value that is returned. When the stored value is NULL, the *size* is equal to 0.

*defaultValueP*OUTPUT

This parameter can be set to NULL, in which case the function does not return any information for this parameter.

defaultValueP is set to MT_TRUE when the attribute has no value in the object, i.e., when the value that is returned corresponds to the default attribute value.

defaultValueP is set to MT_FALSE when the attribute has a value in the object.

Result

MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTENOUGHSPACE
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND

```
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of the attribute is not case sensitive. These functions can be called either from within a transaction or during a version access.

If the attribute has not been assigned for the object and if the attribute has no default value, Matisse assigns the default value of the attribute `default value`, which has the type `MT_NULL`.

When a program calls `MtCtxGetValue` or `MtCtx_GetValue`, Matisse does *not* allocate any memory space. These functions copy the value into a buffer allocated by the calling program. When *`value`* is of type `MT_STRING_LIST`, the value returned by these functions is an array of pointers, followed by the corresponding strings. The program that calls `MtCtxGetValue` or `MtCtx_GetValue` must allocate a buffer large enough to store all the pointers, as well as the strings they point to, which are returned by the functions.

It is preferable to use the `MtCtxGetValue` or `MtCtx_GetValue` functions to retrieve values whose size is fixed, i.e., for values of type `MT_INTERVAL`, `MT_BOOLEAN`, `MT_CHAR`, `MT_DATE`, `MT_DOUBLE`, `MT_FLOAT`, `MT_NUMERIC`, `MT_SHORT`, `MT_INTEGER`, `MT_LONG`, `MT_TIMESTAMP`, and `MT_BYTE`. If this is the case, a program can get better memory management with the functions that do not allocate memory space to store these values than with the functions `MtCtxMGetValue` or `MtCtx_MGetValue` which do allocate memory space.

When a program calls `MtCtxMGetValue` or `MtCtx_MGetValue`, Matisse allocates sufficient space for the value. When *`value`* is of type `MT_STRING_LIST`, the functions return an array of pointers and *not* a multidimensional array of characters. A program that calls `MtCtxMGetValue` or `MtCtx_MGetValue` must declare a variable of the appropriate type and then pass the address of this variable to these functions. When the data is no longer used, you must free the space, using the `MtMFree` function.

Example 1

The following programming example shows how to use the function `MtCtx_GetValue`, which does not allocate memory space:

```
#define BUFSIZE 1000
MtOid person;
MtOid heightAtt;
MtOid ageAtt;
MtOid nameAtt;
MtType type;
MtSize rank;
MtSize size;
MtBoolean defaultValue;
MtInteger age;
MtInteger simpleHeight;
MtDouble complexHeight;
MtString name;
...
```

```
/* Update of person, heightAtt, ageAtt and
 * nameAtt
 */
...
/* We save space for later
 */
name = (MtString) malloc(BUFSIZE);

/* Access when stored type is unknown
 */
size = BUFSIZE;
/* Look for the type of the heightAtt
 * attribute (MT_INTEGER, MT_DOUBLE or MT_NULL)
 */
MtCtx_GetValue
  (person, heightAtt,
   &type, NULL, NULL, NULL, NULL);
switch (type) {
case MT_NULL:
  if (defaultValue)
    printf("prop1Oid is not specified\n");
  else
    printf ("value=nil\n");
  break;
case MT_INTEGER:
  Mt_GetValue
    (person, heightAtt,
     NULL, (void*)&simpleHeight, NULL, NULL,
     &defaultValue);
  printf ("value = %d\n", * simpleHeight;
  break;
case MT_DOUBLE:
  Mt_GetValue
    (person, heightAtt,
     NULL, (void*)&complexHeight, NULL, NULL,
     &defaultValue);
  printf ("value = %f\n", * complexHeight);
  break;
...
default:
  printf("Value of unknown type; %d\n", type);
}
/* Access when type is either MT_NULL or MT_INTEGER
 */
size = BUFSIZE;
Mt_GetValue
  (person, ageAtt,
   &type, (void*) &age, NULL, &size, 0);
if (type == MT_NULL) {
 printf("prop2Oid is not specified\n");
```

```
 exit(0);
}
/* Access when type is either MT_NULL or
 * MT_STRING
 */
size = BUFSIZE;
Mt_GetValue
  (person, nameAtt, &type,
   (void*) name, NULL, &size, 0);
if (type == MT_NULL) {
  printf("prop3Oid is not specified\n");
  exit(0);
}
printf("person %s aged %d \n", name, age);
...
```

Example 2 The following programming example shows how to use the function `Mt_MGetValue`, which does allocate memory space:

```
#include <stdlib.h>
MtOid person;
MtOid heightAtt;
MtOid ageAtt;
MtOid nameAtt;
MtInteger age;
MtString name;
MtType type;
MtSize rank;
MtBoolean defaultValue;
MtInteger simpleHeight;
MtDouble MtDouble complexHeight;
...
/* Update of person, heightAtt, ageAtt and
 * nameAtt
 */
name = (MtString) malloc(BUFSIZE);

/* Access when stored type is unknown
 */
MtCtx_MGetValue
  (person, heightAtt,
   &type, &pValue, &rank, &defaultValue);
switch (type) {
case MT_NULL:
  if (defaultValue)
    printf("heightAtt is not specified\n");
  else
    printf ("value=nil\n");
  break;
```

```
case MT_INTEGER:
  printf ("value = %d\n", (MT_INTEGER*)pValue );
  break;
case MT_DOUBLE:
  printf("value = %f\n", *(MtDouble*)pValue));
  break;
default:
  printf("Value of unknown type; %\n", type);
}
/* Memory space allocated by Matisse for
 * this value is freed
 */
MtMFree(pValue);
/* Access when stored type is known
 * Use MtCtx_MGetValue preferably, so that
 * Matisse does not allocate space only
 * for a long only
 */
MtCtx_MGetValue
   (person, ageAtt,
    &type, (void*) &age, NULL, NULL);
if (type == MT_NULL) {
 printf("ageAtt is not specified\n");
 exit(0);
}
MtCtx_MGetValue
   (person, nameAtt,
    &type, (void*) &name, NULL, NULL);
if (type == MT_NULL) {
  printf("nameAtt is not specified\n");
  exit(0);
}
printf("person %s aged %d \n", name, *age);
...
/* End of use of the values of the
 * attributes name and age. The memory space
 * allocated by Matisse for these values is
 * freed.
 */
MtMFree (age);
MtMFree (name);
```

See also *GetDimension* (p. 76)
GetListElements (p. 80)
SetListElements (p. 149)

IntervalAdd

Syntax

```
MtSTS MtIntervalAdd
  (MtInterval *result,
   MtInterval *interval1,
   MtInterval *interval2)
```

Purpose

This function adds two `MtInterval` values.

Arguments

result INPUT

`MtInterval` result value.

interval1 INPUT

An `MtInterval` value.

interval2 INPUT

An `MtInterval` value.

Result

`MATISSE_SUCCESS`
`MATISSE_NULLPOINTER`
`MATISSE_INVALID_TIMEINTERVAL.`

See also

TimestampGetCurrent (p. 173)
IntervalMultiply (p. 104)
IntervalSubtract (p. 105)

IntervalCompare

Syntax

```
MtSTS MtIntervalCompare
  (MtInteger *result,
   MtInterval *interval1,
   MtInterval *interval2)
```

Purpose

This function compares *interval1* to *interval2*.

Arguments

result OUTPUT

An integer greater than, equal to, or less than 0, if the first interval argument is repectively greater than, equal to, or less than the second one.

interval1 INPUT

An `MtInterval` value.

interval2 INPUT

An `MtInterval` value.

Result

`MATISSE_SUCCESS`
`MATISSE_NULLPOINTER`
`MATISSE_INVALID_TIMEINTERVAL.`

IntervalDivide

Syntax

```
MtSTS MtIntervalDivide
  (MtInterval *result,
   MtInterval *interval
   MtInteger nParts)
```

Purpose This function divides *interval* into `nParts` intervals.

Arguments

result INPUT

`MtInterval` value returned.

interval INPUT

An `MtInterval` value.

nParts INPUT

A signed 32-bit integer.

Result

```
MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_DIVISION_BY_ZERO
MATISSE_INVALID_TIMEINTERVAL.
```

See also *TimestampGetCurrent* (p. 173)
IntervalMultiply (p. 104)
IntervalSubtract (p. 105)

IntervalBuild

Syntax

```
MtSTS MtIntervalBuild
  (MtInterval *interval,
   MtString buffer)
```

Purpose This function creates an `MtInterval` value from its printed representation in *buffer*.

Arguments

interval INPUT

An `MtInterval` value.

buffer INPUT

A `character string` representing an interval in the following format: `[+|-]DD HH-MM-SS[:uuuuuu]`.

Result

```
MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMEINTERVAL.
```

Description The *interval* is built if *buffer* represents a valid interval.
For example:

```
MtIntervalExtract("30 25:00:33", & time);
```

will return MATISSE_INVALID_TIMEINTERVAL because 25 is not a valid value for the hours field.

See also *IntervalBuild* (p. 103)

IntervalMultiply

Syntax

```
MtTimestamp MtIntervalMultiply
   (MtInterval *result,
    MtInterval *interval
    MtInteger nParts)
```

Purpose This function multiplies *interval* by *nParts*.

Arguments

result INPUT

MtInterval value returned.

interval INPUT

An MtInterval value.

nParts INPUT

A signed 32-bit integer.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMEINTERVAL.

See also *TimestampGetCurrent* (p. 173)
IntervalDivide (p. 103)
IntervalSubtract (p. 105)

IntervalPrint

Syntax

```
MtSTS MtIntervalPrint
   (MtString buffer,
    MtSize bufferSize,
    const char *format,
    MtInterval *interval,)
```

Purpose This function outputs *interval* according to *format* into the character string pointed to by *buffer*.

Arguments

buffer OUTPUT

A character string into which the formatted interval will be stored.

bufferSize INPUT

An integer indicating the maximum number of character that can be placed into *buffer*.

format INPUT

A character string containing directives to output the different interval fields; possible directives are:

%s interval sign "-" or "+"

%D days (0 -1491308)

%H hours (00-23)

%M minutes (00-59)

%S seconds (00-59)

%U microseconds (000000..999999)

%% to print %

interval INPUT

The MtInterval structure to print.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMEINTERVAL.

See also

IntervalBuild (p. 103)

IntervalSubtract

Syntax

```
MtSTS MtIntervalSubtract
  (MtInterval *result,
   MtInterval *interval1,
   MtInterval *interval2)
```

Purpose

This function subtracts two MtInterval values.

Arguments

result INPUT

MtInterval result value.

interval1 INPUT

An MtInterval value.

interval2 INPUT

An MtInterval value.

Result

MATISSE_SUCCESS
MATISSE_INVALID_TIMEINTERVAL.

See also

TimestampGetCurrent (p. 173)
IntervalDivide (p. 103)
IntervalMultiply (p. 104)

IsInstanceOf

Syntax

```
MtSTS MtCtxIsInstanceOf
  (MtContext ctx, MtBoolean* result,
```

```
    MtOid object,
    MtString className)
MtSTS MtCtx_IsInstanceOf
   (MtContext ctx, MtBoolean* result,
    MtOid object,
    MtOid class)
```

Purpose This function determines if the object *object* is or is not an instance of the class *className* (or *class*) or an instance of one of its subclasses.

Arguments

result OUTPUT

This argument is equal to MT_TRUE if the object is an instance of the class or of one of its subclasses.

This argument is equal to MT_FALSE otherwise.

object INPUT

A Matisse object.

className INPUT

A class name.

class INPUT

A class object.

Result

MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description The names of classes are not case sensitive.

IsPredefinedObject

Syntax

```
MtSTS MtCtxIsPredefinedObject
  (MtContext ctx, MtBoolean* predefinedP, MtOid object)
```

Purpose This function indicates whether the object specified as an argument is part of the initial meta-schema.

Arguments

predefinedP OUTPUT

This argument is set to 1 when the object belongs to the initial meta-schema.

This argument is set to `0` otherwise.

object INPUT

This is the object to be tested to determine whether or not it is an element of the initial meta-schema.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_WAITTIME
```

LoadObjects

Syntax

```
MtSTS MtCtxLoadNumObjects
  (MtContext ctx, MtSize numObjects, MtOid* objects)
MtSTS MtCtxLoadObjects
  (MtContext ctx, MtSize numObject,
   MtOid firstObject, ...)
```

Purpose

These functions load the value of the objects that are specified as arguments.

Arguments

numObjects INPUT

The number of objects to load.

objects INPUT

An array of objects.

firstObject INPUT

First object to load.

Other INPUT arguments:

The argument *firstObject* is followed by the rest of the arguments to load.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_EXCEEDSLIMIT
MATISSE_INVALNB
MATISSE_MEMORYFAULT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The objects may be specified either in an array or as a variable length list. In MT_DATA_DEFINITION connection mode, when a class is loaded, its superclasses are also loaded. In MT_DATA_MODIFICATION connection mode, all the schema objects are loaded at connection time.

Calling this function ensures that no server access will read any of the objects specified as arguments.

The value of the *numObjects* argument must not exceed the value returned by the function MtCtxGetConfigurationInfo when its type argument is set to MT_MAX_BUFFERED_OBJECTS.

These functions can be called from within a transaction or during a version access.

LockObjects

Syntax

```
MtSTS MtCtxLockNumObjects
   (MtContext ctx, MtSize numObjects,
    MtOid* objects,
    MtLock* locks)
MtSTS MtCtxLockObjects
   (MtContext ctx, MtSize numObjects,
    MtOid firstObject,
    MtLock firstLock,
    ...)
```

Purpose

These functions lock objects.

Arguments

numObjects INPUT

The number of objects to be locked.

objects INPUT

An array that contains the objects to be locked. The database programmer is responsible for the memory space associated with the array.

locks INPUT

An array that contains the locks with which the objects in *objects* must be locked. The value of a lock can be either MT_READ or MT_WRITE. The database programmer is responsible for the memory space associated with the array.

firstObject INPUT

The first object to be locked.

firstLock INPUT

The lock associated with the first object to be locked.

Other INPUT arguments:

The identifiers of all other objects to be locked are entered after *firstObject*.

The argument *firstLock* is followed by the lock (MT_READ or MT_WRITE) associated with the objects.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCK
MATISSE_EXCEEDSLIMIT
MATISSE_FROZENOBJECT
MATISSE_INVALLOCK
MATISSE_INVALNB
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANS
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

Locks are granted atomically: either all locks or no locks are granted.

Note that you can lock only a limited number of objects in a single transaction. This limit is the value returned by the function MtCtxGetConfigurationInfo when the *type* argument is set to MT_MAX_BUFFERED_OBJECTS. If you try to lock more than this number of objects, the error code MATISSE_EXCEEDSLIMIT is returned.

Note that when the error MATISSE_DEADLOCK occurs, the transaction is not aborted, however, no locks have been granted and the request must be performed again.

These functions can be called only from within a transaction.

Example

```
MtSTS status;
MtLock locks[3];
MtOid objects[3];
MtOid obj1;
MtOid obj2;
MtOid obj3;
objects[0] = obj1;
locks[0] = MT_READ;
objects[1] = obj2;
locks[1] = MT_WRITE;
objects[2] = obj3;
locks[2] = MT_READ;
Status = MtCtxLockNumObjects (3, objects, locks);
CheckStatus (status)
...
status = MtCtxLockObjects(3,
                          obj1, MT_READ,
                          obj2, MT_WRITE,
                          obj3, MT_READ);
CheckStatus (status)
```

LockObjectsFromEntryPoint

Syntax

```
MtSTS MtCtxLockObjectsFromEntryPoint
  (MtContext ctx, MtLock lock,
   MtString entryPoint,
   MtString dictName,
   MtString className)
MtSTS MtCtx_LockObjectsFromEntryPoint
  (MtContext ctx, MtLock lock,
   MtString entryPoint,
   MtOid dictionary,
   MtOid class)
```

Purpose

These functions set locks of type *lock* (MT_READ, MT_WRITE) on objects whose entry point is given as an argument.

Arguments

lock INPUT

The type of lock to be set on the objects accessed through *entryPoint*. Its value can be either MT_READ or MT_WRITE.

entryPoint INPUT

The name of an entry point.

dictName INPUT

The name of an entry-point dictionary.

dictionary INPUT

An object (an entry-point dictionary).

className INPUT

A class name. Can be set to NULL.

class INPUT

An object (a class). Can be set to 0.

Result

MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCK
MATISSE_FROZENOBJECT
MATISSE_INVALLOCK
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASS
MATISSE_NOSUCHCLASSATT
MATISSE_NOTRANS
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description Entry points and the name of schema objects are not case sensitive.

If one of these functions fails because of a deadlock or the wait-time expiration, some locks may have already been granted. The request must be performed again.

Note that when the error `MATISSE_DEADLOCK` occurs, the transaction is not aborted, however, no locks have been granted and the request must be performed again.

These functions can be called only from within a transaction.

MakeUserError

Syntax
```
MtSTS MtCtxMakeUserError
  (MtContext ctx, void* error, MtString errorString)
```

Purpose This function allows you to generate a unique user error.

Arguments
error INPUT
The user error. The identifier can be any data allowing the user to specifically identify the error generated by this function.
errorString INPUT
The string to be attached to the user error.

Result `MATISSE_USERERROR`

Description The error identifier is *error*; its code is `MATISSE_USERERROR`; its string (the error explanation) is *errorString*.

See also *GetUserError* (p. 95)
Failure (p. 56)

NextIndexEntry

Syntax
```
MtSTS MtCtxNextIndexEntry
  (MtContext ctx, MtStream stream,
   void* values[],
   MtOid* object)
```

Purpose This function returns information on the next entry in the index stream.

Arguments
stream INPUT
An index stream previously opened using either the `MtCtxOpenIndexEntriesStream` or `MtCtx_OpenIndexEntriesStream` function.

values OUTPUT

The values of the criteria at the index entry.

object OUTPUT

The object indexed by the criteria values.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_ENDOFSTREAM
MATISSE_INVALSTREAM
MATISSE_INVALMAPFUNCTION
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

An index entry is composed of the following information:

criteria values,

the object indexed by the criteria values.

The object identifier returned by *object* is that of the current object in the index stream. The values returned by *values* are those for the index criteria at the current index entry. In other words, *values* contains the values of those object attributes that form the criteria of the index.

Note that you must allocate space for the variable *values* before calling *MtCtxNextIndexEntry*. *values* is an array of pointers, and each pointer points to the memory allocated for each criterion value.

Example

For example, suppose you have an index with the criteria `name` (string of 20 characters) and `age` (`MtInteger`). The declarations for these variables, as well as the declaration of the array that contains pointers to these variables, are as follows:

```
MtOid object;
void *values [2];
MtChar name [20];
MtInteger age;
values [0] = name;
values [1] = &age;
status = MtCtxNextIndexEntry
           (stream, values, &object);
```

See also

NextObject (p. 113)
NextObjects (p. 114)
OpenIndexEntriesStream (p. 126)

NextObject

Syntax

```
MtSTS MtCtxNextObject
  (MtContext ctx, MtStream stream, MtOid* object)
```

Purpose

This function returns the next object in the stream.

