C++ Software transactional Memory

Generated by Doxygen 1.8.11

Sun Mar 11 2018 15:50:35

Contents

1 C++ Software Transactional Memory

File: TM.h Author: Zoltan Fuzesi C00197361, IT Carlow, Software Engineering,

Supervisor : Joe Kehoe,

C++ Software Transactional Memory,

Created on December 18, 2017, 2:09 PM Transaction Manager class fields and methods declarations

1.1 Object Based Software Transactional Memory.

OSTM is a polymorphic solution to store and manage shared memory spaces within c++ programming context. You can store and managed any kind of object in transactional environment as a shared and protected memory space, if your class inherited from the OSTM base class, and follows the required steps.

1.1.1 Brief. Download the zip file from the provided (Windows, Linux, MAC OSX)link in the web-site,

that contains the libostm.so, TM.h, TX.h, OSTM.h files.Unzip the archive file to the desired destination possibly where you program is stored. Copy the library (Shared, Static) to the destination directory. Implement the inheritance from the base class. Create an executable, and run the application.

1.1.2 Step 1: Download the archive file.

1.1.3 Step 2: Unzip in to the target destination.

1.1.4 Step 3: Copy the shared library (libostm.so) to the operating system folder where the other shared library are stored.

It will be different destination folder on different platforms. (Linux, Windows, Mac OS) More Information

1.1.5 Step 4: Achieve the required class hierarchy between the OSTM library and your own class structure.

Details and instruction of class hierarchy requirements can be found on the web-site. www.serversite.info/ostm

1.1.6 Step 5: Create an executable file as you linking together the library, the ∗.h files with your own files.

1.1.7 Step 6: Now your application use transactional environment, that guarantees the consistency between object transactions.

1.1.8 Step 7: Run the application.

2 README

Usage of the STM library on Linux.
In order to use the O_STM library with any C++ application, it need to be placed to the operation system /usr/lib directory.

1. Copy lib_o_stm.so file to /usr/lib: sudo cp lib_o_stm.so /usr/lib
2. Include the TM.h TX.h and the OSTM.h files in your application.
3. Create Makefile :

   **Makefile.mk Documentation
**

```
EXE =Test
CC = g++
PROGRAM = app
CFLAGS =-std=c++14 -pthread
CFILES = main.cpp AIB.cpp ULSTER.cpp BOA.cpp UNBL.cpp SWBPLC.cpp
HFILES = TM.h TX.h OSTM.h AIB.h ULSTER.h BOA.h UNBL.h SWBPLC.h

all:
```

Rule for SHARED linking

```
:
*.cpp -I -L /usr/lib/lib_o_stm.so -o
clean:
rm -f *.o
```

1. Run the application/executable file : ./Test

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 File Index

4.1 File List

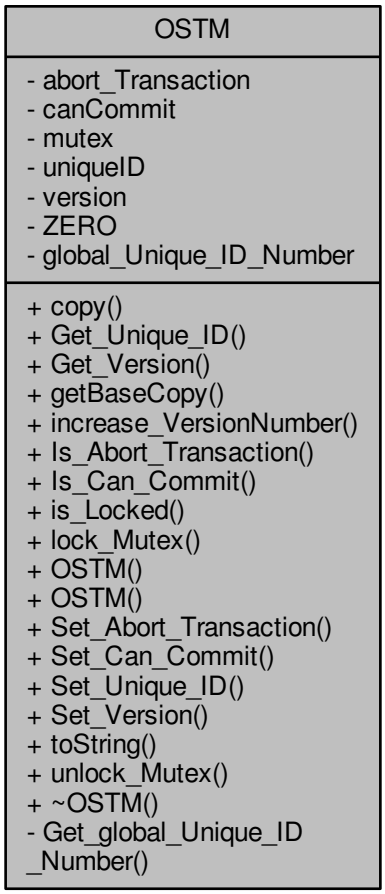
Here is a list of all files with brief descriptions:

5 Class Documentation

5.1 OSTM Class Reference

```
#include <OSTM.h>
```

Collaboration diagram for OSTM:



Public Member Functions

- virtual void copy (std::shared_ptr< OSTM > from, std::shared_ptr< OSTM > to)

  The copy virtual method required for deep copy between objetcs within the transaction.
- int Get_Unique_ID () const

  @82 Function <Get_Unique_ID> getter for uniqueID private field
- int Get_Version () const

  @100 Function <Get_Version> setter for version private filed
- virtual std::shared_ptr< OSTM > getBaseCopy (std::shared_ptr< OSTM > object)

The getbasecopy virtual method required for create a copy of the origin object/pointer and returning a copy of the object/pointer.