Arguments

stream INPUT

A class stream, an entry-point stream, a relationship stream, or an inverse relationship stream.

object OUTPUT

The subsequent object in the stream, or NULL if there is no subsequent element.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_ENDOFSTREAM
MATISSE_INVALSTREAM
MATISSE_INVALMAPFUNCTION
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_STREAMCLOSED
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

Depending on the stream type, the identifier can be a class instance (see MtCtxOpenInstancesStream), an object indexed by an entry point (see MtCtxOpenEntryPointStream) or by an index (see MtCtxOpenIndexEntriesStream), or by the object's successor (see MtCtxOpenSuccessorsStream) or predecessor (see MtCtxOpenIRelStream). Once all the objects have been accessed, the function returns MATISSE_ENDOFSTREAM, and *object* is set to 0.

This function can be called from within a transaction or during a version access.

See also

NextIndexEntry (p. 111)
NextObjects (p. 114)
OpenInstancesStream (p. 132)
OpenEntryPointStream (p. 125)
OpenIndexEntriesStream (p. 126)
OpenPredecessorsStream (p. 137)
OpenSuccessorsStream (p. 139)

NextObjects

Syntax

```
MtSTS MtCtxNextObjects
  (MtContext ctx, MtStream stream, MtOid* objects,
   MtSize* numObjects)
```

Purpose

This function returns the next objects in the stream.

Arguments

stream INPUT

A class stream, an entry-point stream, a relationship stream, or an inverse relationship stream.

object OUTPUT

The subsequent objects in the stream.

numObjects INPUT/OUTPUT

The number of objects required on input, the number of objects obtained on output.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_ENDOFSTREAM
MATISSE_INVALSTREAM
MATISSE_INVALMAPFUNCTION
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_STREAMCLOSED
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

Depending on the stream type, the identifier is a class instance (see MtCtxOpenInstancesStream), an object indexed by an entry point (see MtCtxOpenEntryPointStream) or by an index (see MtCtxOpenIndexEntriesStream), or the object's successor (see MtCtxOpenSuccessorsStream) or predecessor (see MtCtxOpenPredecessorsStream). Once all the objects have been accessed, the function returns MATISSE_ENDOFSTREAM.

This function can be called from within a transaction or during a version access.

See also

NextIndexEntry (p. 111)
NextObject (p. 113)
OpenInstancesStream (p. 132)
OpenOwnInstancesStream (p. 136)
OpenEntryPointStream (p. 125)
OpenIndexEntriesStream (p. 126)

OpenIndexObjectsStream (p. 129)
OpenPredecessorsStream (p. 137)
OpenSuccessorsStream (p. 139)

NextProperty

Syntax

```
MtSTS MtCtxNextProperty
   (MtContext ctx, MtStream objectStream,
    MtOid* property,
    MtBoolean* specifiedP)
```

Purpose

This function gives the subsequent property in the stream.

Arguments

objectStream INPUT

An object properties stream (This can be an object attribute, object relationship or object inverse relationship stream).

property OUTPUT

The attribute, relationship, or 0 if there is no subsequent property.

specifiedP OUTPUT

Is set to MT_TRUE when the property has a value in the *object*; otherwise, set to MT_FALSE.

Result

MATISSE_SUCCESS
MATISSE_DEADLOCKABORT
MATISSE_ENDOFSTREAM
MATISSE_INVALSTREAM
MATISSE_INVALMAPFUNCTION
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER

Description

The stream includes the identifiers of either all the attributes, all the relationships defined for the object, or all the inverse relationships present in the object, depending on the stream type (A stream mapping the attributes, the relationships, or the inverse relationships).

When the property has a value in the object, the *specifiedP* argument is set to MT_TRUE; otherwise, *specifiedP* is set to MT_FALSE.

If the stream has been opened with the MtCtxOpenInverseRelationshipsStream function, the *specifiedP* argument is always set to MT_TRUE since the provided properties are those present in the object.

Once all the properties (attributes, relationships, or inverse relationships) have been returned, the function returns the MATISSE-ENDOFSTREAM status and *property* is set to 0.

This function can be called from within a transaction or during a version access.

See also
OpenAttributesStream (p. 124)
OpenInverseRelationshipsStream (p. 134)
OpenRelationshipsStream (p. 138)

NextVersion

Syntax

```
MtSTS MtCtxNextVersion
  (MtContext ctx, MtStream versionStream,
   MtString buf,
   MtSize bufSize)
```

Purpose
This function provides a string associated with the next version in the stream.

Arguments

versionStream INPUT
The stream containing the enumeration of the saved versions that exist in the database.

buf OUTPUT
The buffer used to insert the name of the next version mode.

bufSize INPUT
The size of the buffer.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_ENDOFSTREAM
MATISSE_NOCURRENTCONNECTION
MATISSE_NULLPOINTER
MATISSE_NOTENOUGHSPACE
```

See also
CommitTransaction (p. 48)
OpenVersionStream (p. 140)

NumericAdd

Syntax

```
MtSTS MtNumericAdd
  (MtNumeric *result,
   MtNumeric *value1,
   MtNumeric *value2)
```

Purpose
Adds two numeric values.

Arguments

result OUTPUT
A numeric value into which the result of the addition is stored.

value1 INPUT

A numeric value.

value2 INPUT

A numeric value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_NUMERICOVERFLOW

NumericBuild

Syntax

```
MtSTS MtNumericBuild
  (MtNumeric *result,
   MtString buffer,
   MtSize precision,
   MtSize scale)
```

Purpose

This function creates a numeric value given a character string and a desired precision and scale.

Arguments

string INPUT

The string containing the numeric value to be stored in the numeric structure

precision INPUT

The desired precision of the numeric to be stored. A maximum precision of 19 is supported.

scale INPUT

The desired scale of the numeric to be stored.

numeric OUTPUT

A pointer to the numeric structure into which the value will be stored.

Results

MATISSE_SUCCESS
MATISSE_NUMERICOVERFLOW
MATISSE_INVALNUMFORMAT

NumericCompare

Syntax

```
MtSTS MtNumericCompare
  (MtInteger *result,
   MtNumeric *value1,
   MtNumeric *value2)
```

Purpose

This function compares value1 to value2.

Arguments

result OUTPUT

A positive integer if `value1` is greater than `value2`, 0 if `value1` equals `value2`, or a negative integer if `value1` is less than `value2`.

value1 INPUT

A numeric value.

value2 INPUT

A numeric value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER

NumericDivide

Syntax

```
MtSTS MtNumericDivide
  (MtNumeric *result,
   MtNumeric *value1,
   MtNumeric *value2)
```

Purpose

Divides value1 by value2.

Arguments

result OUTPUT

A numeric value into which the result of the division is stored.

value1 INPUT

A numeric value.

value2 INPUT

A numeric value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_DIVISION_BY_ZERO
MATISSE_NUMERICOVERFLOW

NumericFromDouble

Syntax

```
MtSTS MtNumericFromDouble
  (MtNumeric *result,
   MtDouble *value)
```

Purpose

To convert an `MtDouble` value into a `numeric` value.

Arguments

result OUTPUT

A numeric value

value INPUT

An MtDouble value to convert.

Results
```
MATISSE_SUCCESS
MATISSE_NUMERICOVERFLOW
```

NumericFromLong

Syntax
```
MtSTS MtNumericFromLong
  (MtNumeric *result,
   MtLong *value)
```

Purpose To convert an `MtLong` value into a `numeric` value.

Arguments
`result` OUTPUT
A numeric value
`value` INPUT
An MtLong value to convert.

Results
`MATISSE_SUCCESS`
`MATISSE_NUMERICOVERFLOW`.

NumericGetPrecision

Syntax
```
MtNumericGetPrecision
  (MtSize *result,
   MtString value)
```

Purpose Get the precision of a `numeric` value represented as a character string.

Arguments
`result` OUTPUT
Number of digits of precision necessary to store the numeric value.
`value` INPUT
A character string containing the `numeric` value.

Result
`MATISSE_SUCCESS`
`MATISSE_INVALFORMAT`.

NumericGetScale

Syntax
```
MtNumericGetScale
  (MtSize *result,
   MtString value)
```

Purpose Get the number of digits after the decimal point of a `numeric` value represented as a character string.

Arguments

result OUTPUT

Number of scale digits necessary to store the numeric value.

value INPUT

A character string containing the numeric value.

Result

MATISSE_SUCCESS
MATISSE_INVALFORMAT.

NumericMultiply

Syntax

```
MtNumericMultiply
  (MtNumeric *result,
   MtNumeric *value1,
   MtNumeric *value2)
```

Purpose

Multiplies *value1* by *value2*.

Arguments

result OUTPUT

A numeric value into which the result is stored.

value1 INPUT

A numeric value.

value2 INPUT

A numeric value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_NUMERICOVERFLOW

NumericPrint

Syntax

```
MtSTS MtNumericPrint
  (MtString buffer,
   MtSize buffsz,
   MtNumeric *value)
```

Purpose

Creates a character string representation of *value* into buffer.

Arguments

buffer OUTPUT

buffer where the numeric value to be printed is stored.

buffsz INPUT

Size of the buffer passed to the function.

numeric INPUT

A numeric value.

Result
MATISSE_SUCCESS
MATISSE_ARRAYTOOSMALL
MATISSE_INVALNUMFORMAT

NumericToDouble

Syntax
```
MtSTS MtNumericToDouble
  (MtDouble *result,
   MtNumeric *value)
```

Purpose To convert an MtNumeric value into an MtDouble value.

Arguments
result OUTPUT
An MtDouble value
value INPUT
A numeric value to convert.

Results
MATISSE_SUCCESS
MATISSE_NUMERICOVERFLOW

NumericToLong

Syntax
```
MtSTS MtNumericToLong
  (MtLong *result,
   MtNumeric *value)
```

Purpose To convert an MtNumeric value into an MtLong value.

Arguments
result OUTPUT
An MtLong value
value INPUT
A numeric value to convert.

Results
MATISSE_SUCCESS
MATISSE_NUMERICOVERFLOW

NumericRound

Syntax
```
MtSTS MtNumericRound
  (MtNumeric *result,
   MtNumeric *value,
   MtSize scale,
   MtRounding roundingMethod)
```

Purpose To round a `numeric` value to the specified scale, using the rounding method specified by *`roundingMethod`*.

Arguments

`result` `OUTPUT`

The numeric value the newly rounded value is to be stored in.

`value` `INPUT`

The `numeric` value to be rounded.

`roundingMethod` `INPUT`

The rounding method to be used.

Results

`MATISSE_SUCCESS`

`MATISSE_INVALNUMFORMAT`

`MATISSE_NUMERICOVERFLOW`

Description The parameter *`scale`* contains the number of digits to the right of the decimal point to which to round the passed `numeric` value. If the value of *`scale`* is zero, all the digits to the right of the decimal point will be truncated. If the value of *`scale`* is negative, the function will act as if it was passed a zero. A value of *`scale`* greater than the current scale will leave the result unchanged. The scale of the new rounded value will be the same as that of the *`scale`* parameter. The precision however will be unchanged.

The parameter *`roundingMethod`* designates which type of rounding method is to be used. It can take the following values:

`MT_ROUND_HALF_UP`: If the digit to the right of the digit to be rounded to is greater than or equal to five, the rounded digit will be incremented by one. If the digit to the right of the rounded digit is less than five, the digits to the right of the rounded digit will simply be discarded.

`MT_ROUND_DEFAULT`: Same as `MT_ROUND_HALF_UP`.

`MT_ROUND_CEILING`: If the value of the numeric to be rounded is positive, increment the digit to be rounded by one. Otherwise simply discard the digits to the right of the digit to be rounded to.

`MT_ROUND_HALF_EVEN`: Also known as *Banker Rounding*. If the digit to the right of the digit to be rounded to is greater than five, increment to digit to be rounded to by one. If the digit to the right of the digit to be rounded to is less than five, simply discard the digits to the right of the rounding digit. If the value of the digit to the right of the digit to be rounded to is equal to five, if the rounding digit is an odd number, increment it so it is an even number. If the rounding digit is even, simply discard the digits to the right.

`MT_ROUND_DOWN`: Truncates the digits to the right of the digit to be rounded to.

MT_ROUND_FLOOR: If the value of the numeric to be rounded is negative, increment to digit to be rounded to by one. Otherwise simply discard the digits to the right of the digit to be rounded to.

NumericSubtract

Syntax

```
MtNumericSubtract
   (MtNumeric *result,
    MtNumeric *value1,
    MtNumeric *value2)
```

Purpose Subtracts *value2* from *value1*.

Arguments

result OUTPUT

A numeric value into which the result is stored.

value1 INPUT

A numeric value.

value2 INPUT

A numeric value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_NUMERICOVERFLOW

ObjectSize

Syntax

```
MtSTS MtCtxObjectSize
   (MtContext ctx, MtSize* size,MtOid object)
```

Purpose This function returns the size (in bytes) of the object when it is written to disk.

Argument

size OUTPUT

The size, expressed in bytes, of the object on the server.

object INPUT

An object.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description The size returned should help you estimate the cost (in bytes) of sending the object across a network.

This function can be called either from within a transaction or during a version access.

OidEQ

Syntax
```
int MtOidEQ
  (MtOid object1, MtOid object2)
```

Purpose Each Matisse object has a unique identifier (of type `MtOid`) that provides a means to denote or refer to the object. This function indicates if the two Oids refer to the same object.

Arguments

object1 INPUT
The identifier of a Matisse object.

*object2*INPUT
The identifier of a Matisse object.

Result `1` if the two Oids refer to the same object; `0` otherwise.

OpenAttributesStream

Syntax
```
MtSTS MtCtxOpenAttributesStream
  (MtContext ctx, MtStream* attStream,
   MtOid object)
```

Purpose This function opens the object attribute stream *objectAttStream* on the specified object. The function `MtCtxNextProperty` will use the stream to provide the user with the attributes of *object*.

Arguments

attStream OUTPUT
The object attribute stream.

object INPUT
An object.

Result
```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEALOCKABORT
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
```

```
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description This function can be called from within a transaction or during a version access.

See also *CloseStream* (p. 47)
GetAllAttributes (p. 60)
NextProperty (p. 115)

OpenEntryPointStream

Syntax

```
MtSTS MtCtxOpenEntryPointStream
  (MtContext ctx, MtStream* entryPointStream,
   MtString entryPoint,
   MtString dictName,
   MtString className,
   MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenEntryPointStream
  (MtContext ctx, MtStream* entryPointStream,
   MtString entryPoint,
   MtOid dictionary,
   MtOid class,
   MtSize nbObjectsPerCall)
```

Purpose These functions initialize the entry point stream: *entryPointStream*, based on the arguments specified. The function `MtCtxNextObject` uses this stream to provide the user with the objects accessed by the entry point *entryPoint*.

Arguments

entryPointStream`OUTPUT`
: The entry point stream.

entryPoint`INPUT`
: An entry-point value.

attributeName`INPUT`
: An entry-point dictionary name.

dictionary`INPUT`
: An entry-point dictionary.

className`INPUT`
: A class name. May be set to `NULL`.

class `INPUT`
: A class object. May be set to `0`.

nbObjectsPerCall`INPUT`
: This argument allows you to specify the maximum number of objects that will be retrieved at each server call. You may use the `MT_MAX_PREFETCHING` keyword to prefetch the maximum number of objects that can be handled in a request to the server.

Result

```
MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASS
MATISSE_NOSUCHCLASSATT
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The name of classes are not case sensitive. These functions can be called either from within a transaction or during a version access.

Adjusting the value of the *nbObjectsPerCall* argument allows you to tune the maximum response time for further calls to the `MtCtxNextObject` function. The greater the value of *nbObjectsPerCall*, the shorter is the overall enumeration.

See also

CloseStream (p. 47)
GetObjectsFromEntryPoint (p. 83)
NextObject (p. 113)
NextObjects (p. 114)
SetValue (p. 151)

OpenIndexEntriesStream

Syntax

```
MtSTS MtCtxOpenIndexEntriesStream
  (MtContext ctx, MtStream* indexStream,
   MtString indexName,
   MtString className,
   MtDirection direction,
   MtSize nbOfCriteria,
   void* startValues[],
   void* endValues[],
   MtSize nbEntriesPerCall)
MtSTS MtCtx_OpenIndexEntriesStream
  (MtContext ctx, MtStream* indexStream,
   MtOid index,
   MtOid class,
   MtDirection direction,
   MtSize nbOfCriteria,
```

```
    void* startValues[],
    void* endValues[],
    MtSize nbEntriesPerCall)
```

Purpose These functions initialize the index stream *indexStream* based on the arguments specified. This stream enables you to assemble all the objects that are within the bounds set by the arguments: *startValues* and *endValues*.

Arguments

*indexStream*OUTPUT

The stream of the index.

*indexName*INPUT

An index name.

index INPUT

An index identifier.

*className*INPUT

A class name. Can be set to NULL.

class INPUT

A class identifier. Can be set to 0.

*direction*INPUT

The scan direction of the index stream. The direction can be from start to end or from end to start.

*nbofCriteria*INPUT

The number of criteria to be considered in the start and end values.

*startValues*INPUT

Start values of the index request.

*endValues*INPUT

End values of the index request.

*nbEntriesPerCall*INPUT

This argument allows you to specify the maximum number of entries that will be retrieved at each server call. You may use the MT_MAX_PREFECTCHING keyword to prefetch the maximum number of entries that can be handled in a request to the server.

Result

```
MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INDEXEXPECTED
MATISSE_INVALCRITERIANB
MATISSE_INVALDIRECTION
MATISSE_INVALINTERVAL
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSCANNABLEINDEX
MATISSE_NOSUCHCLASS
MATISSE_NOSUCHCLASSINDEX
MATISSE_NOSUCHINDEX
```

```
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The argument *class* is optional. You can specify a class if you want to put an additional constraint on the index stream. For example, if the index groups together instances of two or more classes, you can specify that instances of only one class be associated with the stream. Alternatively, you can set the argument to NULL.

Whether or not you specify a class through the argument *class*, the instances that are grouped together in the stream are those whose attributes possess values within the intervals specified by the arguments *startValues* and *endValues*.

The argument *nbOfCriteria* designates the number of criteria taken into account when an index stream is opened. In other words, this argument designates how many elements of the arrays *startValues* and *endValues* to take into account. You can specify NULL for its value. If you set *nbOfCriteria* to 0, the values set for the arguments *startValues* and *endValues* are ignored.

The arguments *startValues* and *endValues* are arrays of pointers. It is possible to leave an interval undefined for one or more criteria. To do this, set the pointer to NULL for the criterion whose interval you want to leave undefined in *startValues* or *endValues*.

The limits that you set with *startValues* and *endValues* must adhere to the following constraint:

```
startValues ≤ endValues
```

To illustrate this concept, suppose you have an index with the two criteria: LastName and FirstName. Both of these criteria are built in ascending order. Suppose that you are searching for all instances indexed, which have a Name value equal to or greater than Flanagan and less than or equal to Petrocelli. In addition, all the instances must have a value for FirstName that is greater than or equal to Mike and less than or equal to Rico.

The values for these criteria are valid because the constraint *startValues* ≤ *endValues* is met. Flanagan ≤ Petrocelli and Mike ≤ Rico, as shown in the following table:

Arguments	Last Name	First Name
startValues	Flanagan	Mike
endValues	Petrocelli	Rico

If *startvalues* were (Petrocelli, Rico) and *endValues* were (Flanagan, Mike), then these arguments would not have correct values. Because Petrocelli and Rico are respectively greater than Flanagan and Mike, the constraint *startValues* ≤ *endValues* would not be met.

Note that the compare operator ≤ deals with the ordering of the criteria. If the name criterion had been created in descending order, then the constraint described in the previous paragraph would be the reverse.

The argument *direction* lets you specify a direction for the stream. You can specify a stream that ascends from the instance with the lowest value to the highest, or you can specify the a stream that descends from the instance with the highest value to the lowest.

When a stream is opened on an index, the index in question is considered frozen. No subsequent modifications made on the index will be visible during the scan. Modifications will be visible when the next stream is opened on the index.

These functions can be called from within a transaction or during a version access.

Adjusting the value of the *nbEntriesPerCall* argument allows you to tune the maximum response time for further calls to `MtCtxNextIndexEntry` function. The greater is the value, the shorter the overall enumeration.

See also
CloseStream (p. 47)
GetIndex (p. 77)
GetIndexInfo (p. 78)
NextIndexEntry (p. 111)
NextObject (p. 113)
NextObjects (p. 114)

OpenIndexObjectsStream

Syntax

```
MtSTS MtCtxOpenIndexObjectsStream
  (MtContext ctx, MtStream* indexStream,
   MtString indexName,
   MtString className,
   MtDirection direction,
   MtSize nbOfCriteria,
   void* startValues[],
   void* endValues[],
   MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenIndexObjectsStream
  (MtContext ctx, MtStream* indexStream,
   MtOid index,
   MtOid class,
```

```
MtDirection direction,
MtSize nbOfCriteria,
void* startValues[],
void* endValues[],
MtSize nbObjectsPerCall)
```

Purpose

These functions initialize the index stream *indexStream* depending on the arguments specified. This stream enables you to assemble all the objects that are within the bounds set by the arguments *startValues* and *endValues*.

Arguments

*indexStream*OUTPUT

The stream of the index.

*indexName*INPUT

An index name.

index INPUT

An index identifier.

*className*INPUT

A class name. This argument can be set to NULL.

class INPUT

A class identifier. This argument can be set to 0.

*direction*INPUT

The scanning direction of the index stream. The direction can be from start to end or from end to start.

*nbofCriteria*INPUT

The number of criteria to be considered in the start and end values.

*startValues*INPUT

Start values of the index request.

*endValues*INPUT

End values of the index request.

*nbObjectsPerCall*INPUT

This argument allows you to adjust the maximum number of objects that will be retrieved for each server call. You may use the MT_MAX_PREFETCHING keyword to prefetch the maximum number of objects that can be handled in a request to the server.

Result

MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INDEXEXPECTED
MATISSE_INVALCRITERIANB
MATISSE_INVALDIRECTION
MATISSE_INVALINTERVAL
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSCANNABLEINDEX
MATISSE_NOSUCHCLASS

```
MATISSE_NOSUCHCLASSINDEX
MATISSE_NOSUCHINDEX
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

The argument *class* is optional. You can specify a class if you want to put an additional constraint on the index stream. For example, if the index groups together instances of two or more classes, you can specify that instances of only one class be associated with the stream. Alternatively, you can set the argument to NULL. Whether or not you specify a class with the argument *class*, the instances that are grouped together in the stream are those whose attributes possess values within the intervals specified by the arguments *startValues* and *endValues*.

The argument *nbOfCriteria* designates the number of criteria taken into account when an index stream is opened. In other words, this argument designates how many elements of the arrays *startValues* and *endValues* to take into account. You can specify NULL for its value. If you set *nbOfCriteria* to 0, the values set for the arguments *startValues* and *endValues* are ignored.

The arguments *startValues* and *endValues* are arrays of pointers. It is possible to leave an interval undefined for one or more criteria. To do this, set the pointer to NULL for the criterion whose interval you want to leave undefined in *startValues* or *endValues*.

The limits that you set with *startValues* and *endValues* must adhere to the following constraint:

```
startValues ≤ endValues
```

To illustrate this concept, suppose you have an index with the two criteria: Name and FirstName. Both of these criteria are built in ascending order. Suppose that you want to search for all the instances indexed that have a value for Name that is equal to or greater than Flanagan and less than or equal to Petrocelli. In addition, all the instances must have a value for FirstName that is greater than or equal to Mike and less than or equal to Rico.

The values for these criteria are valid because the constraint *startValues* ≤ *endValues* is met. Flanagan ≤ Petrocelli and Mike ≤ Rico, as shown in the following table:

Arguments	Last Name	First Name
startValues	Flanagan	Mike
endValues	Petrocelli	Rico

If *startvalues* were (Petrocelli, Rico) and *endValues* were (Flanagan, Mike), then these arguments would not have correct values. Because Petrocelli and Rico are respectively greater than Flanagan and Mike, the constraint *startValues* ≤ *endValues* would not be met.

Note that the compare operator ≤ deals with the ordering of the criteria. If the name criterion had been created in descending order, then the constraint described in the previous paragraph would be the reverse.

The argument *direction* lets you specify a direction for the stream. You can specify a stream that ascends from the instance with the lowest value to the highest, or you can specify the a stream that descends from the instance with the highest value to the lowest value.

When a stream is opened on an index, the index in question is considered frozen. No subsequent modifications made on the index will be visible during the scan. Modifications will be visible when the next stream is opened on the index.

These functions can be called from within a transaction or during a version access.

Adjusting the value of the *nbObjectsPerCall* argument allows you to tune the maximum response time for further calls to `MtCtxNextObject(s)` functions. The greater is the value, the shorter is the overall enumeration.

When using these functions, the function `MtCtxNextIndexEntry` will return the error `MATISSE_INVALMAPFUNCTION`.

See also
CloseStream (p. 47)
GetIndex (p. 77)
GetIndexInfo (p. 78)
NextIndexEntry (p. 111)
NextObject (p. 113)
NextObjects (p. 114)

OpenInstancesStream

Syntax

```
MtSTS MtCtxOpenInstancesStream
  (MtContext ctx, MtStream* instStream,
   MtString className,
   MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenInstancesStream
  (MtContext ctx, MtStream* instStream, MtOid class,
   MtSize nbObjectsPerCall)
```

Purpose These functions initialize the stream of class instances *instStream* with the class specified as an argument. The function MtCtxNextObject (or MtCtxNextObjects) uses the stream to provide the user with the instances of the class className (or class, depending on the function used).

Arguments

*instStream*OUTPUT
The class stream.

*className*INPUT
A class name.

class INPUT
A class.

*nbObjectsPerCall*INPUT
This argument allow you to specify the maximum number of instances that will be retrieved at each server call. You may use the MT_MAX_PREFETCHING keyword to prefetch the maximum number of objects that can be handled in a request to the server.

Result

MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description The name of class is not case sensitive. These functions can be called either from within a transaction or during a version access.

Adjusting the value of the *nbObjectsPerCall* argument allows you to tune the maximum response time for further calls to MtCtxNextObject(s) functions. The greater the value, the shorter the overall enumeration.

See also *OpenOwnInstancesStream* (p. 136)
CloseStream (p. 47)
NextObject (p. 113)
NextObjects (p. 114)

OpenInverseRelationshipsStream

Syntax

```
MtSTS MtCtxOpenInverseRelationshipsStream
  (MtContext ctx, MtStream* iRelStream,
   MtOid object)
```

Purpose

This function opens the inverse relationship stream *iRelStream* on the specified object. The function MtCtxNextProperty uses the stream to provide the inverse relationships present in *object*.

Arguments

*iRelStream*OUTPUT

The inverse relationship stream.

object INPUT

An object identifier.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEALOCKABORT
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

An instance of a class can have a relationship that is not defined in the class.

Example

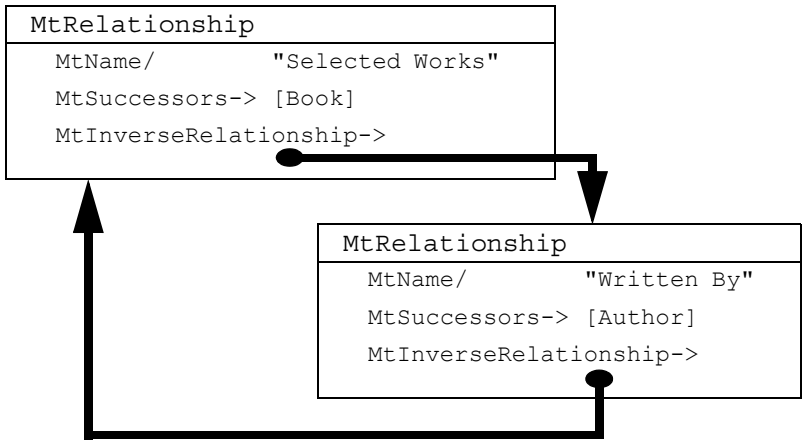
For example, consider the following two class definitions:

```
MtClass
  MtName/              "Author"
  MtAttributes->       Last Name/
  MtRelationships->
```

```
MtClass
  MtName/              "Book"
  MtAttributes->       Title/
  MtRelationships-> Written By->
```

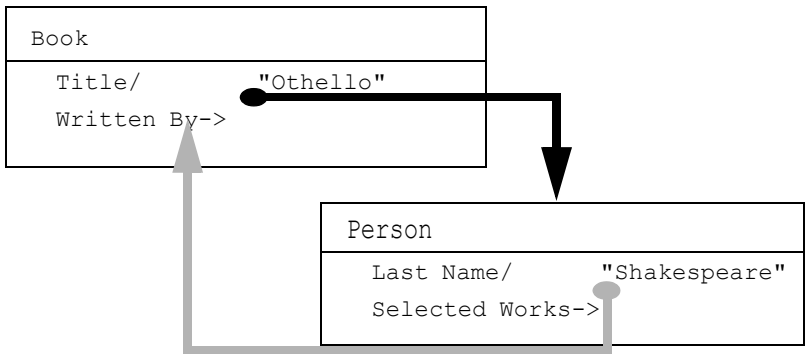
Note that class Book defines a relationship Written By. This relationship of course defines an inverse relationship.

The following diagram illustrates the definitions of the relationship `Written By` and its inverse relationship, `Selected Works`:



Imagine that one instance of `Author` and one instance of `Book` are created and that for the instance of `Book`, the value of `Written By` is assigned to the instance of `Author`.

The following diagram illustrates the resulting link established between an instance of the class `Book` and an instance of the class `Author` through the relationship `Written By`:



You can see that in the instance of `Book`; `Othello`, the relationship `Written By` is assigned to the instance of `Author; Shakespeare`.

Also, the inverse relationship `Selected Works` is created automatically for the instance `Shakespeare`.

A stream opened by the function `MtCtxOpenInverseRelationshipsStream` retrieves only those inverse relationships that exist for an object. An object inverse relationship stream opened on the instance `Shakespeare`, for example, will retrieve the inverse relationship `Selected Works`.

The stream opened by the function `MtCtxOpenInverseRelationshipsStream` gives access to all the inverse relationships that are set for an object.

This function can be called either from within a transaction or during a version access.

Listing Possible Inverse Relationships of a Class

A stream opened by the function `MtCtxOpenInverseRelationshipsStream` retrieves only those inverse relationships that exist for an object.

It is possible to determine all of inverse relationships types that can exist for instances of a particular class. You can retrieve this information with the `GetAllInverseRelationships` functions. These functions return a listing of all possible types of inverse relationships for a class.

`MtCtxGetAllInverseRelRelationships` retrieves information on all the possible inverse relationships implied at the schema level. `MtCtxOpenInverseRelationshipsStream` retrieves all the inverse relationships that have been established for an instance of a given class.

See also

CloseStream (p. 47)
GetAllInverseRelationships (p. 62)
NextProperty (p. 115)

OpenOwnInstancesStream

Syntax

```
MtSTS MtCtxOpenOwnInstancesStream
  (MtContext ctx, MtStream* instStream,
   MtString className,
   MtSize nbObjectsPerCall)
MtSTS MtCtx_OpenOwnInstancesStream
  (MtContext ctx, MtStream* instStream, MtOid class,
   MtSize nbObjectsPerCall)
```

Purpose

These functions initialize the stream of instances of the class specified by `classStream` (subclasses are not initialized) with the class specified as an argument. The function `MtCtxNextObject` (or `MtCtxNextObjects`) uses the stream to provide the user with the instances of the class *`className`* (or *`class`*, depending on the function used). The instances of any subclasses are not returned by this function.

Arguments

`instStream``OUTPUT`
The class stream.

`className``INPUT`
A class name.

`class` `INPUT`
A class.

*nbObjectsPerCall*INPUT

This argument allows you to specify the maximum number of instances that will be retrieved at each server call. You may use the MT_MAX_PREFETCHING keyword to prefetch the maximum number of objects that can be handled in a request to the server.

Result

MATISSE_SUCCESS
MATISSE_CLASSEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALOP
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASS
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

The name of class is not case sensitive. These functions can be called either from within a transaction or during a version access.

Adjusting the value of the *nbObjectsPerCall* argument allows you to tune the maximum response time for further calls to MtCtxNextObject(s) functions. The greater the value, the shorter the time of the overall enumeration.

See also

OpenInstancesStream (p. 132)
CloseStream (p. 47)
NextObject (p. 113)
NextObjects (p. 114)

OpenPredecessorsStream

Syntax

```
MtSTS MtCtxOpenPredecessorsStream
   (MtContext ctx, MtStream* predStream,
    MtOid object,
    MtString relationshipName)
MtSTS MtCtx_OpenPredecessorsStream
   (MtContext ctx, MtStream* predStream,
    MtOid object,
    MtOid relationship)
```

Purpose

These functions initialize the relationship stream *predStream*. The function MtCtxNextObject (or MtCtxNextObjects) uses the stream to provide the user with the predecessors of the object *object* through the relationship *relationshipName* (or *relationship*, depending on the function used).

Arguments

*predStream*OUTPUT

The stream of the relationship.

object INPUT

An object.

*relationshipName*INPUT

A relationship name.

*relationship*INPUT

A relationship object.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALIREL
MATISSE_INVALOP
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_TRANABORTED
MATISSE_WAITTIME

Description

The name of the relationship is not case sensitive. These functions can be called either from within a transaction or during a version access.

See also

CloseStream (p. 47)
NumericGetScale (p. 119)
NextObject (p. 113)

OpenRelationshipsStream

Syntax

```
MtSTS MtCtxOpenRelationshipsStream
   (MtContext ctx, MtStream* relStream,
    MtOid object)
```

Purpose

This function opens the object relationship stream *objectRelStream* on the specified object. The function MtCtxNextProperty uses the stream to provide the relationships that are set for *object*.

Arguments

*relStream*OUTPUT

The object relationship stream.

object INPUT

An object.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEALOCKABORT
MATISSE_INVALOP
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description

Note that the function `MtCtxGetAllRelationships` retrieves all the relationships possible for an instance of a given class. `MtCtxGetAllRelationships` works at the schema level, while `MtCtxOpenRelationshipsStream` works at the data level.

This function can be called from within a transaction or during a version access.

See also

CloseStream (p. 47)
GetAllRelationships (p. 66)
NextProperty (p. 115)

OpenSuccessorsStream

Syntax

```
MtSTS MtCtxOpenSuccessorsStream
  (MtContext ctx, MtStream* succStream,
   MtOid object,
   MtString relationshipName)
MtSTS MtCtx_OpenSuccessorsStream
  (MtContext ctx, MtStream* succStream,
   MtOid object,
   MtOid relationship)
```

Purpose

These functions open the relationship stream *relStream* on *object*. The function `MtCtxNextObject` uses the stream to provide the user with the successors of the object *object* through the relationship *relationshipName* (or *relationship*, depending on the function used).

Arguments

succStream OUTPUT

The stream of relationship.

object INPUT

An object.

*relationshipName*INPUT

A relationship name.

*relationship*INPUT

A relationship.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INVALOP
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHREL
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_TRANABORTED
MATISSE_WAITTIME
```

Description The name of the relationship is not case sensitive. These functions can be called either from within a transaction or during a version access.

See also *CloseStream* (p. 47)
GetSuccessors (p. 93)
NextObject (p. 113)
NextObjects (p. 114)

OpenVersionStream

Syntax

```
MtSTS MtCtxOpenVersionStream
  (MtContext ctx, MtStream* versionStream)
```

Purpose This function initializes the stream of versions stored in the database. The function `MtCtxNextVersion` uses this stream to return the version identifier

Arguments

*versionStream*OUTPUT

The stream of saved versions that exist in the database.

Result

```
MATISSE_SUCCESS
MATISSE_NOCURRENTCONNECTION
```

See also *NextVersion* (p. 116)
StartVersionAccess (p. 169)
CloseStream (p. 47)

PError

Syntax

```
void MtCtxPError (MtContext ctx, MtString comment)
```

Purpose This function prints the entire error message on the stream `stderr`.

Arguments *comment*`INPUT`

The error message is prefixed with the string *comment*.

Example After an error of type `NOSUCHHOST`, the call to `MtCtxPError ("Ask system engineer for help")` results in the following message:

```
Ask system engineer for help: MATISSE_E_NOSUCHHOST, host
bentley not found.
```

Print

Syntax

```
MtSTS MtCtxPrint
  (MtContext ctx, MtOid object, FILE* stream)
```

Purpose This function prints the object *object*.

Arguments *object* `INPUT`

The object to be printed.

stream `INPUT`

The print stream. Use `stdout` if you want the message to be printed to the screen.

Result

```
MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_TRANABORTED
```

Description This function can be called from within a transaction or during a version access.

RemoveAllSuccessors

Syntax

```
MtSTS MtCtxRemoveAllSuccessors
  (MtContext ctx, MtOid object, MtString relationshipName)
MtSTS MtCtx_RemoveAllSuccessors
  (MtContext ctx, MtOid object, MtOid relationship)
```

Purpose These functions remove the relationship *relationshipName* and its successors from *object*.

Arguments

object INPUT

An object.

*relationshipName*INPUT

A relationship name.

*relationship*INPUT

A relationship object.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALMODIF
MATISSE_INVALPROREMOVE
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCCESSORS
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHFUNC
MATISSE_NOSUCHREL
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME

Description

Modifications are checked during MtCtxCommitTransaction.

The name of relationship is not case sensitive. These functions can be called only from within a transaction.

See also

GetRemovedSuccessors (p. 91)
RemoveSuccessors (p. 143)

RemoveObject

Syntax

```
MtSTS MtCtxMtCtxRemoveObject (MtContext ctx, MtOid object)
```

Purpose

This function deletes *object* and updates the inverse links, entry points, and indexes.

Arguments

object INPUT

An object.