- void increase_VersionNumber ()

  @108 Function <increase_VersionNumber> commit time increase the version number associated with the object
- bool Is_Abort_Transaction () const

  @140 Function <Is_Abort_Transaction> return boolean value stored in the <abortTransaction> private filed
- bool Is_Can_Commit () const

  @124 Function <Is_Can_Commit> boolean function to determin the object can comit or need to roolback.
- bool is_Locked ()

  @162 Function <is_Locked> Boolean function to try lock the object. If the object not locked then locks and return True it otherwise return False.
- void lock_Mutex ()

  @145 Function <lock_Mutex> setter for mutex to lock the object
- OSTM ()

  @21 Default constructor
- OSTM (int _version_number_, int _unique_id_)

  @39 Custom Constructor Used to copying objects
- void Set_Abort_Transaction (bool abortTransaction)

  @132 Function <Set_Abort_Transaction> setter for abortTransaction private filed
- void Set_Can_Commit (bool canCommit)

  @117 Function <Set_Can_Commit> setter for canCommit private filed
- void Set_Unique_ID (int uniqueID)

  @75 Function <Set_Unique_ID> setter for uniqueID private field
- void Set_Version (int version)

  @92 Function <Set_Version> setter for version private filed
- virtual void toString ()

  The toString function displaying/representing the object on the terminal is string format.
- void unlock_Mutex ()

  @154 Function <unlock_Mutex> setter for mutex to unlock the object
- virtual ~OSTM ()

Private Member Functions

- int Get_global_Unique_ID_Number ()

  @61 Get_global_Unique_ID_Number function, If <global_Unique_ID_Number> equals to 10000000 then reset back to ZERO, to make sure the value of global_Unique_ID_Number never exceed the MAX_INT value

Private Attributes

- bool abort_Transaction
- bool canCommit
- std::mutex mutex
- int uniqueID
- int version
- const int ZERO = 0

Static Private Attributes

- static int global_Unique_ID_Number = 0

5.1.1 Detailed Description

File: OSTM.h Author: Zoltan Fuzesi C00197361, IT Carlow, Software Engineering,

Supervisor : Joe Kehoe,

C++ Software Transactional Memory,

Created on December 18, 2017, 2:09 PM The OSTM class is the base class to all the inherited classes that intend to used with the Software Transactional memory library

Definition at line 23 of file OSTM.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 OSTM::OSTM ( )

@21 Default constructor

@24 Integer field <version> indicates the version number of the inherited child object

@26 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM library

@28 Boolean value <canCommit> to determine the object can or cannot commit

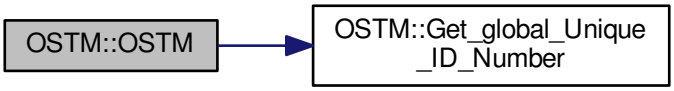
@30 Boolean field <abort_Transaction> to determine the object can or cannot commit

Definition at line 21 of file OSTM.cpp.

References abort_Transaction, canCommit, Get_global_Unique_ID_Number(), uniqueID, version, and ZERO.

```
00022 {
00024     this->version = ZERO;
00026     this->uniqueID = Get_global_Unique_ID_Number();
00028     this->canCommit = true;
00030     this->abort_Transaction = false;
00031 }
```

Here is the call graph for this function:



5.1.2.2 OSTM::OSTM (int *_version_number_*, int *_unique_id_*)

@39 Custom Constructor Used to copying objects

Parameters

version_number	Integer value used to create a copy of the object with the actual version
unique_id	Integer value used to create a copy of the object with the original unique ID

@42 Integer field <version> indicates the version number of the inherited child object

@44 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM library

@46 Boolean value <canCommit> to determine the object can or cannot commit

@48 Boolean value <abort_Transaction> to determine the object can or cannot commit

Definition at line 39 of file OSTM.cpp.

References abort_Transaction, canCommit, uniqueID, and version.

```
00040 {
00042     this->uniqueID = _unique_id_;
00044     this->version = _version_number_;
00046     this->canCommit = true;
00048     this->abort_Transaction = false;
00049 }
```

5.1.2.3 OSTM::~OSTM ( ) [virtual]

@54 Default De-constructor Delete the object.

Definition at line 54 of file OSTM.cpp.

```
00054             {
00056 }
```

5.1.3 Member Function Documentation

5.1.3.1 virtual void OSTM::copy (std::shared_ptr< OSTM > *from*, std::shared_ptr< OSTM > *to*) [inline], [virtual]

The copy virtual method required for deep copy between objetcs within the transaction.

See also

copy function implementation in inherited class class

Definition at line 41 of file OSTM.h.

```
00041 {};
```

5.1.3.2 int OSTM::Get_global_Unique_ID_Number ( ) [private]

@61 Get_global_Unique_ID_Number function, If <global_Unique_ID_Number> equals to 10000000 then reset back to ZERO, to make sure the value of global_Unique_ID_Number never exceed the MAX_INT value

Returning global_Unique_ID_Number to the constructor @64 Checking the global_Unique_ID_Number

@65 Reset global_Unique_ID_Number to ZERO

@67 return static global_Unique_ID_Number

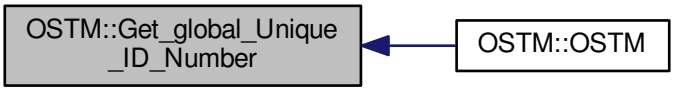
Definition at line 61 of file OSTM.cpp.

References global_Unique_ID_Number.

Referenced by OSTM().

```
00061                                   {
00063     if(global_Unique_ID_Number > 10000000)
00065         global_Unique_ID_Number = 0;
00067     return ++global_Unique_ID_Number;
00068 }
```

Here is the caller graph for this function:



5.1.3.3 int OSTM::Get_Unique_ID ( ) const

@82 Function <Get_Unique_ID> getter for uniqueID private field

@85 return Object uniqueID

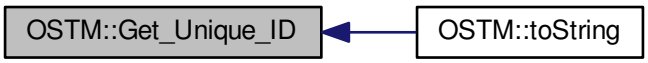
Definition at line 82 of file OSTM.cpp.

References uniqueID.

Referenced by toString().

```
00083 {
00085     return uniqueID;
00086 }
```

Here is the caller graph for this function:



5.1.3.4 int OSTM::Get_Version ( ) const

@100 Function <Get_Version> setter for version private filed

return object version number

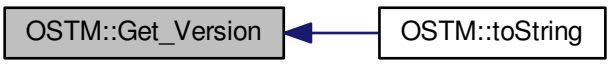
Definition at line 100 of file OSTM.cpp.

References version.

Referenced by toString().

```
00101 {
00103     return version;
00104 }
```

Here is the caller graph for this function:



5.1.3.5 virtual std::shared_ptr<OSTM> OSTM::getBaseCopy (std::shared_ptr< OSTM > *object*) [inline], [virtual]

The getbasecopy virtual method required for create a copy of the origin object/pointer and returning a copy of the object/pointer.

See also

getBaseCopy function implementation in child class

Definition at line 46 of file OSTM.h.

```
00046 {};//std::cout << "[OSTM GETBASECOPY]" << std::endl;};
```

5.1.3.6 void OSTM::increase_VersionNumber ( )

@108 Function <increase_VersionNumber> commit time increase the version number associated with the object

@111 increase object version number

Definition at line 108 of file OSTM.cpp.

References version.