Result

```
MATISSE_SUCCESS
MATISSE_CLASSWITHINSTANCES
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALMODIF
MATISSE_INVALOP
MATISSE_INVALSTATUS
MATISSE_MESSWITHINTERP
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHFUNC
MATISSE_NOTRANS
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description

If the object was the only successor of a property, the property is removed from the object that it qualified. If the object was the only object pointed to by an entry point, the entry point is deleted.

During `MtCtxCommitTransaction`, all objects indirectly modified are checked and an error can be generated at this point.

This function can be called only from within a transaction.

RemoveSuccessors

Syntax

```
MtSTS MtCtxRemoveSuccessors
  (MtContext ctx, MtOid object,
   MtString relationshipName,
   MtSize numSuccessors, ...)
MtSTS MtCtx_RemoveSuccessors
  (MtContext ctx, MtOid object,
   MtOid relationship,
   MtSize numSuccessors, ...)
MtSTS MtCtxRemoveNumSuccessors
  (MtContext ctx, MtOid object,
   MtString relationshipName,
   MtSize numSuccessors,
    MtOid* successors)
MtSTS MtCtx_RemoveNumSuccessors
  (MtContext ctx, MtOid object,
   MtOid relationship,
   MtSize numSuccessors,
   MtOid* successors)
```

Purpose These functions remove the successors from the relationship.

Arguments

object INPUT

An object.

*relationshipName*INPUT

A relationship name (a string).

*relationship*INPUT

A relationship object.

*numSuccessors*INPUT

The number of successors to remove.

*successors*INPUT

The array of the successors to be removed.

Other INPUT arguments:

For MtCtxRemoveSuccessors and MtCtx_RemoveSuccessors, the argument *numSuccessors* must be followed by the successors (type MtOid) to be removed.

Result

MATISSE_SUCCESS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALCLASSMODIF10
MATISSE_INVALINDEXMODIF3
MATISSE_INVALINDEXMODIF5
MATISSE_INVALMODIF
MATISSE_INVALNB
MATISSE_INVALPROREMOVE
MATISSE_INVALREL
MATISSE_INVALSTRINGSIZE
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHCLASSREL
MATISSE_NOSUCHFUNC
MATISSE_NOSUCHREL
MATISSE_NOSUCHSUCC
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_RELEXPECTED
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_UNEXPECTEDDUPLICATES
MATISSE_USERERROR
MATISSE_WAITTIME

Description These functions do not apply to inverse relationships.

Modifications are checked and saved on the server during MtCtxCommitTransaction.

The name of relationship is not case sensitive.

These functions can be called only from within a transaction.

See also *GetRemovedSuccessors* (p. 91)
RemoveAllSuccessors (p. 141)

RemoveValue

Syntax

```
MtSTS MtCtxRemoveValue
  (MtContext ctx, MtOid object,MtString attributeName)
MtSTS MtCtx_RemoveValue
  (MtContext ctx, MtOid object, MtOid attribute)
```

Purpose These functions remove the value associated with *attribute* in *object*. *Subsequent calls to retrieve the associated value will return the attribute default value*.

Arguments

object INPUT
An object.

*attributeName*INPUT
An attribute name.

*attribute*INPUT
An attribute object.

Result

```
MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INVALMODIF
MATISSE_INVALSTRINGSIZE
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOSUCHFUNC
MATISSE_NOTRANS
MATISSE_NOVALUE
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description Modifications are validated and saved during `MtCtxCommitTransaction` (the default value must be valid for the object).

The name of attributes is not case sensitive.

These functions can be called only from within a transaction.

See also *SetValue* (p. 151)

SetConnectionOption

Syntax

```
MtSTS MtCtxSetConnectionOption
  (MtContext connection,
   MtConnectionOption option, ...)
```

Purpose

This function sets a connection option.

Arguments

*connection*INPUT

A previously allocated structure that contains the information about the database connection.

*option*INPUT

The connection option to be set. Possible values are:

MT_SERVER_EXECUTION_PRIORITY, MT_LOCK_WAIT_TIME, MT_DATA_ACCESS_MODE, MT_LOCKING_POLICY

...INPUT

The other input arguments are option specific. See below for a full description.

Result

```
MATISSE_SUCCESS
MATISSE_INVALOP
MATISSE_INVALPRIO
MATISSE_INVALWAITTIME
MATISSE_INVALCONNECTOPTION
MATISSE_INVALCONNECTION
```

Description

Connection options affect the way you can interact with the database. You can specify different values for the following options:

- MT_DATA_ACCESS_MODE. This option allows you to specify the type of access that you intend to use when connecting to the database.
  Possible values are:
  - MT_DATA_READONLY allows read only access to the data objects and to the schema. Any attempt to start a transaction will fail (only MtCtxStartVersionAccess is allowed).
  - MT_DATA_MODIFICATION allows read/write access to the data objects and read only access to the schema. This is the default mode.
  - MT_DATA_DEFINITION allows read/write access to the data objects and to the schema.

    The first two access modes optimize the access to the schema. The DATA_DEFINITION access mode must be used only when schema or meta-schema updates are necessary.
    This option cannot be changed when the connection to the database is open.
- MT_LOCK_WAIT_TIME. This option allows you to specify the amount of time the server waits for access conflicts to be resolved; if a timeout occurs (wait-time expires), the explicit or implicit lock request is rejected. The possible values are:

- MT_NO_WAIT: If the lock cannot immediately be granted, the lock request is released and the function returns immediately.
- MT_WAIT_FOREVER: The server waits until there is a deadlock or until the lock is granted. This is the default value.
- A positive integer of type MtLockWaitTime: This is the time (in milliseconds) that the server waits for the lock to be granted. If the wait-time expires, the lock request is rejected. If a deadlock occurs, the transaction fails or the lock request is rejected (explicit locks requested for example through MtCtxLockObjects do not cause a transaction to fail).

  When multiple objects are requested, the wait-time applies to each object request individually. The wait-time affects the process of obtaining locks for reads and writes within transactions. Object version requests are affected neither by locks nor by wait-times.

◆ MT_SERVER_EXECUTION_PRIORITY. This option allows you to specify the priority of the requests that the connection will send to the database server. The higher it is the faster the requests will be executed. The possible values are:

- MT_MIN_SERVER_EXECUTION_PRIORITY. This is the default value.
- MT_NORMAL_SERVER_EXECUTION_PRIORITY
- MT_ABOVE_NORMAL_SERVER_EXECUTION_PRIORITY
- MT_MAX_SERVER_EXECUTION_PRIORITY.

This option cannot be changed when the connection to the database is open.

◆ MT_LOCKING_POLICY. This option allows the server to be configured to handle requests for read locks using write locks instead. The possible values are:

- MT_DEFAULT_ACCESS (default): Normal behavior, requests for read locks result in read locks.
- MT_ACCESS_FOR_UPDATE: Requests for read locks result in write locks.

This option may be changed at any time.

Changing the locking policy to MT_ACCESS_FOR_UPDATE is a conservative approach to prevent deadlocks. It serializes other transactions accessing the same objects and thus may degrade performance, user applications should change the setting back to MT_DEFAULT_ACCESS as soon as practical.

◆ MT_MEMORY_TRANSPORT. This option allows use of the shared memory transport rather than tcp or ticots for local access. The connection is first opened using tcp or ticots, then if shared memory resources are available on the machine, the connection is reopened in shared memory. The possible values are:

- MT_OFF (default): Does not allow shared memory transport for local connection. This option cannot be changed when the connection to the database is open.
- MT_ON: Allows shared memory transport for local connection. The database's confifguration file MEMORYTRANS parameter must be set to 1 (the default is 0) or this MT_ON will have no effect.

- MT_NETWORKTRANS_BUFSZ: Sets the size of a network connection buffer. The values are expressed in kilobytes. Allowed values are 32, 64, 128, and 256. The default value is 64.
- MT_MEMORYTRANS_BUFSZ: Sets the size of a memory transport connection buffer. The values are expressed in kilobytes. Allowed values are 32, 64, 128, and 256. The default value is 64

See also *GetConnectionOption* (p. 74)
ConnectDatabase (p. 50)
DisconnectDatabase (p. 53)

SetListElements

Syntax

```
MtSTS MtCtxSetListElements
  (MtContext ctx, MtOid object, MtString attributeName,
   MtType type,
   void* bufList,
   MtSize* numElts,
   MtSize eltOffset,
   MtBoolean discardAfter)
MtSTS MtCtx_SetListElements
  (MtContext ctx, MtOid object, MtOid attribute,
   MtType type,
   void* bufList,
   MtSize numElts,
   MtSize eltOffset,
   MtBoolean discardAfter)
```

Purpose

These functions store the *bufList* content as a subset of the existing list value of the attribute for the specified object. The subset is stored at *firstEltOffset* and is *numElts* long.

Arguments

object INPUT
An object.

*attributeName*INPUT
An attribute name.

*attribute*INPUT
An attribute.

type INPUT
The expected type of the list value. Possible types are MT_DOUBLE_LIST, MT_FLOAT_LIST, MT_NUMERIC_LIST, MT_SHORT_LIST, MT_INTEGER_LIST, MT_AUDIO, MT_IMAGE, MT_VIDEO, and MT_BYTES.

*bufList*INPUT
The address of a variable allocated by the calling program. The content if this variable is a subset of the list.

*numElts*INPUT

The number of elements of the subset. The maximum list length is limited to MT_LIST_MAX_LEN.

*eltOffset*INPUT

The offset (or position) of an element in the list value. The subset will be stored starting at this offset. The first element in the list value has a 0 offset.

Three specific values are allowed for *eltOffset*:

- MT_BEGIN_OFFSET,
- MT_CURRENT_OFFSET
- MT_END_OFFSET

The MT_CURRENT_OFFSET parameter allows the user to access “the next element immediately after the las accessed element”.

*discardAfter*INPUT

This parameter indicates whether or not the rest of the existing list immediately after the subset (i.e from the element at offset *firstEltOffset* + *numElts* until the end) should be discarded.

Result

MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_INDEXEDATT
MATISSE_INVALLISTOFFSET
MATISSE_INVALLISTSIZE
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOTENOUGHSPACE
MATISSE_NOTRANORVERSION
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_SCHEMAWITHDAEMONS
MATISSE_TRANABORTED
MATISSE_TYPEMISMATCH
MATISSE_TYPENOTALLOWED
MATISSE_WAITTIME

Description

The name of the attribute is not case sensitive. These functions must be called from within a transaction.

Matisse internally manages an offset for each list. This offset is set to *firstEltOffset* + *numElts* after each call to the MtCtx*GetListElements or MtCtx*SetListElements functions. It can be used for subsequent accesses by specifying MT_CURRENT_OFFSET as value for *firstEltOffset* argument. There is no default offset, therefore, MT_CURRENT_OFFSET cannot be specified at the first call. The offset management remains coherent during the same transaction or version access only.

A NULL value is valid for *bufList* if *numElts* is set to 0. Such a call does nothing when *discardAfter* is set to MT_FALSE or if *firstEltOffset* is set to MT_END_OFFSET.

The *type* argument can be different from the existing list type only if *firstEltOffset* is set to 0 and *discardAfter* is set to MT_TRUE. If this condition is not met, the MATISSE_TYPEMISMATCH error status is returned.

CAUTION: This function does not support entry point or index management. An error will be returned if the attribute is an index criteria.

See also *GetListElements* (p. 80)
GetValue (p. 95)
SetValue (p. 151)

SetOwnPassword

Syntax

```
MtSTS MtCtxSetOwnPassword
    (MtContext ctx, MtString oldPassword,
     MtString newPassword)
```

Purpose This function allows currently connected users to update their password.

Arguments

*oldPassword*INPUT
The current password.

*newPassword*INPUT
The new password that will be used upon a subsequent user connection.

Result

MATISSE_SUCCESS
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSECURITY
MATISSE_INVALPASSWDLEN

Description This function can be called when a database connection is selected.

See also *ConnectDatabase* (p. 50)

SetValue

Syntax

```
MtSTS MtCtxSetValue
   (MtContext ctx, MtOid object, MtString attributeName,
    MtType type,
    void* value,
    MtSize rank,
    ...)
```

```
MtSTS MtCtx_SetValue
  (MtContext ctx, MtOid object, MtOid attribute,
   MtType type,
   void* value,
   MtSize rank,
   ...)
```

Purpose These functions update the attribute in the object *object*, with the new value *value*.

Arguments

object INPUT

An object.

attributeName INPUT

An attribute name.

attribute INPUT

An attribute object.

type INPUT

The type of the attribute. Possible types are MT_BOOLEAN, MT_BOOLEAN_LIST, MT_CHAR, MT_DATE, MT_DATE_LIST, MT_DOUBLE, MT_DOUBLE_LIST, MT_FLOAT, MT_FLOAT, MT_FLOAT_LIST, MT_INTERVAL, MT_INTERVAL_LIST, MT_NULL, MT_SHORT, MT_SHORT_LIST, MT_INTEGER, MT_INTEGER_LIST, MT_LONG, MT_LONG_LIST, MT_NUMERIC, MT_NUMERIC_LIST, MT_STRING, MT_STRING_LIST, MT_TIMESTAMP, MT_TIMESTAMP_LIST, MT_BYTE, MT_TEXT, MT_VIDEO, MT_AUDIO, MT_IMAGE and MT_BYTES.

value INPUT

The attribute value. *value* must be a pointer to the value. For the types MT_STRING_LIST, *value* must be an array of pointers (not a two-dimensional character array).

Null pointers are supported in this array.

value can be NULL for the following types:

- MT_STRING
- MT*_LIST, MT_BYTES, MT_AUDIO, MT_VIDEO, MT_IMAGE

rank INPUT

It must be set to 0 when *value* is NULL.

When the value is a list (types MT_*_LIST), the parameter must be set to 1 when *value* is not NULL, or set to 0 otherwise.

When the value is of one of the following types, the parameter must be set to 0: MT_BOOLEAN, MT_CHAR, MT_DATE, MT_DOUBLE, MT_FLOAT, MT_INTERVAL, MT_NULL, MT_NUMERIC, MT_SHORT, MT_INTEGER, MT_LONG, MT_STRING, MT_TEXT, MT_TIMESTAMP, MT_BYTE.

Other INPUT arguments:

When the attribute value is a list or an array and *value* is not NULL, the argument *rank* must be followed by the appropriate dimensions.

When the attribute value is a list or a one-dimensional array, there must be only one value that indicates the size of the array or the number of elements of the list. The maximum list length is `MT_LIST_MAX_LEN`.

When the attribute value is a multidimensional array, there must be *n* number of values (where *n* equals *rank*). If this is the case, each value indicates the size of the array in the dimension. Refer to the sample code that follows in the results section below.

Result

```
MATISSE_SUCCESS
MATISSE_ATTEXPECTED
MATISSE_CLASSEXISTS
MATISSE_CONNLOST
MATISSE_DEADLOCKABORT
MATISSE_FROZENOBJECT
MATISSE_INCOMPRANKVALUE
MATISSE_INCOMPTYPE
MATISSE_INDEXEXISTS
MATISSE_INVALATTMODIF2
MATISSE_INVALATTMODIF3
MATISSE_INVALATTMODIF5
MATISSE_INVALATTMODIF6
MATISSE_INVALCARDINALITY
MATISSE_INVALDIM
MATISSE_INVALINDEXMODIF1
MATISSE_INVALMODIF
MATISSE_INVALOP
MATISSE_INVALRANK
MATISSE_INVALSTATUS
MATISSE_INVALNAMESIZE
MATISSE_INVALSTRINGSIZE
MATISSE_INVALTIMESTAMP
MATISSE_INVALTIMEINTERVAL
MATISSE_INVALTYPE
MATISSE_METASCHEMAOBJECT
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHATT
MATISSE_NOSUCHCLASSATT
MATISSE_NOSUCHFUNC
MATISSE_NOTRANS
MATISSE_NULLPOINTER
MATISSE_OBJECTDELETED
MATISSE_OBJECTNOTFOUND
MATISSE_PROPERTYEXISTS
MATISSE_RECURSIVESETVALUE
MATISSE_SELECTOREXISTS
MATISSE_SFUNCERRORABORT
MATISSE_TRANABORTED
MATISSE_USERERROR
MATISSE_WAITTIME
```

Description The value of the attribute is modified, the entry-point is updated if there is an entry-point function, the entries for the object in any index attached to the class are updated.

Entry points and the name of attributes are not case sensitive.

These functions can be called either from within a transaction or during a version access.

NOTE: For the type `MT_STRING_LIST`, *`value`* must be an array of pointers (and not a two-dimensional array of characters).

NOTE: with `MtCtxSetValue`, an attribute cannot be removed (i.e., its new value corresponds to the default value). When an attribute is specified, even with the value of type `MT_NULL` or with a value equal to the default value defined for this property, it is saved. If a property value is equal to the property default value, and if the default value is modified, the property still has the same value. If the property is not specified, its value corresponds to the new default value. In order to make an attribute unspecified, use `MtCtxRemoveValue`.

CAUTION: Under no circumstances should *`value`* be set to a variable of type `MtOid`. The definition of the programming type `MtOid` may change in future releases of Matisse. You must always use relationships to establish links between objects.

CAUTION: You should set the value of an attribute by passing a variable of a datatype that corresponds to what is passed as the *`type`* argument.

Example

```
MtOid objOid;
MtOid propOid;
MtInteger value=22;
MtInteger tab1[] = {1, 2, 8, 1, 9};
MtByte tab2[5][3] = {{'1', 'b', '7'},
                     {'2', 'c', '8'},
                     {'3', 'd', '9'},
                     {'4', 'e', '0'},
                     {'5', 'f', '1'}};
...
/* Insertion of an integer of type MT_INTEGER
 * in the object objOid
 * for the attribute propOid
 */
MtCtxSetValue(objOid, propOid, MT_INTEGER, &value, 0);

/* Insertion of a one-dimension array of
 * integers in the object objOid for the
```

```
 * attribute propOid.
 */
MtCtxSetValue
  (objOid, propOid, MT_INTEGER_LIST, tab1, 1, 5);

/* Insertion of a two-dimension array of
 * characters in the object objOid for the
 * attribute propOid
 */
MtCtxSetValue
  (objOid, propOid,
   MT_BYTES, tab2, 2, 5, 3);
```

See also

GetListElements (p. 80)
GetValue (p. 95)
GetObjectsFromEntryPoint (p. 83)
OpenEntryPointStream (p. 125)
RemoveValue (p. 145)
GetListElements (p. 80)

SQLAllocStmt

Syntax
```
MtSTS MtCtxSQLAllocStmt (MtContext ctx, MtSQLStmt* stmt)
```

Arguments
```
stmt OUTPUT
```
Statement handle.

Result
```
MATISSE_SUCCESS
```

Purpose Allocate a new SQL statement.

SQLExecDirect

Syntax
```
MtSTS MtCtxSQLExecDirect
  (MtContext ctx, MtSQLStmt stmt,
   MtString stmtStr)
```

Arguments Parameters must be provided as literal constants.

```
stmt INPUT
```
Statement handle.

```
stmtStr INPUT
```
The SQL statement to be executed.

Result
```
MATISSE_SUCCESS
```

All MATISSE error status results are possible.

Purpose Execute a SQL statement. The statement to be executed is contained in the *stmtStr*.

Description A statement is executed in a transaction context or a version (read-only) context. The context is usually set in the application, if not the SQL execution automatically starts a version context for read-only statements like `SELECT`, or a transaction context for statements performing updates like `INSERT`, `DELETE` and `UPDATE`.