Referenced by toString().

```
00109 {
00111     this->version += 1;
00112 }
```

Here is the caller graph for this function:

5.1.3.7 bool OSTM::Is_Abort_Transaction ( ) const

@140 Function <Is_Abort_Transaction> return boolean value stored in the <abortTransaction> private filed

Parameters

abort_Transaction	Boolean to determine the object can or cannot commit

@142 return abort_Transaction object boolean value

Definition at line 140 of file OSTM.cpp.

References abort_Transaction.

Referenced by toString().

```
00140                                   {
00142     return abort_Transaction;
00143 }
```

Here is the caller graph for this function:

5.1.3.8 bool OSTM::Is_Can_Commit ( ) const

@124 Function <Is_Can_Commit> boolean function to determin the object can comit or need to roolback.

@126 return canCommit boolean value TRUE/FALSE

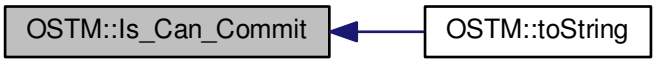
Definition at line 124 of file OSTM.cpp.

References canCommit.

Referenced by toString().

```
00124                                   {
00126     return canCommit;
00127 }
```

Here is the caller graph for this function:



5.1.3.9 bool OSTM::is_Locked ( )

@162 Function <is_Locked> Boolean function to try lock the object. If the object not locked then locks and return True it otherwise return False.

@164 Try to unlock the mutex, return TRUE if the lock was acquired successfully, otherwise return FALSE

Definition at line 162 of file OSTM.cpp.

References mutex.

Referenced by toString().

```
00162                    {
00164     return this->mutex.try_lock();
00165 }
```

Here is the caller graph for this function:

5.1.3.10 void OSTM::lock_Mutex ( )

@145 Function <lock_Mutex> setter for mutex to lock the object

@149 Locking the mutex

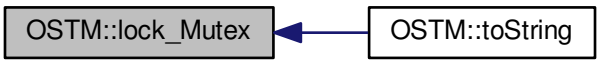
Definition at line 147 of file OSTM.cpp.

References mutex.

Referenced by toString().

```
00147                    {
00149     this->mutex.lock();
00150 }
```

Here is the caller graph for this function:



5.1.3.11 void OSTM::Set_Abort_Transaction (bool *abortTransaction*)

@132 Function <Set_Abort_Transaction> setter for abortTransaction private filed

Parameters

abortTransaction	Boolean to determine the object can or cannot commit

@134 set abort_Transaction object variable to parameter boolean value

Definition at line 132 of file OSTM.cpp.

References abort_Transaction.

Referenced by toString().

```
00132                                                   {
00134     this->abort_Transaction = abortTransaction;
00135 }
```

Here is the caller graph for this function:

5.1.3.12 void OSTM::Set_Can_Commit (bool *canCommit*)

@117 Function <Set_Can_Commit> setter for canCommit private filed

Parameters

canCommit	Boolean value to determine the object can or cannot commit

@119 set canCommit object variable to parameter boolean value

Definition at line 117 of file OSTM.cpp.

References canCommit.

Referenced by toString().

```
00117                                       {
00119     this->canCommit = canCommit;
00120 }
```

Here is the caller graph for this function:

5.1.3.13 void OSTM::Set_Unique_ID (int *uniqueID*)

@75 Function <Set_Unique_ID> setter for uniqueID private field

Parameters

uniqueID	int Every object inherit from OSTM class will include a version number that is unique for every object. The STM library used this value to find object within the transaction to make changes or comparism ith them.

@77 set object uniqueID to parameter integer value

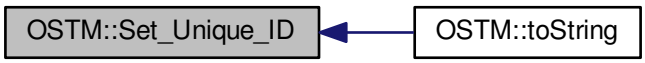
Definition at line 75 of file OSTM.cpp.

References uniqueID.

Referenced by toString().

```
00075                                   {
00077     this->uniqueID = uniqueID;
00078 }
```

Here is the caller graph for this function:



5.1.3.14 void OSTM::Set_Version (int *version*)

@92 Function <Set_Version> setter for version private filed

Parameters

version	integer The verion number ZERO by default when the object created. When a transaction make changes with the object, then the version number will be increased, to indicate the changes on the object.