The following example shows how to allocate a statement, execute it and retrieve values from the result set.

```
MtSTS sts;
MtSQLStmt stmt;
MtSize size;
MtType type;
char name[32];

sts = MtCtxSQLAllocStmt (&stmt);
sts = MtCtxSQLExecDirect (stmt,
          "SELECT FirstName FROM person");

if ( MtFailure(sts) ) {
  printf ("Error!! code = %d, message = %s\n", sts,
          MtCtxError());
  return ...;
}

/* open a row stream on the result set */
sts = MtCtxSQLOpenStream (&stream, stmt);

/* Get the type and value for the first column */
MtCtxSQLNext (stream);
size = 32;
MtCtxSQLGetRowValue(stream, 1, &type, name, &size);

sts = MtCtxCloseStream (stream);
sts = MtCtxSQLFreeStmt (stmt);
```

The next example shows how to use the `REF()` function within the select list of a `SELECT` statement to return object identifiers, and then directly access the attributes and relationships from the objects.

```
MtSTS sts;
MtSQLStmt stmt;
MtSize size;
MtType type;
MtOid obj;
char name[32];

sts = MtCtxSQLAllocStmt (&stmt);

sts = MtCtxSQLExecDirect (stmt, "SELECT REF(p) FROM person
p");
if ( MtFailure(sts) ) {
  printf ("Error!! code = %d, message = %s\n", sts,
          MtCtxError());
  return ...;
}

sts = MtSQLOpenStream (&stream, stmt);

MtSQLNext ();
while ( MtSQLNext (stream) == MATISSE_SUCCESS ) {
  /* first get the object id */
  size = sizeof (obj);
  MtSQLGetRowValue (stream, 1, (void*)&obj, &size);

  /* access the attributes from the object id */
  size = 32;
  MtGetValue (obj, "FirstName", &type, name, 0, &size, 0);
}

sts = MtCloseStream (stream);
sts = MtSQLFreeStmt (stmt);
```

SQLFreeStmt

Syntax `MtSTS MtCtxSQLFreeStmt (MtContext ctx, MtSQLStmt` *`stmt`*`)`

Arguments *`stmt`* `INPUT`

Statement handle.

Result `MATISSE_SUCCESS`
`MATISSE_INVALSTMT`

Purpose Free a SQL statement.

Before freeing a SQL statement, you must make sure that there is no currently open stream on the result set for this statement.

SQLGetColumnInfo

Syntax

```
MtSTS MtCtxSQLGetColumnInfo
  (MtContext ctx, MtSQLStmt stmt,
   MtSize colNum,
   MtType* coltype,
   MtString colname,
   MtSize* sz)
```

Arguments

stmt INPUT

Statement handle.

colNum INPUT

Column number, starting at 1.

coltype OUTPUT

The column type.

colname OUPUT

The column name.

sz INPUT/OUTPUT

Column name length.

Result

MATISSE_SUCCESS
MATISSE_INVALARG

Purpose This function returns the column type and the column name for a given column. It can be used after successful completion of a SELECT statement.

SQLGetParamDimensions

```
MtSTS MtCtxSQLGetParamDimensions
  (MtContext ctx, MtSQLStmt stmt,
   MtSize  paramNumber,
   MtSize* rank,
   MtSize  dimensions)
```

Arguments

stmt INPUT

SQL statement.

paramNumber INPUT

Index of parameter, starting from 1 or MTSQL_RETVALUE for the return value. Currently only MTSQL_RETVALUE is supported.

rank OUTPUT

Number of dimensions.

dimensions OUTPUT

Dimensions.

Result

MATISSE_SUCCESS
MATISSE_INVALARG

Purpose

Get rank and dimensions for the list and array values. Caller should pass an array of 8 dimensions. See also MtCtxGetValue and MtCtxGetDimension in the *MATISSE C API Reference* for details of how to handle list and array values.

This function can be called after successful completion of a CALL statement or a block statement.

SQLGetParamListElements

```
MtSTS MtCtxSQLGetParamListElements
  (MtContext ctx, MtSQLStmt stmt,
   MtSize  paramNumber,
   MtType  type,
   void*   buf,
   MtSize* buf_size,
   MtSize  firstEltOffset)
```

Arguments

stmt INPUT

Statement handle.

paramNumber INPUT

Index of parameter, starting from 1 or MTSQL_RETVALUE for the return value. Currently only MTSQL_RETVALUE is supported.

type OUTPUT

Type of the value. Can be set to NULL.

buf OUTPUT

Space to copy the value. Can be set to NULL.

buf_size INPUT/OUTPUT

Buffer size. MtCtxSQLGetParamValue() returns NOTENOUGHSPACE error if there is not enough space to copy data.

firstEltOffset INPUT

Offset of the first element of the list to be copied, starting at 0.

Result

MATISSE_SUCCESS
MATISSE_INVALARG

Purpose

Retrieve a portion of the list value for this parameter. The subset begins at firstEltOffset. The interface is similar to MtCtxSQLGetRowListElements.

This function can be called after successful completion of a CALL statement.

SQLGetParamValue

```
MtSTS MtCtxSQLGetParamValue
  (MtContext ctx, MtSQLStmt stmt,
   MtSize paramNumber,
   MtType* type,
   void* value,
   MtSize* size)

MtSTS MtCtxSQLMGetParamValue
  (MtContext ctx, MtSQLStmt stmt,
   MtSize paramNumber,
   MtType* type,
   void** value,
   MtSize* size)
```

Arguments

stmt INPUT

Statement handle.

paramNumber INPUT

Index of parameter, starting from 1 or MTSQL_RETVALUE for the return value. Currently only MTSQL_RETVALUE is supported.

type OUTPUT

Type of the value. Can be set to NULL.

value OUTPUT

Space to copy the value. Can be set to NULL.

size INPUT/OUTPUT (using MtCtxSQLGetParamValue)
size OUTPUT (using MtCtxSQLMGetParamValue)

Buffer size. `MtCtxSQLGetParamValue()` returns `NOTENOUGHSPACE` error if there is not enough space to copy data.

Result

```
MATISSE_SUCCESS
MATISSE_INVALARG
```

Purpose Get the return value of the SQL method invoked. The interface is similar to `MtCtxSQLGetRowValue`.

Description This function can be called after successful completion of a `CALL` statement or a block statement.

The following example shows how to retrieve the value returned by a `CALL` statement.

```
MtSTS sts;
MtSQLStmt stmt;
MtInteger value;
MtType type;
MtSize size = sizeof (value);

sts = MtCtxSQLAllocStmt (&stmt);

/* call the static method 'bonus' on class 'employee' */
sts = MtCtxSQLExecDirect (stmt,
             "CALL employee.bonus(12, 'Smith')" );

/* Get the return value */
sts = MtCtxSQLGetParamValue (stmt, MTSQL_RETVALUE, &type,
                        (void*) &value, &size);
  ...
sts = MtCtxSQLFreeStmt (stmt);
```

SQLGetRowListElements

Syntax

```
MtSTS MtCtxSQLGetRowListElements
  (MtContext ctx, MtStream stream,
   MtSize colNum,
   MtType colType,
   void* bufList,
   MtSize* numElts,
   MtSize firstEltOffset)
```

Arguments *stream* INPUT

A stream opened on a SELECT statement after successful execution.

colNum INPUT

Column number, starting at 1.

colType INPUT

The column type. Can be set to one of the media types (MT_AUDIO, MT_IMAGE, MT_VIDEO) or MT_BYTES.

bufList OUTPUT

This argument contains the address of a buffer allocated by the calling program. The subset retrieved is copied in this buffer.

numElts INPUT/OUTPUT

In input, this parameter indicates the maximum number of elements to be read for the subset. In output it indicates the actual number of elements read.

firstEltOffset INPUT

This parameter indicates the offset (or position) of the first element of the subset to be retrieved. The first element of the stored list has the offset 0.

Two specific values are allowed for *firstEltOffset*:

- MT_BEGIN_OFFSET
- MT_CURRENT_OFFSET

MT_CURRENT_OFFSET means “the next element immediately after the last accessed element”.

Result

MATISSE_SUCCESS
MATISSE_INVALARG
MATISSE_INVALLISTOFFSET
MATISSE_NOTENOUGHSPACE
MATISSE_NULLPOINTER
MATISSE_TYPEMISMATCH
MATISSE_TYPENOTALLOWED

Purpose

This function allows reading of a large attribute chunk by chunk directly from the server, without internal caching in the MATISSE client.

When a program calls MtCtxSQLGetRowListElements, MATISSE does *not* allocate any memory space. This function copies the subset, according to *numElts*, into a buffer allocated by the calling program.

MATISSE internally manages an offset for each list value. This offset is set to *firstEltOffset + numElts* after every call to the MtCtxSQLGetRowListElements function. The offset can be used for further access by specifying MT_CURRENT_OFFSET as value for the *firstEltOffset*

argument. There is no default offset so MT_CURRENT_OFFSET cannot be specified at the first call. The offset management remains coherent only within the same transaction or version access.

Note that you need to call MtCtxSQLNext() before calling this function.

SQLGetRowValue

Syntax

```
MtSTS MtCtxSQLGetRowValue
  (MtContext ctx, MtStream stream,
   MtSize colNum,
   MtType* colType,
   void* value,
   MtSize* size)

MtSTS MtCtxSQLMGetRowValue
  (MtContext ctx, MtStream stream,
   MtSize colNum,
   MtType* colType,
   void** value,
   MtSize* size)
```

Arguments

stream INPUT

A stream opened on a successfully executed SELECT statement.

colNum INPUT

Column number, starting at 1.

colType OUTPUT

The column type. Can be set to NULL, in which case the function does not return the type of the column.

value OUTPUT

For the function MtCtxGetRowValue that does not allocate memory—this argument is the address of a buffer allocated in the calling program. After the function is called, the value retrieved is copied in this buffer.

For the function MtCtxMGetRowValue that allocates memory—this argument is the address of a pointer variable declared in the calling program. After this function is called, the pointer contains the address of a buffer that contains the value retrieved by the function.

Can be set to NULL, in which case the function does not return the value of the attribute.

size INPUT/OUTPUT

In input, only for the function MtCtxGetRowValue, *size* corresponds to the size in bytes of the buffer provided by the user. In output for both functions, *size* corresponds to the size of the buffer that contains the value that is returned.

Can be set to NULL in which case the function does not return the size. In this case, the argument *value* must also be set to NULL.

In output, for all of the functions, *size* corresponds to the size of the value that is returned. When the stored value is NULL, then size is equal to 0.

Result

```
MATISSE_SUCCESS
MATISSE_INVALARG
MATISSE_NOTENOUGHSPACE
MATISSE_NULLPOINTER
```

Purpose

When a program calls MtCtxSQLGetRowValue, MATISSE does *not* allocate any memory space. This function copies the value into a buffer allocated by the calling program.

It is preferable to use this function to retrieve values whose size is fixed, i.e., for the values of type MT_BOOLEAN, MT_BYTE, MT_SHORT, MT_INTEGER, MT_LONG, MT_FLOAT, MT_DOUBLE, MT_NUMERIC, MT_CHAR, MT_DATE, MT_TIMESTAMP, MT_TIME_INTERVAL. In these cases, this function's memory management is better than MtCtxSQLMGetRowValue's.

When a program calls MtCtxSQLMGetRowValue, MATISSE allocates sufficient space for the value. The program must declare a variable of the appropriate type and then pass the address of this variable to the function. When the data is no longer used, you have to free the space, using the MtMFree function.

Note that you need to call MtCtxSQLNext() before calling these functions.

SQLGetStmtInfo

Syntax

```
MtSTS MtCtxSQLGetStmtInfo
   (MtContext ctx, MtSQLStmt stmt,
    MtSQLStmtAttr stmtAttr,
    void* value,
    MtSize* size)
```

Arguments

stmt INPUT

Statement handle.

stmtAttr INPUT

Statement attributes to retrieve.

value OUTPUT

String containing the attribute value.

size INPUT/OUTPUT

In input, size in bytes of the value specified by the user. In output, size of the value that is returned.

Result

```
MATISSE_SUCCESS
MATISSE_INVALARG
```

Purpose

This function can be called after execution of a SQL statement to obtain some information about the statement.

Table 3.1 SQL Statement Attributes

MtSQLStmtAttr	SQL Statement	description
MTSQL_STMT_OPTION	SET OPTION	set option
MTSQL_STMT_VALUE	SET OPTION	value for set option
MTSQL_STMT_NUMOBJECTS	SELECT, INSERT, UPDATE, DELETE	Number of objects returned or updated
MTSQL_STMT_NUMQUALIFIED	SELECT	Number of objects qualified, not affected by SET MAXOBJECTS
MTSQL_STMT_ERRPOSITION	any	Syntax error position
MTSQL_STMT_ERRLINE	any	Syntax error line
MTSQL_STMT_READONLY	SET TRANSACTION	Start version or transaction access
MTSQL_STMT_VERSION	SET TRANSACTION, COMMIT	Version name
MTSQL_STMT_PRIORITY	SET TRANSACTION	Transaction priority
MTSQL_STMT_SELECTION	DROP SELECTION, SELECT INTO	Selection name
MTSQL_STMT_CLASS	CREATE, ALTER, DROP	Class name
MTSQL_STMT_SUPERCLASS	CREATE, ALTER, DROP	Superclass name
MTSQL_STMT_ATTRIBUTE	CREATE, ALTER, DROP	Attribute name
MTSQL_STMT_RELATIONSHIP	CREATE, ALTER, DROP	Relationship name
MTSQL_STMT_INDEX	CREATE, DROP	Index name
MTSQL_STMT_ENTRYPOINT	CREATE, DROP	Entry point dictionary name

SQLGetStmtType

Syntax

```
MtSTS MtCtxSQLGetStmtType
   (MtContext ctx, MtSQLStmt stmt,
    MtSQLStmtType* stmtType)
```

Arguments

stmt INPUT

Statement handle.

stmtType OUTPUT

Statement Type. See table.

Result

MATISSE_SUCCESS
MATISSE_INVALARG

Purpose

Get the statement type. The statement type of a newly allocated statement is MTSQL_ALLOCATED, after successful execution it indicates the type of SQL statement that has been executed.

Table 3.2 SQL Statement Types

MtSQLStmtType	description
MTSQL_ALLOCATED	not yet executed
MTSQL_SELECT	execute select
MTSQL_SET_TRANSACTION	set transaction
MTSQL_SET_OPTION	set option
MTSQL_DROP_SELECTION	drop selection
MTSQL_COMMIT	commit
MTSQL_ROLLBACK	rollback
MTSQL_UPDATE	execute update
MTSQL_DELETE	execute delete
MTSQL_INSERT	execute insert
MTSQL_ALTER_ADD	alter
MTSQL_ALTER_DROP	alter
MTSQL_ALTER_ALTER	alter
MTSQL_DROP	drop
MTSQL_CREATE	create class, method..
MTSQL_METHOD	execute call method
MTSQL_PROCEDURE	execute block statement
MTSQL_ERROR	syntax or execution error

SQLNext

Syntax

```
MtSTS MtCtxSQLNext
  (MtContext ctx, MtStream stream)
```

Arguments

*stream*INPUT

A stream opened on a SELECT statement after successful execution.

Result

MATISSE_SUCCESS
MATISSE_ENDOFSTREAM
MATISSE_INVALARG

Purpose

Fetch the next row from a result set produced by the successful execution of an SQL SELECT statement. The values for the columns of the current row can then be retrieved with the functions MtCtxSQL*GetRowValue and MtCtxSQLGetRowListElements.

SQLNumResultCols

Syntax

```
MtSTS MtCtxSQLNumResultCols
  (MtContext ctx, MtSQLStmt stmt,
   MtSize* numcols)
```

Arguments

stmt INPUT

Statement handle.

numcols OUTPUT

Number of columns in the result set.

Purpose

Return the number of columns from the result set produced by the successful execution of a `SELECT` statement.

SQLOpenStream

Syntax

```
MtSTS MtCtxSQLOpenStream
  (MtContext ctx, MtStream* stream.
   MtSQLStmt stmt)
```

Arguments

stream OUTPUT

SQL projection stream.

stmt INPUT

Statement handle.

Purpose

Open a stream on a successfully executed `SELECT` statement. The stream can then be used with `MtSQLNext()` to visit each row in SQL projection.

Note: `MtNextObject()` cannot be used with this type of stream.

StartTransaction

Syntax

```
MtSTS MtCtxStartTransaction
  (MtContext ctx, MtTranPriority priority)
```

Purpose

This function starts a transaction.

Arguments

*priority*INPUT

The value used by the server to solve access conflicts (in case of deadlock). The value must fall between `MT_MIN_TRAN_PRIORITY` (lowest priority) and `MT_MAX_TRAN_PRIORITY` (highest priority).

Result

```
MATISSE_SUCCESS
MATISSE_INVALOP
MATISSE_INVALPRIO
MATISSE_NESTEDTRANS
MATISSE_NOCURRENTCONNECTION
MATISSE_STREAMCLOSED
MATISSE_TRANSDISABLED
MATISSE_TRANSNOTALLOWED
MATISSE_VERSIONMODE
```

Description

A transaction is the smallest granularity operation on a database. It is atomic: all the elements of the transaction either succeed or fail. If they fail, the transaction is aborted. An abort may be initiated by the server or by the user.

Within a transaction, access to the database may be blocked for various reasons:

- If competing transactions mutually prohibit access (deadlock), one of the transactions is aborted (depending on transaction priority) or if the cache is flushed;
- If the wait-time is exceeded or if a Matisse error occurs, an error status is returned.

The cache is flushed upon exiting a transaction: all objects read into client memory during the transaction are deleted and all the locks on these objects are released.

A transaction is relative to a single connection only.

The number of locks that are granted is proportional to the number of objects that a transaction modifies. Therefore, transactions that modify objects should be as short as possible to avoid affecting other users.

See also

AbortTransaction (p. 42)
CommitTransaction (p. 48)

StartVersionAccess

Syntax

```
MtSTS MtCtxStartVersionAccess
  (MtContext ctx, MtString versionName)
```

Purpose

This function starts a sequence for a version access.

Arguments

*versionName*INPUT

The identifier of an instance view of the database (defined on a previous `MtCtxCommitTransaction`) To access the current version, this argument should be set to `NULL`.

Result

```
MATISSE_SUCCESS
MATISSE_INVALOP
```

```
MATISSE_NESTEDVERSION
MATISSE_NOCURRENTCONNECTION
MATISSE_NOSUCHVERSION
MATISSE_STREAMCLOSED
MATISSE_TRANSOPENED
```

Description

Historical versions are stamped by a string specific to each version.

Within the scope of an `MtCtxStartVersionAccess - MtCtxEndVersionAccess`, you can access either a version of the database that has been previously saved or the current version. The latter option will allow you to access the most recent version of the database objects without having to enter a transaction context.

In order to access a specific version, specify the string that is returned by `MtCtxCommitTransaction` as an argument of `MtCtxStartVersionAccess`.

Within the scope of `MtCtxStartVersionAccess - MtCtxEndVersionAccess`, it is not permitted to perform object modifications.

This function cannot be called within a transaction.

See also

EndVersionAccess (p. 54)
OpenVersionStream (p. 140)

Success

Syntax

```
int MtSuccess (MtSTS status)
```

Purpose

This macro indicates whether or not a Matisse function has executed successfully.

Arguments

status INPUT

The status returned by a Matisse function.

Result

`0` if the status corresponds to a failure; a non-null integer otherwise.

See also

Failure (p. 56)

TimestampAdd

Syntax

```
MtSTS MtTimestampAdd
  (MtTimestamp *result,
   MtTimestamp *time
   MtInterval *interval)
```

Purpose

This function adds an `MtInterval` value to an `MtTimestamp` value.

Arguments

result OUTPUT

Result value.

time INPUT

Timestamp value.

*interval*INPUT

Interval value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMESTAMP

See also

TimestampDiff (p. 172)
TimestampSubtract (p. 174)

TimestampBuild

Syntax

```
MtSTS MtTimestampBuild
  (MtTimestamp *result
   MtString buffer,
   MtTimeZone timeZone)
```

Purpose

This function builds an MtTimestamp value from a text representation.

Arguments

result OUTPUT

Timestamp result value.

*buffer*INPUT

A character string representing a time in the following format:
YYYY-MM-DD HH-mm-SS[:uuuuuu]

timeZone INPUT

Time zone for the string representation, can be either MT_LOCAL_TIMESTAMP or MT_UNIVERSAL_TIMESTAMP.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMESTAMP

Description

The MtTimestamp structure fields are extracted if *buffer* is in the right format and represents a valid time.
For example:

```
MtTimestampBuild("1997-02-30 20:00:33", & time);
```

will return MT_INVALID_TIMESTAMP because february 30 does not exist.

If the MT_LOCAL_TIMESTAMP time zone is specified, the value is converted from the local time zone to universal time, which is also known as UTC. With the MT_UNIVERSAL_TIMESTAMP time zone no time conversion is applied.

To ensure the portability of applications across different time zones, all time values should be stored in universal time.

See also *TimestampPrint* (p. 173)

TimestampCompare

Syntax

```
MtSTS MtTimestampCompare
  (MtInteger *result,
   MtTimestamp *time1,
   MtTimestamp *time2)
```

Purpose

This function compares the first `MtTimestamp` argument to the second `MtTimestamp` argument.

Arguments

result OUTPUT

Comparison result.

time1 INPUT

A `timestamp value`.

time2 INPUT

A `timestamp value`.

Result

`MATISSE_SUCCESS`
`MATISSE_NULLPOINTER`
`MATISSE_INVALID_TIMESTAMP`

Description

Returns an integer greater than, equal to or less than 0 if the first argument is greater than, equal to, or less than the second one respectively.

TimestampDiff

Syntax

```
MtSTS MtTimestampDiff
  (MtInterval *result,
   MtTimestamp *time1,
   MtTimestamp *time2)
```

Purpose

This function subtracts the second `MtTimestamp` argument from the first `MtTimestamp` argument.

Arguments

result OUTPUT

Interval result value.

time1 INPUT

A timestamp value.

time2 INPUT

A timestamp value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMESTAMP

Description Returns an MtInterval value representing the time interval between the *time1* and *time2* arguments.

See also *TimestampAdd* (p. 170)
TimestampSubtract (p. 174)

TimestampGetCurrent

Syntax

```
MtSTS MtTimestampGetCurrent (MtTimestamp *currentTime)
```

Purpose This functions returns the current timestamp.

Arguments *currentTime*OUTPUT

The current timestamp.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER

Descrtiption Returns a MtTimestamp value representing the current UTC timestamp.

See also *CurrentDate* (p. 53)

TimestampPrint

Syntax

```
MtSTS MtTimestampPrint
  (MtString buffer,
   MtSize *bufferSize,
   MtString format,
   MtTimeStamp *time,
   MtTimeZone timeZone)
```

Purpose This function outputs *time* according to *format* into the character string pointed by *buffer*.

Arguments *time* INPUT

The MtTimestamp value to print.

*format*INPUT

A character string containing directives to output the different time fields; possible directives are:

%Y year, including century (for example, 1988)

%y year within century (00..99)

%B month, using full month names

%b month, using abbreviated month names

%m month number (01..12)

%D day of month (01..31)

%H hour (00..23)

%M minute (00..59)

%S seconds (00..59)

%U microseconds (000000..999999)

%% same as %

buffer OUTPUT

A character string into which the time desired time representation will be placed.

*bufferSize*INPUT

An integer indicating the maximum number of character that can be placed into *buffer*.

timeZone INPUT

Time zone for the string representation, can be either MT_LOCAL_TIMESTAMP or MT_UNIVERSAL_TIMESTAMP.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMESTAMP

Description

The timestamp value is assumed to be a universal time value.

If the MT_LOCAL_TIMESTAMP time zone is specified, the character string value is converted from universal time to the local time zone. With the MT_UNIVERSAL_TIMESTAMP time zone no time conversion is applied.

To ensure the portability of applications across different time zones, all time values should be stored in universal time.

See also *TimestampGetCurrent* (p. 173)

TimestampSubtract

Syntax

```
MtSTS MtTimestampSubtract
  (MtTimestamp *result,
   MtTimestamp *time
   MtInterval *interval)
```

Purpose

This function subtracts an MtInterval value to a MtTimestamp value.

Arguments

result OUTPUT

Result value.

time INPUT

Timestamp value.

*interval*INPUT

Interval value.

Result

MATISSE_SUCCESS
MATISSE_NULLPOINTER
MATISSE_INVALID_TIMESTAMP

See also

TimestampAdd (p. 170)
TimestampDiff (p. 172)

4 Error Code Reference

This section lists the errors that may result from the use of the Object Oriented Services.

ALREADYSUCC

successor *object* already exists

This error occurs when one of the following functions is called:

MtCtxAddNumSuccessors
MtCtx_AddNumSuccessors
MtCtxAddSuccessor
MtCtx_AddSuccessor
MtCtxAddSuccessors
MtCtx_AddSuccessors

and when one of the successors to be added is already present in the object for the defined relationship.

AMBIGUOUS_IDENTIFIER

In a SQL statement, the same identifier is used to specify a class and a selection or a property and a selection.

ARG_OUTOFBOUND

A numeric argument for a SQL function is out of bounds.

ARRAYTOOSMALL

array too small. *x* elements needed

This error occurs when one of the following functions is called:

MtCtx_GetAddedSuccessors

MtCtx_GetAllAttributes

MtCtx_GetAllInverseRelationships

MtCtx_GetAllRelationships

MtCtx_GetAllSublasses

MtCtx_GetAllSuperclasses

MtCtx_GetObjectsFromEntryPoint

MtCtx_GetPredecessors

MtCtx_GetRemovedSuccessors

MtCtx_GetSuccessors

MtCtxGetAddedSuccessors

MtCtxGetAllAttributes

MtCtxGetAllInverseRelationships

MtCtxGetAllRelationships

MtCtxGetAllSubclasses

MtCtxGetAllSuperclasses

MtCtxGetObjectsFromEntryPoint

MtCtxGetPredecessors

MtCtxGetRemovedSuccessors

MtCtxGetSuccessors

and when the size of the array specified by the user to position the objects is too small. This size is specified in the first argument.

Solution Set a higher value to the first argument of the function.

ATTEXPECTED

object is not an attribute

This error occurs when calling one of the following functions:

MtCtx_GetDimension
MtCtx_GetObjectsFromEntryPoint
MtCtx_GetValue
MtCtx_LockObjectsFromEntryPoint
MT_MGetObjectsFromEntryPoint
MtCtx_MGetValue
MtCtx_OpenEntryPointStream
MtCtx_RemoveValue
MtCtx_SetValue

and when the specified identifier is not an attribute identifier.

CLASSEXISTS

"*class_name*" is already the name of the class *class*

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling one of the following functions:

- MtCtxSetValue
- MtCtx_SetValue

This error indicates that the class external name is already that of a class. Two different classes cannot share the same name, so the error is returned and the transaction aborted.

Solutions Change the class name.

CLASSEXPECTED

object is not a class

This error occurs when access functions to Matisse objects are called, with the Oid of a Matisse object that is not of the class type being specified as argument (the Oid of a Matisse object of some other type is specified instead).

This error can occur when calling one of the following functions:

MtCtx_CreateObject
MtCtx_GetAllAttributes
MtCtx_GetAllInverseRelationships
MtCtx_GetAllRelationships
MtCtx_GetAllSubclasses
MtCtx_GetAllSuperclasses
MtCtx_GetInstancesNumber
MtCtx_GetObjectsFromEntryPoint
MtCtx_IsInstanceOf
MtCtx_LockObjectsFromEntryPoint
MtCtx_MGetAllAttributes
MtCtx_MGetAllInverseRelationships
MtCtx_MGetAllRelationships
MtCtx_MGetAllSubclasses
MtCtx_MGetAllSuperclasses
MtCtx_MGetObjectsFromEntryPoint
MtCtx_OpenEntryPointStream
MtCtx_OpenInstancesStream

CLASS_NAME_USED

In a SQL statement, the identifier in the INTO clause is the name of a class.

CLASSWITHINSTANCES

you cannot remove *Class* which has instances

This error occurs in MT_DATA_DEFINITION connection mode exclusively when calling the function MtCtxRemoveObject.

It occurs when the user tries to remove a class which has instances (either the class's own instances, or the instances of the class's subclass). The transaction is aborted.

Solutions Remove the instances.

CONNECTREJECT

Connection rejected by database *database* on host *host*

This error can occur with the command `mt_init_database`, or when calling the function `MtCtxConnectDatabase`.

You probably want to connect to a stand-alone server from a remote host.

`CONNLOST`

`Connection with database database on host host has been lost`

This error can occur during any server access.

Solutions Try to reconnect.

Check if the server machine is OK.

`CONNTIMEOUT`

`Database database on host host is not responding`

This error can occur with the command `mt_init_database`, or when calling the function `MtCtxConnectDatabase`. This error indicates that the host did not respond in the allotted time.

Solution Try again. And if you still have the same problem, determine whether or not the server process is sleeping.

`CONSTANT_TOO_LONG`

In a SQL statement, a constant is too long.

`DBALREADYINITED`

`database is already initialized`

This error can occur when executing the command `mt_init_database`.

It occurs when the database has already been initialized and the user tries to initialize it again.

Solutions Check your database name.

`DBNAMETOOLONG`

`Database name should not exceed 11 characters`

This error occurs when the database name specified for the connection request is greater than 11 characters.

Solution Use a shorter name for the database.

`DBINWRONGSTATE`

`Database "database" on host "host" is not in state INITED`

This error occurs when you try to disconnect from a database that is not connected or is not in the current context.

Solution Check if the application has connected to the database. If it has connected to the database, check the current context.

DBNOTINIT

database "*database*" on host "*host*" is not initialized

This error occurs when the MtCtxConnectDatabase function is called with no meta-schema having been previously defined on the database.

Solution Use the command mt_init_database to write the meta-schema in the database.

DBNOTOPENED

database "*database*" on host "*host*" is not opened

This error occurs when the MtCtxDisconnectDatabase.

Solution Check whether or not the database has been closed.

DEADLOCK

locks not acquired due to deadlock

This error occurs with a lock function exclusively. It indicates that no lock has been set, otherwise a deadlock would have been generated. The transaction is not aborted.

Solutions Repeat the operation again, until no error is returned (if the deadlock situation still exists, the error is systematically returned).

Either commit or abort the transaction (depending on the context) to escape the deadlock and restart the whole operation.

With the system engineer, find who has set locks on the objects.

DEADLOCKABORT

transaction aborted due to deadlock

This error can occur when an object is accessed (through a read or modification function). It indicates there has been a deadlock and the transaction has been aborted.

Solution Start a new transaction.

DIVISION_BY_ZERO

In a SQL statement, the evaluation of an expression leads to a division by zero.

EMPTYSTRING

attribute's value of object *object* should be a non empty string

ENDOFSTREAM

`end of stream - all values enumerated`

This error, which can occur when there is a stream enumeration (functions `MtCtxNextObject`, `MtCtxNextProperty` and `MtCtxNextTime`), indicates that the enumeration is over: all the elements of the stream have been returned.

Solution Close the stream.

EXCEEDSLIMIT

`Number of elements` *`numObjects`* `exceeds limit of` *`maxObjects`*

This error occurs when calling one of the following functions:

`MtCtx_CreateNumObjects`
`MtCtxCreateNumObjects`
`MtCtxLoadNumObjects`
`MtCtxLoadObjects`
`MtCtxLoadObjects`
`MtCtxLockNumObjects`
`MtCtxLockObjects`

when the number of objects specified is greater than the limit returned by `MtCtxGetConfigurationInfo` and when the type argument is set to the `MT_MAX_BUFFERED_OBJECTS`.

Solution: Call the function as many times as needed with `numObjects` less than or equal to the limit value.

FAILURE

This indicates an internal error that occurs during SQL statement resolution.

FROZENOBJECT

`object` `is frozen and cannot be modified`

This error occurs in `MT_DATA_MODIFICATION` connection mode exclusively, when trying to modify a schema object (any schema object is frozen in order to prevent modifications on them).

Solution Use DS to modify a schema object.

INCOMPCRITERIANUMBER

`Criteria number,` *`nb`*`, is not compatible with` *`criteria_order`*`'s value or` *`criteria_size`*`'s value`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxCommitTransaction`.

INCOMPCRITERIASIZE

Criteria size *size*, is not compatible with criteria type *type*

This error can occur in MT_DATA_DEFINITION connection mode with MtCtxCommitTransaction.

INCOMPOP

incompatible operation with type *type*

This error occurs when one of the two following functions is called:

MtCtxGetDimension
MtCtx_GetDimension

when the data (on which you are requesting dimensional information) is neither an array nor a list.

INCOMPRANKVALUE

rank and value are not compatible

This error occurs during a MtCtxSetValue or MtCtx_SetValue, when the rank specified by the user is incompatible with the data type (for example, a rank is declared as equal to 2 for a data of type MT_CHAR).

INCOMPTYPE

Type *type* incompatible type with make-entry-function

This error is returned by the "make-entry" method when the type of the value specified as argument is not one of the following: MT_DATE, MT_NULL, MT_SHORT, MT_INTEGER, MT_STRING, MT_TIMESTAMP, MT_BYTE.

"make-entry" is the default entry point creation function. It is called when an attribute value is modified in an object, when "make-entry" is the entry point creation function of the attribute.

Solutions Modify the type of the attribute.

INCOMPVERSION

database version is incompatible

This error occurs during the connection to a base, or when calling MtCtxConnectDatabase, if the base has been generated with a Matisse version older than the version currently running.

Solutions Use an older Matisse version in order to work with the desired base.

Upgrade the base, so that it can run with the desired Matisse version.

INDEXEXISTS

"*index_name*" is already the name of the index *index*.

This error can occur in MT_DATA_DEFINITION connection mode with MtCtxSetValue, MtCtx_SetValue.

INDEXEXPECTED

object is not an index

This error can occur with MtCtxOpenIndexEntriesStream, MtCtx_OpenIndexEntriesStream, MtCtxOpenIndexObjectsStream, MtCtx_OpenIndexObjectsStream, MtCtxGetIndexInfo.

INDEXEDATT

Attribute *attribute* is an index criterion or has a make entry function

This error can occur with MtCtxSetListElements, MtCtx_SetListElements.

INDEXINCREATION

Index *indexName* is being created

This error can occur with MtCtxGetIndexInfo, MtCtx_GetIndexInfo, MtCtxMGetIndexInfo, MtCtx_MGetIndexInfo.

INTERNALERROR

...

This error should never happen but it might occur after any call to a Matisse function.

Solution Contact your Matisse Software support center.

INVALARG

invalid number of arguments (*numArgs*)

This error can occur when any of the modification functions are called.

INVALATTMODIF1

you cannot reduce *attribute*'s value of *Attribute* which is an attribute of a class which has instances

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

This error indicates that the number of types that are allowed for a property's attribute has been modified during the transaction, but there is at least one class that has instances and that is associated with this attribute.

Because the type associated with the attribute for an instance may be one of the deleted types, it is impossible to restrict the number of types.

Solutions Re-specify the accepted type that has been removed.

Cancel the transaction.

INVALATTMODIF2

attribute's value in *Attribute* must be a list of different MtType elements

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

One of the types specified in the attribute is not a Matisse type (refer to where all Matisse data types are listed).

Solutions Check the possible values.

Cancel the transaction.

INVALATTMODIF6

You cannot modify *attribute* in *object*. *object* is a criterion of an index that has been created in a previous transaction.

This error can occur in MT_DATA_DEFINITION connection mode with MtCtxSetValue, MtCtx_SetValue.

INVALATTTYPE

attribute's value of object *object* has an invalid type

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

and when the type of the new value does not belong to the list of authorized types for the attribute.

Solution Cancel the transaction.

INVALBOOL

Not a valid MtBoolean value.

This error can occur when calling MtCtxSetValue, MtCtx_SetValue when the given value is different from MT_TRUE or MT_FALSE.

INVALCARDINALITY

<attribute cardinality>'s value of *relationship* is invalid

This error can occur in MT_DATA_DEFINITION connection mode exclusively, when calling MtCtxCommitTransaction.

The cardinality attached to the relationship is invalid; it must respect the following format:

- `(0 -1)`: the object can have any number of successors, or none;
- `(1 -1)`: at least one successor is required (no upper limit);
- `(1 6)`: the first number is the minimum number of successors. The second number is the maximum number of successors;
- `(1 1)`: it is the exact number of successors which must have the attribute and this attribute then becomes required;

By default, the cardinality is `(0 -1)`.

INVALCLASSMODIF4

`you cannot remove property from Class which has instances`

This error can occur in `MT_DATA_DEFINITION` connection mode exclusively, when calling `MtCtxCommitTransaction`.

You cannot remove an attribute or a relationship from a class that has instances, unless the attribute or the relationship is also inherited from a superclass or if the attribute or relationship has been destroyed.

Solutions Put back all the properties that have been removed during the transaction.

Cancel the transaction.

INVALCLASSMODIF5

`you cannot add relationship (whose minimal cardinality is minimum-cardinality) to Class which has instances`

This error can occur in `MT_DATA_DEFINITION` connection mode exclusively, when calling `MtCtxCommitTransaction`.

You cannot add a relationship which has a minimal cardinality to a class which has instances.

Solutions Set the minimal cardinality of the added relationship to 0.

Remove the relationship from the class.

Cancel the transaction.

INVALCLASSMODIF9

`You cannot add the superclass class to class, because the superclass has an index that has been created during a previous transaction.`

This error can occur in `MT_DATA_DEFINITION` connection mode calling one of the following functions:

`MtCtx_AddNumSuccessor`

MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtxAddNumSuccessor
MtCtxAddSuccessor
MtCtxAddSuccessors

INVALCLASSMODIF10

You cannot remove the superclass *class* from *class*, because the superclass has an index that has been created during a previous transaction.

This error can occur in MT_DATA_DEFINITION connection mode when calling one of the following functions:

MtCtxRemoveNumSuccessor
MtCtx_RemovedNumSuccessor
MtCtxRemoveSuccessors
MtCtx_RemoveSuccessors

INVALCLASSMODIF11

You cannot add the metaschema object *object* to the definition of a class.

This error can occur in MT_DATA_DEFINITION connection mode exclusively, when calling MtCtxCommitTransaction.

You are not allowed to add a meta-schema object to the definition of a class in the list of the relationships or in the list of the attributes of a class.

Solution Abort the transaction.

INVALCONNECTION

1204 in not a valid connection

This error occurs when the MtCtxConnectDatabase, MtCtxDisconnectDatabase, or MtCtxFreeContext functions are called with a wrong argument, may be a non allocated connection.

Solution Check if you have called MtCtxAllocateContext.

INVALCONNECTOPTION

345 is not a valid connection option

This error occurs when you try to set or get a connection option.

Solution Check the option you specified.

INVALCONNECTIONSTATE

Database "*database*" on host "*host*" is not in state INITED

This error occurs when you try to disconnect from a database that is not connected.

Solution Check if the application has connected to the database.

`INVALCREATION`

`invalid creation in runtime mode`

This error occurs in mode `MT_DATA_MODIFICATION`, when the functions `MtCtxCreateObject` or `MtCtx_CreateObject` are called, when the creation is related to a schema object.