@95 set object version to parameter integer value

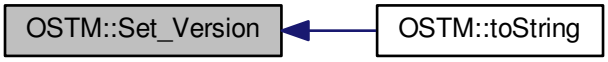
Definition at line 92 of file OSTM.cpp.

References version.

Referenced by toString().

```
{
    this->version = version;
}
```

Here is the caller graph for this function:



5.1.3.15 virtual void OSTM::toString ( ) [inline], [virtual]

The toString function displaying/representing the object on the terminal is string format.

See also

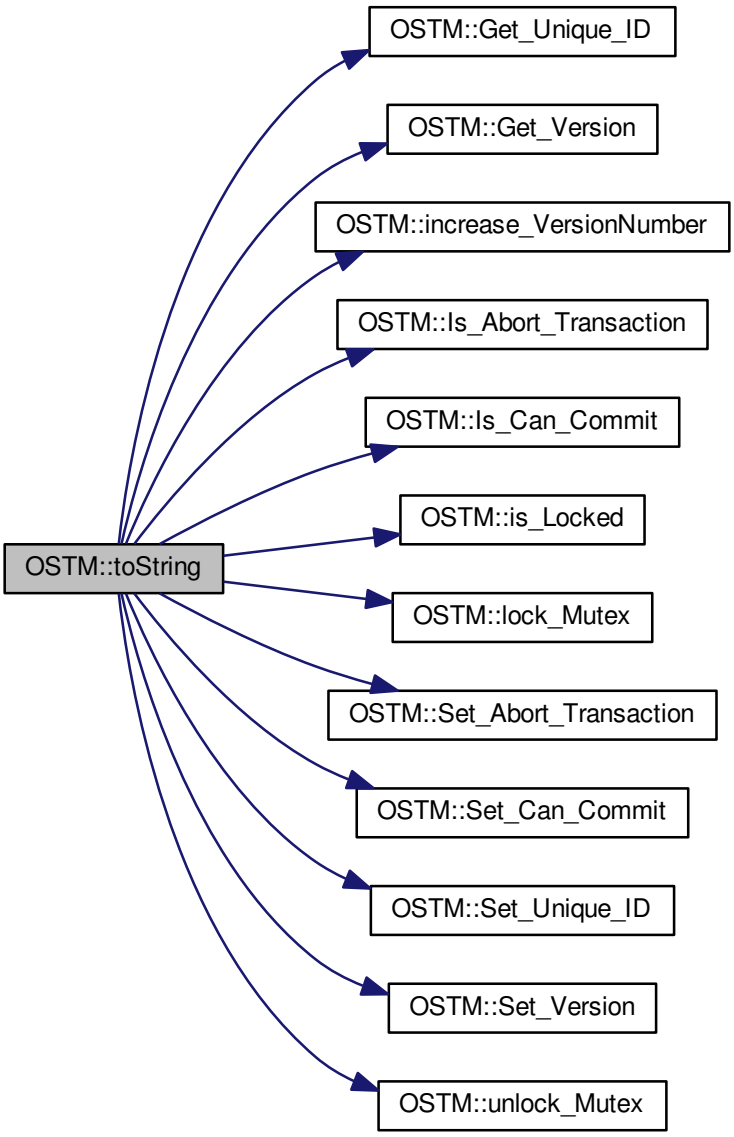
toString function implementation in child class

Definition at line 51 of file OSTM.h.

References canCommit, Get_Unique_ID(), Get_Version(), increase_VersionNumber(), Is_Abort_Transaction(), Is_Can_Commit(), is_Locked(), lock_Mutex(), Set_Abort_Transaction(), Set_Can_Commit(), Set_Unique_ID(), Set_Version(), uniqueID, unlock_Mutex(), and version.

```
{};
```

Here is the call graph for this function:



5.1.3.16 void OSTM::unlock_Mutex ( )

@154 Function <unlock_Mutex> setter for mutex to unlock the object

@156 Locking the mutex

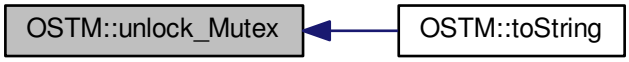
Definition at line 154 of file OSTM.cpp.

References mutex.