Solution Use `MT_DATA_DEFINITION` connection mode to create a schema instance.

`INVALCRITERIACLASS`

`Class` *`class`* `does not have criteria` *`attribute`* `in its definition`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxCommitTransaction`.

`INVALCRITERIANB`

`nbOfCriteria` `should be less or equal to the index criteria number` *`x`*

This error occurs when calling `MtCtxOpenIndexEntriesStream`, `MtCtx_OpenIndexObjectsStream`, `MtCtxOpenIndexObjectsStream`, `MtCtx_OpenIndexEntriesStream`, with a number of criteria for the start and end values that exceeds the number of criteria defining the index.

`INVALCRITERIAORDER`

`attribute``'s value of index` *`index`* `is invalid`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxCommitTransaction`.

`INVALCRITERIASIZE`

`attribute``'s value of index` *`index`* `is invalid.`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxCommitTransaction`.

`INVALCRITERION`

`Attribute` *`attribute`* `cannot be a criterion. It is not of the right type`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxCommitTransaction`.

INVALDATAACCESSMODE

Invalid data access mode

This error occurs when calling the function MtCtxSetConnectionOption with an invalid value for DATA_ACCESS_MODE option.

Solutions The possible values for this option are MT_DATA_READONLY, MT_DATA_MODIFICATION, MT_DATA_DEFINITION.

INVALDIM

The dimension for *attribute's* value for object *object* must have a dimension between 1 and x

This error can occur when one of the following functions is called:

MtCtxSetValue
MtCtx_SetValue

The property value cannot be stored in the database because the dimension specified as an argument is either less than to 1, or greater than the highest possible value for a dimension (specified by the database constraints).

INVALDIRECTION

Invalid direction. A direction should be equal to MT_DIRECT or MT_REVERSE

This error can occur with MtCtxOpenIndexEntriesStream, MtCtx_OpenIndexEntriesStream, MtCtx_OpenIndexObjectsStream, MtCtxOpenIndexObjectsStream.

INVALID_ALIAS

An alias that was not previously defined is used in a SQL statement.

INVALID_CLASS

In a SQL statement, an identifier specified in a FROM clause does not correspond to a class or to a selection.

INVALID_DEFAULTVALUE

In a SQL statement, an incompatible type of value was specified as a default value for an attribute definition.

INVALID_EP_ATTRIBUTE

In a SQL statement, an argument of the keyword ENTRY_POINT is not correct.

INVALID_ESCAPE_CHAR

In a SQL statement, the escape character specified in a LIKE clause was incorrect.

INVALID_NUM_VALUE

In a SQL statement, anon-numeric value was specified where a numeric value was expected.

INVALID_IDENTIFIER

In a SQL statement, an identifier is invalid. If an identifier begins by a number, enclose the identifier in quotation marks " ".

INVALID_PROPERTY

In a SQL statement, aproperty specified in a statement is not associated with any class referenced in the command.

INVALID_REQUEST

The SQL statement evaluated was not recognized by the analyzer.

INVALID_SCALAR_VALUE

A scalar value is incorrect for a SQL expression that was analyzed.

INVALID_TIMEINTERVAL

In a SQL statement, an invalid format was used for a time interval constant.

INVALID_TIMESTAMP

In a SQL statement, an invalid format was used for a data or a timestamp constant.

INVALINDEXMODIF1

`You cannot modify attribute in index that has been created during a previous transaction.`

This error can occur in `MT_DATA_DEFINITION` connection mode with `MtCtxSetValue, MtCtx_SetValue`.

INVALINDEXMODIF2

`You cannot add a class to index that has been created during a previous transaction.`

This error can occur in `MT_DATA_DEFINITION` connection mode when calling one of the following functions:

`MtCtx_AddNumSuccessor`
`MtCtx_AddSuccessor`
`MtCtx_AddSuccessors`
`MtCtxAddNumSuccessor`
`MtCtxAddSuccessor`

MtCtxAddSuccessors

INVALINDEXMODIF3

You cannot remove a class from *index* that has been created during a previous transaction.

This error can occur in MT_DATA_DEFINITION connection mode when calling one of the following functions:

MtCtx_RemovedNumSuccessor
MtCtx_RemoveSuccessors
MtCtxRemoveNumSuccessor
MtCtxRemoveSuccessors

INVALINDEXMODIF4

You cannot add a criterion to *index* that has been created during a previous transaction.

This error can occur in MT_DATA_DEFINITION connection mode when calling one of the following functions:

MtCtx_AddNumSuccessor
MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtxAddNumSuccessor
MtCtxAddSuccessor
MtCtxAddSuccessors

INVALINDEXMODIF5

You cannot remove a criterion from *index* that has been created during a previous transaction.

This error can occur in MT_DATA_DEFINITION connection mode when calling one of the following functions:

MtCtx_RemovedNumSuccessor
MtCtx_RemoveSuccessors
MtCtxRemoveNumSuccessor
MtCtxRemoveSuccessors

INVALINTERVAL

Start value must be less or equal to end value

This error occurs with MtCtxOpenIndexEntriesStream, MtCtx_OpenIndexEntriesStream, MtCtx_OpenIndexObjectsStream, MtCtxOpenIndexObjectsStream. The comparison takes into account the ordering, that is, the way the Oids have been indexed.

INVALIREL

object's class is not a valid successor of *relationship*

This error occurs for the following functions:

```
MtCtxGetPredecessors
MT_GetPredecessors
MtCtxMGetPredecessors
MT_MGetPredecessors
MtCtxOpenPredecessorsStream
MtCtx_OpenPredecessorsStream
```

This error is returned when the object specified as an argument is not a possible successor for the relationship *relationship*. The classes that are allowed are specified in the property "successors" of the relationship *relationship* and the *successors*'s class is not part of this list.

INVALLOCK

invalid lock. A lock should be equal MT_READ or MT_WRITE

This error can occur when calling one of the following functions:

```
MtCtxLockNumObjects
MtCtxLockObjects
MtCtxLockObjectsFromEntryPoint
MtCtx_LockObjectsFromEntryPoint
```

The only authorized locks are MT_READ and MT_WRITE.

INVALLISTOFFSET

The first element offset exceeds the list total number of elements

This error can occur when calling one of the following functions:

```
MtCtxGetListElements
MtCtx_GetListElements
MtCtxSetListElements
MtCtx_SetListElements
```

INVALLISTSIZE

The list size is limited to LONG_MAX elements

This error can occur when calling MtCtxSetListElements or MtCtx_SetListElements.

INVALMAPFUNCTION

you cannot use *function* on *stream*

This error can occur when calling MtCtxNextObject or MtCtxNextProperty

The function `MtCtxNextProperty` can be used only with an `object attribute stream`, an `object relationship stream` or an `object inverse relationship stream`.

The function `MtCtxNextObject` can be used only with a `class stream`, an `entrypoint stream`, a `relationship stream` or an `inverse relationship stream`.

`INVALMODIF`

`you cannot modify a terminal instance and a meta-schema instance within the same transaction`

This error occurs in `MT_DATA_DEFINITION` connection mode exclusively.

It occurs when a terminal instance and a schema instance are modified within the same transaction.

It can thus occur when calling one of the following functions:

```
MtCtx_AddNumSuccessors
MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtx_CreateObject
MtCtx_RemoveAllSuccessors
MtCtx_RemoveNumSuccessors
MtCtx_RemoveSuccessors
MtCtx_SetValue
MtCtxAddNumSuccessors
MtCtxAddSuccessor
MtCtxAddSuccessors
MtCtxCreateObject
MtCtxRemoveAllSuccessors
MtCtxRemoveNumSuccessors
MtCtxRemoveObject
MtCtxRemoveSuccessors
MtCtxRemoveValue
MtCtxSetValue
```

Solution In the same transaction, do not perform operations both on the schema and on terminal instances.

`INVALNAMESIZE`

`*attribute*'s value in *object* must be a string between 1 and 256 characters`

TK does this still exist now that check functions are gone?

This error can occur in `MT_DATA_DEFINITION` connection mode exclusively.

This error occurs at transaction commit.

INVALNB

```
number of elements %d should be positive
```

This error occurs when calling a function that possesses either an array of elements in input, or variable arguments (such as `MtCtxAddSuccessors` or `MtCtxRemoveNumSuccessors`), when the argument that indicates the number of elements specified is negative or null.

INVALOP

```
invalid operation. Function function is not allowed in this
context (state)
```

This error can occur when calling any Matisse function (except reading functions), when the function cannot be called in the current context.

`INVALOP` introduces the concept of *`state`*. *`state`* corresponding to a Matisse state once the error has been generated. The following table lists all possible states:

State	Description
`MT_COMMIT_WAIT`	A transaction is in course but any modification is forbidden following a halt in writing in `MtCtxCommitTransaction`.
`MT_CONNECTED`	A database connection has been performed, a database is selected, but no transaction or access in version mode has been performed.
`MTSTREAMMODIF`	A stream opening has been performed in a context where some modifications are authorized.
`MTSSTREAMMODIFCONTEXT`	A stream opening has been performed in a context where some modifications are authorized.
`MTSTREAMREAD`	A stream opening has been performed in a context where no writing can be performed.
`MTSTREAMVERSIONMODE`	A stream opening has been performed in a version access context.
`MT_TRANSACTION`	A transaction has been initiated.
`MT_VERSION`	A version access is initiated. Any read access on database objects can be performed.

INVALPASSWD

```
Invalid user password
```

This error occurs when calling the functions `MtCtxConnectDatabase` or `MtCtxSetOwnPassword`, when the specified password is invalid.

Solutions With `MtCtxConnectDatabase`, check if the given password is not NULL.

With MtCtxSetOwnPassword, check that the old password is not NULL.

INVALPASSWDLEN

Invalid password length

This error occurs when calling the functions MtCtxConnectDatabase or MtCtxSetOwnPassword, when the specified password is invalid.

Solutions

With MtCtxConnectDatabase, check if the given password is not too long or if the user name is NULL; if it is, the password must be NULL.

With MtCtxSetOwnPassword, check the old and new password lengths, they must less or equal to MT_USER_PASSWORD_MAX_LEN.

INVALPRIO

priority should be between 0 and *x*

This error occurs when calling the functions MtCtxSetConnectionOption for the option MT_SERVER_EXECUTION_PRIORITY or MtCtxStartTransaction with an invalid value.

Solutions

With MtCtxConnectDatabase, check that the priority is between MT_MIN_SERVER_EXECUTION_PRIORITY and MT_MAX_SERVER_EXECUTION_PRIORITY.

With MtCtxStartTransaction, check that the priority is between MT_MIN_TRAN_PRIORITY and MT_MAX_TRAN_PRIORITY.

INVALPROPREMOVE

you cannot remove *successor* from *object*. *successor* belongs to the meta-chema

This error occurs in MT_DATA_DEFINITION connection mode exclusively.

It occurs when calling one of the following functions:

MtCtx_RemoveAllSuccessors
MtCtx_RemoveNumSuccessors
MtCtx_RemoveSuccessors
MtCtxRemoveAllSuccessors
MtCtxRemoveNumSuccessors
MtCtxRemoveSuccessors

when the modified object is a meta-schema object, and when the successor to be removed belongs to the meta-schema. The transaction is aborted. You can remove from a meta-schema object only attributes and relationships which have previously been added according to the modification constraints.

Solutions

Add the properties that have been removed;

INVALRANK

rank should be between 0 and *x*

This error occurs when calling the functions MtCtxSetValue or MtCtx_SetValue, when the specified rank is either negative, or greater than the limit indicated in the database constraints.

INVALRANKINDEX

rankIndex should be between 0 and *rank-1*

This error occurs when calling the functions MtCtxGetDimension or MtCtx_GetDimension, when the specified dimension is either negative, or greater than the maximum dimension allowed in the database constraints, or if it is greater than the rank of the value stored in the base.

INVALREL

relationship is invalid

This error occurs in MT_DATA_DEFINITION connection mode exclusively, and when calling one of the following functions:

MtCtx_AddNumSuccessors
MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtx_GetPredecessors
MtCtx_GetSuccessors
MtCtx_MGetPredecessors
MtCtx_MGetSuccessors
MtCtx_OpenPredecessorsStream
MtCtx_OpenSuccessorsStream
MtCtx_RemoveAllSuccessors
MtCtx_RemoveNumSuccessors
MtCtx_RemoveSuccessors
MtCtxAddNumSuccessors
MtCtxAddSuccessor
MtCtxAddSuccessors
MtCtxGetPredecessors
MtCtxGetSuccessors
MtCtxMGetPredecessors
MtCtxMGetSuccessors
MtCtxOpenPredecessorsStream
MtCtxOpenSuccessorsStream
MtCtxRemoveAllSuccessors
MtCtxRemoveNumSuccessors
MtCtxRemoveSuccessors

This error occurs when a relationship with no inverse relationship is specified as an argument. Only a valid relationship (with its inverse relationship) can be specified as an argument for the identified functions.

In MT_DATA_MODIFICATION connection mode, relationships are necessarily valid: this error never occurs in this mode.

Solution Add the inverse relationship to the relationship before calling the function.

INVALRELDELETE

you have deleted a relationship without deleting its inverse relationship.

This error occurs in MT_DATA_DEFINITION connection mode exclusively.

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

When you delete a relationship, you must also delete its inverse relationship.

Solutions Delete the relationship's inverse relationship. Abort the transaction.

INVALRELMODIF1

you cannot modify a *property* of a *relationship* that specifies an integrity constraint.

This error occurs in MT_DATA_DEFINITION connection mode exclusively.

It occurs when calling MtCtxCommitTransaction.

This error can occur whenever you try to modify any of a relationship's properties that specify an integrity constraint. Properties that specify an integrity constraint are the attribute MtCardinality, the relationship MtCtxSuccessors and the attribute MtCtxRelationshipCheckFunction.

Solution Cancel the transaction.

INVALRELMODIF2

you cannot remove *Class* (which has instances) from *successors* in *relationship*

This error occurs in MT_DATA_DEFINITION connection mode exclusively.

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

You cannot remove a class from the list of valid successors of a relationship if the class (or one of its subclasses) has instances.

Solutions Put back the classes that have been removed during the transaction.

Cancel the transaction.

INVALRELMODIF3

Class has a relationship which is the inverse relationship of *relationship*, but *Class* is not a valid successor (directly or indirectly) of *relationship*

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

if the definition of a class specifies that the class has a relationship, but the class is not a valid successor for the inverse relationship of the relationship (taking inheritance into account).

Solutions

Remove the relationship from the class.

Add the class to the list of valid successors for the inverse relationship of the added relationship.

Cancel the transaction.

INVALRELMODIF4

relationship cannot be a relationship of *Class* which is not a valid successor of its inverse relationship

This error occurs in MT_DATA_DEFINITION connection mode exclusively, when calling:

MtCtxCommitTransaction

You cannot add a relationship in the class definition if the class is not a valid successor for the inverse relationship of the added relationship (taking inheritance into account).

Solutions

Remove the relationship from the class.

Add the class to the list of valid successors for the inverse relationship of the added relationship.

Cancel the transaction.

INVALRELMODIF5

you cannot add one class to *relationship* in *object* which is a universal relationship and is attached to a class having instances.

This error can occur in MT_DATA_DEFINITION connection mode exclusively, when calling MtCtxCommitTransaction. The transaction is aborted.

When a relationship is attached to at least one class which has instances, and when it is a universal relationship (the relationship MtSuccessors has no value, so all the database classes are considered as possible successors for the relationship), you cannot add successors to this universal relationship through

the relationship `MtSuccessors`. This would reduce the possible successors for the relationship and some instances specified with this relationship could become invalid.

Solutions Remove the instances from the class to which the relationship is attached.

Remove the classes that have been added to the relationship via the relationship `MtSuccessors`.

`INVALSTREAM`

`stream` `is not a valid stream for the selected database`

This error occurs when calling one of the following functions:

```
MtCtxCloseStream
MtCtxNextIndexEntry
MtCtxNextObject
MtCtxNextProperty
```

when the stream specified as an argument does not correspond to a valid open stream (the stream may have been opened in another connection).

`INVALSTRINGSIZE`

`entry point's length should be between 1 and 32`

This error can occur when calling any function with an entry-point specified as an argument (more specifically, all the functions whose argument is the string associated with a schema object).

`INVALSUCCESSOR`

`successor``'s class is not a valid successor of` *`relationship`*

This error can occur when calling `MtCtxCommitTransaction`.

This error is returned when the successor of an object through a relationship is not of an appropriate class. The classes that are allowed are specified in the property `MtSuccessors` of the relationship *`relationship`* and the *`successors`*'s class is not part of this list.

`INVALSUCCREMOVE`

`relationship` `cannot become a universal relationship because it is a metaschema relationship.`

This error can occur in `MT_DATA_DEFINITION` connection mode only when calling `MtCtxCommitTransaction`.

A meta-schema relationship cannot become universal. This means by definition that you cannot remove all the successors of a meta-schema relationship.

Solutions Abort the transaction.

INVALSUCCSNB

`invalid number of successors x for relationship relationship`

This error can occur when calling `MtCtxCommitTransaction` when the number of successors of the relationship in the object does not match the cardinality.

INVALSUPCLASS

`class Class cannot be a Superclass of class Class, otherwise a cycle in the inheritance is created`

This error occurs in `MT_DATA_DEFINITION` connection mode exclusively.

It occurs when calling one of the following functions:

`MtCtx_AddNumSuccessors`
`MtCtx_AddSuccessor`
`MtCtx_AddSuccessors`
`MtCtxAddNumSuccessors`
`MtCtxAddSuccessor`
`MtCtxAddSuccessors`

You cannot add a superclass to a class if the superclass equals the class or is already one of the class's subclasses. The transaction is aborted.

Solutions Remove the class from the list of the superclasses of the modified class.

INVALTIMESTAMP

`Not a valid MtTimestamp value`

This error occurs when calling the `MtCtxSetValue` or `MtCtx_SetValue` functions, when the specified time is not valid (i.e. one of its fields has an invalid value).

INVALTIMEINTERVAL

`Not a valid MtInterval value`

This error occurs when calling the `MtCtxSetValue` or `MtCtx_SetValue` functions, when the specified time interval is not valid (i.e. one of its fields has an invalid value).

INVALTYPE

`x is not a valid Matisse type`

This error occurs when calling the `MtCtxSetValue` or `MtCtx_SetValue` functions, when the specified type is not valid (i.e. does not belong to the `enum MtType`).

INVALUSERNAMELEN

`Invalid User name length`

This error occurs when calling the `MtCtxConnectDatabase` function when the specified user name is too long.

Solutions Check the user name length, it must be less or equal to `MT_USER_NAME_MAX_LENGTH`.

`INVALWAITTIME`

`wait` `must be greater or equal to -1`

This error occurs when calling the `MtCtxSetConnectionOption` function for the option `MT_LOCK_WAIT_TIME` with a value less than -1.

`INVALWHERE`

`Invalid argument where. Should be equal to MT_FIRST, MT_AFTER or MT_APPEND.`

This error occurs when calling the `MtCtxAddSuccessor` function with the argument where different from `MT_FIRST`, `MT_AFTER` and `MT_APPEND`.

`INVTRANSPORT`

`Attempted to connect with an invalid transport`

This error occurs, at connect, when an incompatibility exists between specified transport and other parameters. For example, if you try to connect to a database localized on a different host with a local transport (i.e. same host), this error is returned.