Referenced by toString().

```
00154                         {
00156     this->mutex.unlock();
00157 }
```

Here is the caller graph for this function:



5.1.4 Member Data Documentation

5.1.4.1 bool OSTM::abort_Transaction [private]

Boolean value <abort_Transaction> to determine the object can or cannot commit

Definition at line 125 of file OSTM.h.

Referenced by Is_Abort_Transaction(), OSTM(), and Set_Abort_Transaction().

5.1.4.2 bool OSTM::canCommit [private]

Boolean value <canCommit> to determine the object can or cannot commit

Definition at line 121 of file OSTM.h.

Referenced by Is_Can_Commit(), OSTM(), Set_Can_Commit(), and toString().

5.1.4.3 int OSTM::global_Unique_ID_Number = 0 [static], [private]

Unique object number start at ZERO The value stored in the static class level <global_Unique_ID_Number> increase every OSTM type object creation.

Definition at line 130 of file OSTM.h.

Referenced by Get_global_Unique_ID_Number().

5.1.4.4 std::mutex OSTM::mutex [private]

Mutex lock <mutex> use to lock the object with transaction, to make sure only one transaction can access the object at the time

Definition at line 139 of file OSTM.h.

Referenced by is_Locked(), lock_Mutex(), and unlock_Mutex().

5.1.4.5 int OSTM::uniqueID [private]

Object unique identifier Every object inherit from OSTM class will include a version number that is unique for every object. The STM library used this value to find object within the transaction to make changes or comparism ith them.

Definition at line 117 of file OSTM.h.

Referenced by Get_Unique_ID(), OSTM(), Set_Unique_ID(), and toString().

5.1.4.6 int OSTM::version [private]

Object private version number. The verion number ZERO by default when the object created. When a transaction make changes with the object, then the version number will be increased, to indicate the changes on the object.

Definition at line 111 of file OSTM.h.

Referenced by Get_Version(), increase_VersionNumber(), OSTM(), Set_Version(), and toString().

5.1.4.7 const int OSTM::ZERO = 0 [private]

Integer <ZERO> meaninful string equalient to 0

Definition at line 134 of file OSTM.h.

Referenced by OSTM().

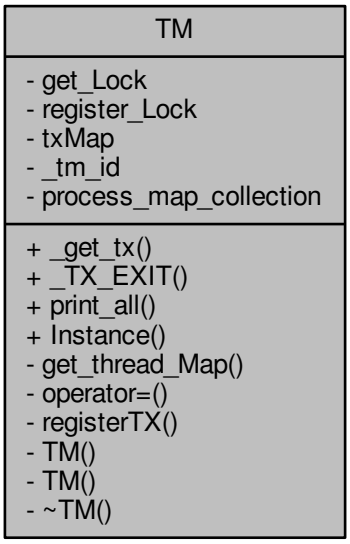
The documentation for this class was generated from the following files:

- OSTM.h
- OSTM.cpp

5.2 TM Class Reference

`#include <TM.h>`

Collaboration diagram for TM:



Public Member Functions

- std::shared_ptr< TX > const _get_tx ()

  @81 _get_tx std::shared_ptr<TX>, return an trtansaction Object as a shared_ptr, if TX not exists then create and register.# If the transaction Object exists then increasing the nesting level within the Transaction Object.
- void _TX_EXIT ()

  @108 _TX_EXIT void, when the thread calls the ostm_exit function in the transaction, and it will clear all elements from the shared global collection associated with the main process
- void print_all ()

  @132 ONLY FOR TESTING print_all void function , print out all object key from txMAP collection associated with the main process.

Static Public Member Functions

- static TM & Instance ()

  @31 Instance TM, Scott Meyer's Singleton creation, thread safe Transaction Manager instance creation.