`MEMORYFAULT`

`No more memory available for operation`

This error occurs when there is no memory left on your client machine.

Solution: Free memory. You may want to free cache memory using `MtCtxFreeObjects`.

`METASCHEMAOBJECT`

`you cannot modify` *`object`* `which is a meta-schema object`

This error occurs in `MT_DATA_DEFINITION` connection mode exclusively.

It occurs when calling one of the following functions:

```
MtCtx_AddNumSuccessors
MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtx_RemoveAllSuccessors
MtCtx_RemoveNumSuccessors
MtCtx_RemoveSuccessors
MtCtx_RemoveValue
MtCtx_SetValue
MtCtxAddNumSuccessors
```

MtCtxAddSuccessor
MtCtxAddSuccessors
MtCtxRemoveAllSuccessors
MtCtxRemoveNumSuccessors
MtCtxRemoveObject
MtCtxRemoveSuccessors
MtCtxRemoveValue
MtCtxSetValue

You cannot delete an object of the original meta-schema, modify an attribute of an object of the original meta-schema, or add a superclass to a class of the original meta-schema.

Solutions If the problem arises from the addition of a class, remove the class.

`NESTEDVERSION`

`attempt to start a version access while another is still started`

This error occurs when calling the functions `MtCtxStartVersionAccess`, while a version access is in progress.

Solution End the version access using `MtCtxEndVersionAccess`.

`NESTEDTRANS`

`attempt to start a transaction while another is still opened`

This error occurs when calling the function `MtCtxStartTransaction`, while another transaction is already opened.

Solution End the current transaction using `MtCtxCommitTransaction` or `MtCtxAbortTransaction`.

`NOFREETOKEN`

`All the tokens on the host` *`host`* `are being used by different connections`

This error occurs when calling the function `MtCtxConnectDatabase`.

Solution Wait until one or more connections are available.

`NONULLVALUE`

`Attribute_name` `for object` *`OID`* `requires a non null value.`

This error occurs when attempting to commit an object containing a non-nullable attribute for which no value has been specified.

Solution Specify a value or make the attribute nullable, as appropriate .

`NOPMADDR`

`Unable to get Port Monitor address.`

This error occurs when the Port Monitor address can't be retrieved.

Solution Depending on the host, check NIS or environment variables `MTS_PORTMON_ADDR` and `MTS_PORTMON_NAME` or the file `/etc/services`.

`NOSECURITY`

```
Invalid operation: access control not used for this database
```

This error occurs when calling `MtCtxSetOwnPassword`.

`NOSCANNABLEINDEX`

```
You cannot scann index index. This index has been created
during the current transaction.
```

This error occurs in `MT_DATA_DEFINITION` connection mode only with `MtCtxOpenIndexEntriesStream`, `MtCtx_OpenIndexEntriesStream`, `MtCtx_OpenIndexObjectsStream`, `MtCtxOpenIndexObjectsStream`,.

`NOSUCCESSORS`

```
object object has no successors for the relationship
relationship
```

This error occurs when calling one of the following functions:

```
MtCtxRemoveAllSuccessors
MtCtx_RemoveAllSuccessors
```

when the object has no successor via the relationship specified as an argument.

`NOSUCHATT`

```
attribute "attribute_name" is undefined
```

This error occurs when calling one of the following functions:

```
MtCtxGetAttribute
MtCtxGetDimension
MtCtxGetObjectsFromEntryPoint
MtCtxGetValue
MtCtxLockObjectsFromEntryPoint
MtCtxMGetObjectsFromEntryPoint
MtCtxMGetValue
MtCtxOpenEntryPointStream
MtCtxRemoveValue
MtCtxSetValue
```

when the specified string is not associated with an attribute.

`NOSUCHCLASS`

```
class "class_name" is undefined
```

This error occurs when calling one of the following function :

```
MtCtxCreateObject
```

MtCtxGetAllAttributes
MtCtxGetAllInverseRelationships
MtCtxGetAllRelationships
MtCtxGetAllSubclasses
MtCtxGetAllSuperclasses
MtCtxGetClass
MtCtxGetInstancesNumber
MtCtxGetObjectsFromEntryPoint
MtCtxIsInstanceOf
MtCtxLockObjectsFromEntryPoint
MtCtxMGetAllAttributes
MtCtxMGetAllInverseRelationships
MtCtxMGetAllRelationships
MtCtxMGetAllSubclasses
MtCtxMGetAllSuperclasses
MtCtxMGetObjectsFromEntryPoint
MtCtxOpenEntryPointStream
MtCtxOpenInstancesStream

when the class *class_name* is not a defined class.

NOSUCHCLASSATT

invalid attribute *attribute_property* for class *Class*

This error occurs when calling one of the following functions:

MtCtx_GetDimension
MtCtx_GetObjectsFromEntryPoint
MtCtx_GetValue
MtCtx_LockObjectsFromEntryPoint
MtCtx_MGetObjectsFromEntryPoint
MtCtx_MGetValue
MtCtx_RemoveValue
MtCtx_SetValue
MtCtxGetDimension
MtCtxGetObjectsFromEntryPoint
MtCtxGetValue
MtCtxLockObjectsFromEntryPoint
MtCtxMGetObjectsFromEntryPoint
MtCtxMGetValue
MtCtxRemoveValue
MtCtxSetValue

This error occurs when the attribute *Attribute* is not defined for the class *Class*.

NOSUCHCLASSINDEX

Invalid index *index* for class *class*

This error occurs when calling `MtCtxOpenIndexEntriesStream`, `MtCtx_OpenIndexEntriesStream`, `MtCtx_OpenIndexObjectsStream`, `MtCtxOpenIndexObjectsStream`. The index *index* is not defined for the class *class*.

NOSUCHCLASSREL

invalid relationship *relationship* for class *Class*

This error occurs when calling one of the following functions:

`MtCtx_AddNumSuccessors`
`MtCtx_AddSuccessor`
`MtCtx_AddSuccessors`
`MtCtx_GetAddedSuccessors`
`MtCtx_GetRemovedSuccessors`
`MtCtx_GetSuccessors`
`MtCtx_MGetAddedSuccessors`
`MtCtx_MGetSuccessors`
`MtCtx_OpenSuccessorsStream`
`MtCtx_RemoveAllSuccessors`
`MtCtx_RemoveNumSuccessors`
`MtCtx_RemoveSuccessors`
`MtCtxAddNumSuccessors`
`MtCtxAddSuccessor`
`MtCtxAddSuccessors`
`MtCtxGetAddedSuccessors`
`MtCtxGetRemovedSuccessors`
`MtCtxGetSuccessors`
`MtCtxM_GetRemovedSuccessors`
`MtCtxMGetAddedSuccessors`
`MtCtxMGetRemovedSuccessors`
`MtCtxMGetSuccessors`
`MtCtxOpenSuccessorsStream`
`MtCtxRemoveAllSuccessors`
`MtCtxRemoveNumSuccessors`
`MtCtxRemoveSuccessors`

This error occurs when the relationship *relationship* is not defined for the class *Class*.

NOSUCHDB

database "*database*" not found on host "*host*"

The database does not exist on the host *host*.

Solution Check the name of your database and of your host.

NOSUCHHOST

Host "*host*" not found

This error occurs when calling the function `MtCtxConnectDatabase`. It is impossible to find the server *host*.

Solution Check the name of your host. Ask the system engineer if the server is running.

NOSUCHINDEX

Index "*index_name*" is undefined

This error can occur with `MtCtxOpenIndexEntriesStream`, `MtCtx_OpenIndexEntriesStream`, `MtCtxGetIndexInfo`, `MtCtxMGetIndexInfo`, `MtCtx_OpenIndexObjectsStream`, `MtCtxOpenIndexObjectsStream`.

NOSUCHREL

relationship "*relationship_name*" is undefined

This error occurs when calling a function with a string that identifies a relationship as an argument or when calling one of the following functions:

`MtCtxAddNumSuccessors`
`MtCtxAddSuccessor`
`MtCtxAddSuccessors`
`MtCtxGetAddedSuccessors`
`MtCtxGetPredecessors`
`MtCtxGetRelationship`
`MtCtxGetRemovedSuccessors`
`MtCtxGetSuccessors`
`MtCtxMGetAddedSuccessors`
`MtCtxMGetPredecessors`
`MtCtxMGetRemovedSuccessors`
`MtCtxMGetSuccessors`
`MtCtxOpenPredecessorsStream`
`MtCtxOpenSuccessorsStream`
`MtCtxRemoveAllSuccessors`
`MtCtxRemoveNumSuccessors`
`MtCtxRemoveSuccessors`

when the specified string is not associated with a relationship.

NOSUCHSELECTION

The specified SQL selection does not exist.

NOSUCHSUCC

successor *successor* does not exist

This error occurs when calling one of the following functions:

```
MtCtxAddSuccessor
MtCtx_AddSuccessor
MtCtxRemoveNumSuccessors
MtCtx_RemoveNumSuccessors
MtCtxRemoveSuccessors
MtCtx_RemoveSuccessors
```

For the remove successor functions, one of the successors to be deleted does not exist in the object. No deletion has been initiated.

For the adding successor functions, the successor specified behind the `MT_AFTER` argument does not exist in the object. No addition has been performed.

`NOSUCHVERSION`

```
version "versionname" is undefined
```

This error can occur in the function `MtCtxStartVersionAccess`.

This error occurs when there is an attempt to position to the specified version at the time that corresponds to *`versionname`* but no `MtCtxCommitTransaction` has been performed with *`versionname`* as a prefix. It is therefore impossible to position to the time *`timename`*.

`NOTENOUGHSPACE`

```
not enough space to copy data. num bytes needed
```

This error occurs when calling one of the following functions:

```
MtCtxGetValue
MtCtx_GetValue
MtCtxNextTime
```

Matisse attempts to copy the data in the space allocated by the user. The pointer and the size are specified in the arguments. Matisse has insufficient space to copy the data.

Solution — Increase the size so as to make it at least equal to *`num`* bytes.

`NOTRANORVERSION`

```
attempt to access objects without a transaction or version
access
```

This error can occur in any function where an access to an object is performed without previously opening a transaction or without being in version mode.

`NOTRANS`

```
transaction not opened
```

This error occurs during a modification function. No transaction is opened.

NOVALUE

attribute *Attribute* has no value in object *object*

This error occurs when calling one of the following functions:

MtCtxRemoveValue

MtCtx_RemoveValue

when the attribute has no value in the object.

NOVERSIONACCESS

no version access

This error occurs when calling the function MtCtxEndVersionAccess. You can stop the version mode if no MtCtxStartVersionAccess has been started.

NULLPOINTER

null pointer

A null pointer is specified as an argument, and this pointer should not be null.

OBJECTDELETED

object_identifier has been deleted

This error can occur in any function where an object Oid is specified. This means that the object no longer exists. It has been deleted with the function MtCtxRemoveObject within the current transaction.

OBJECTNOTFOUND

object_identifier not found

This error can occur in any function where an object Oid is specified as an argument. This means that the object does not exist.

OPDENIED

Operation denied: insufficient privileges or wrong password

This error can occur when calling MtCtxConnectDatabase. There are three possible reasons for this:
- the user is not authorized,
- the specified password is wrong,
- the user has insufficient privileges for the data access mode specified.

PMCONFAILED

Unable to connect to Port Monitor

This error occurs at connection, when you are unable to connect to the Port Monitor.

Solution Check that Port Monitor is running

PROPERTYEXISTS

"*property_name*" is already the name of the property *property*

This error occurs in MT_DATA_DEFINITION connection mode exclusively.

It occurs when calling one of the following functions:

```
MtCtxSetValue
MtCtx_SetValue
```

You cannot set as the external name for a property, a name that is already used for an existing property. The transaction is aborted.

Solutions

Set as the external name of the property, a name that does not already exist.

RELEXPECTED

object is not a relationship

This error occurs either when calling a function defined with a relationship identifier as an argument, or when calling one of the following functions:

```
MtCtx_AddNumSuccessors
MtCtx_AddSuccessor
MtCtx_AddSuccessors
MtCtx_GetAddedSuccessors
MtCtx_GetPredecessors
MtCtx_GetRemovedSuccessors
MtCtx_GetSuccessors
MtCtx_MGetAddedSuccessors
MtCtx_MGetPredecessors
MtCtx_MGetRemovedSuccessors
MtCtx_MGetSuccessors
MtCtx_OpenPredecessorsStream
MtCtx_OpenSuccessorsStream
MtCtx_RemoveAllSuccessors
MtCtx_RemoveNumSuccessors
MtCtx_RemoveSuccessors
```

when the specified identifier is not a relationship.

SCHEMAWITHDAEMONS

class or attribute has before or/and after modification daemons(s)

It occurs when calling one of the following functions:

```
MtCtxSetListElements
MtCtx_SetListElements
```

SELECTIONSTREAMOPEN

There are some streams open associated with the SQL selection. Close these streams before calling the function.

STMT_TOO_COMPLEX

SQL statement too complex.

STREAMCLOSED

```
stream opened by application closed by DBA tool
```

This error occurs when a stream opened by the application is inadvertently aborted by server administration utilities. This message can be returned by any of the following functions:

```
MtCtxCloseStream
MtCtxConnectDatabase
MtCtxNextObject
MtCtxStartVersionAccess
MtCtxStartTransaction
```

Solution Make sure that the system administrator and/or other users do not abort streams opened by the application.

SUCCESS

Status returned upon successful execution of a SQL statement.

SYNTAX_ERROR

Miscellaneous SQL syntax error: incorrect use of parentheses, invalid expression, etc.

SYSTEMERROR

```
system error
```

This error should never happen but it might occur after a call to a Matisse function.

Solution Contact your Matisse Software support center.

TOO_MANY_VALUES

In a SQL statement, too many values are specified in INSERT statement's VALUE clause.

TOO_FEW_VALUES

In a SQL statement, too few values are specified in INSERT statement's VALUE clause.

TRANABORTED

transaction opened by application aborted by server admin

This error occurs when a transaction opened by the application is inadvertently aborted by server administration utilities. This message can be returned by any of the following functions:

MtCtx_OpenIndexObjectsStream
MtCtxAbortTransaction
MtCtxAddNumSuccessors
MtCtxAddSuccessor
MtCtxAddSuccessors
MtCtxCommitTransaction
MtCtxCreateNumObjects
MtCtxCreateObject
MtCtxGetAllAttributes
MtCtxGetAllInverseRelationships
MtCtxGetAllRelationships
MtCtxGetAllSubclasses
MtCtxGetAllSuperclasses
MtCtxGetAttribute
MtCtxGetClass
MtCtxGetDimension
MtCtxGetInstancesNumber
MtCtxGetObjectClass
MtCtxGetObjectsFromEntryPoint
MtCtxGetPredecessors
MtCtxGetRelationship
MtCtxGetSuccessors
MtCtxGetValue
MtCtxIsInstanceOf
MtCtxLoadNumObjects
MtCtxLoadObjects
MtCtxLockNumObjects
MtCtxLockObjects
MtCtxLockObjectsFromEntryPoint
MtCtxNextObject
MtCtxObjectSize
MtCtxOpenAttributesStream
MtCtxOpenEntryPointStream
MtCtxOpenIndexEntriesStream
MtCtxOpenIndexObjectsStream
MtCtxOpenInstancesStream
MtCtxOpenInverseRelationshipsStream
MtCtxOpenPredecessorsStream
MtCtxOpenRelationshipsStream

```
MtCtxOpenSuccessorsStream
MtCtxPrint
MtCtxRemoveAllSuccessors
MtCtxRemoveNumSuccessors
MtCtxRemoveObject
MtCtxRemoveSuccessor
MtCtxRemoveSuccessors
MtCtxRemoveValue
MtCtxSetValue
```

TRANSDISABLED

```
Transaction Processing has been disabled
```

This error can occur when calling `MtCtxStartTransaction`, or when calling `MtCtxConnectDatabase` in `MT_DATA_READONLY` mode. This error can also be returned when using the commands `mt_init_database`. This error indicates that transaction processing has been disabled by the administrator.

Solution Check with the database administrator to see if transaction processing can be enabled. Normally, transaction processing can be enabled with the transaction processing option in the DBA Tool.

TRANSNOTALLOWED

```
You are connected in a version only mode
```

This error can occur when calling `MtCtxStartTransaction`.

It occurs when the user attempts to open a transaction once a database has been opened with `MT_DATA_READONLY` access mode.

Solution Call `MtCtxSetConnectionOption` with `MT_DATA_MODIFICATION` or `MT_DATA_DEFINITION` mode.

TRANSOPENED

```
attempt to set a time inside a transaction
```

This error occurs when calling `MtCtxStartVersionAccess`. This error indicates that you are in transaction mode and cannot make an access in version mode.

Solution Commit or abort your transaction to exit.

TYPEMISMATCH

```
The attribute's value type does not correspond to the type
argument
```

This error occurs when calling one of the following functions:

```
MtCtxGetListElements, MtCtx_GetListElements
MtCtxSetListElements, MtCtx_SetListElements
```

TYPENOTALLOWED

`The specified type is not allowed for the current function`

This error occurs when calling one of the following functions:

```
MtCtxGetListElements, MtCtx_GetListElements
MtCtxSetListElements, MtCtx_SetListElements
```

UNEXPECTEDDUPLICATES

`successor` `is referenced twice unexpectedly`

This error occurs when adding or removing the same successor to an object multiple times.

This error can occur when calling one of the following functions:

```
MtCtxAddSuccessors
MtCtxRemoveSuccessors
```

UNLOADABLEOBJECT

`object` `cannot be unloaded`

This error can occur when calling one of the following functions:

```
MtCtxFreeNumObjects
MtCtxFreeObjects
```

This error indicates that one of the objects specified as an argument is a schema object, or has been modified during the transaction, or is an object on which a

```
relationship stream
inverse relationship stream
object attribute stream
object relationship stream
object inverse relationship stream
```

has been opened.

CAUTION: `MtCtxFreeObjects` and `MtCtxFreeNumObjects` are atomic functions : either all the objects specified as arguments are retrieved, or none are.

VERSIONMODE

`attempt to start a transaction in version mode`

This error occurs when calling `MtCtxStartTransaction` after you have previously called `MtCtxStartVersionAccess`. This error indicates that you are then in version mode and cannot perform modifications. As a result, you cannot open a transaction.

Solution Use the function `MtCtxEndVersionAccess` to exit the transaction mode.

WAITTIME

```
lock not obtained due to short wait-time
```

This error can occur during a read, write or lock operation. When trying to obtain a read or write lock on the server, you are positioned in a queue. Your position in this queue depends on the number of seconds specified in the functions `MtCtxSetConnectionOption`. If the lock is not obtained after the time has elapsed, this error is returned.

If a deadlock is detected however, the `DEADLOCKABORT` error is returned.

WRITEWAITTIME

```
write lock not obtained due to short wait-time
```

This error can occur if the wait time, set with `MtCtxConnect` or `MtSetWaitTime`, is different from `MT_WAIT_FOREVER`. If write locks cannot be acquired while the objects are being written, the `MATISSE_WRITEWAITTIME` error occurs. Even though the transaction is neither committed nor aborted, no other modifications are allowed. All modification functions will return `MATISSE_INVALOP`.

If the wait time is `MT_WAIT_FOREVER`, a deadlock is detected and the `DEADLOCKABORT` error is returned.

Index

L

M

N

O

P

R

S

T