Private Member Functions

- std::map< std::thread::id, int > get_thread_Map ()

  @148 get_thread_Map std::map, returning a map to store all unique ID from all objects from all transactions within the main processes
- TM & operator= (const TM &)=delete

  TM copy operator, prevent from copying the Transaction Manager.
- void registerTX ()

  @45 registerTX void function, register a new TX Transaction object into ythe txMap/Transaction Map to manage all the transactions within the shared library. TM Transaction managger checking the Process ID existence in the process map collection, If not in the map then register.
- TM ()=default
- TM (const TM &)=delete

  TM copy constructor, prevent from copying the Transaction Manager.
- ~TM ()=default

Private Attributes

- std::mutex get_Lock
- std::mutex register_Lock
- std::map< std::thread::id, std::shared_ptr< TX > > txMap

Static Private Attributes

- static pid_t _tm_id
- static std::map< pid_t, std::map< std::thread::id, int > > process_map_collection

5.2.1 Detailed Description

Definition at line 57 of file TM.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 TM::TM ( ) [private],[default]

5.2.2.2 TM::~TM ( ) [private],[default]

5.2.2.3 TM::TM (const TM &) [private],[delete]

TM copy constructor, prevent from copying the Transaction Manager.

5.2.3 Member Function Documentation

5.2.3.1 std::shared_ptr< TX > const TM::_get_tx ( )

@81 _get_tx std::shared_ptr<TX>, return an trtansaction Object as a shared_ptr, if TX not exists then create and register.# If the transaction Object exists then increasing the nesting level within the Transaction Object.

_get_tx std::shared_ptr<TX>, returning a shared pointer transaction object @85 guard std::lock_guard, locks the get_Lock mutex, unlock automatically when goes out of the scope

@85 get_Lock std::mutex, used by the lock_guard to protect txMap from race conditions

@87 txMap try to find the TX Transaction object by it's actual thread ID if registred in the txMap

@89 Check if iterator pointing to the end of the txMap then insert

@92 If cannot find then call the register function to register the thread with a transaction

@94 If it's registered first time then we need to find it after registration

@98 If transaction already registered, it means the thread participating in nested transactions, and increase the nesting

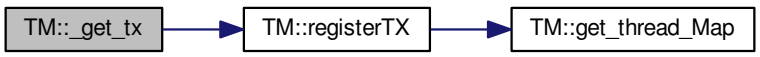
@101 Returning back the transaction (TX) object to the thread

Definition at line 81 of file TM.cpp.

References get_Lock, registerTX(), and txMap.

```
00082 {
00085     std::lock_guard<std::mutex> guard(get_Lock);
00087     std::map<std::thread::id, std::shared_ptr<TX>>::iterator it = txMap.find(std::this_thread::get_id(
      ));
00089     if(it == txMap.end())
00090     {
00092        registerTX();
00094        it = txMap.find(std::this_thread::get_id());
00095
00096     } else {
00098         it->second->_increase_tx_nesting();
00099     }
00101     return it->second;
00102
00103 }
```

Here is the call graph for this function:



5.2.3.2 void TM::_TX_EXIT ( )

@108 _TX_EXIT void, when the thread calls the ostm_exit function in the transaction, and it will clear all elements from the shared global collection associated with the main process

_TX_EXIT void function, the thread (TX object) calls the ostm_exit function from the transaction, and clear all elements from the shared global collection associated with the main process @110 Transaction manger create a local Transaction Object to access the TX class function without nesting any transaction

@112 getppid() return the actual main process thread id, I used it to associate the Transactionas with the main processes

@114 process_map_collection try to find the main process by it's ppid if registred in the library

@116 Check if iterator NOT pointing to the end of the process map then register

@118 Iterate through the process_map_collection to find all transaction associated with main process

@120 Delete all transaction associated with the actual main process

@123 When all transaction deleted, delete the main process from the Transacion Manager

@126 TX class delete all Global Object shared between the transaction. This function calls only when the main process exists to clear out memory

Definition at line 108 of file TM.cpp.

References TX::ostm_exit(), process_map_collection, and txMap.

```
00108                 {
00110     TX tx(std::this_thread::get_id());
00112     pid_t ppid = getppid();
00114     std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
      TM::process_map_collection.find(ppid);
00116     if (process_map_collection_Iterator != TM::process_map_collection.end()) {
00118         for (auto current = process_map_collection_Iterator->second.begin(); current !=
      process_map_collection_Iterator->second.end(); ++current) {
00120             txMap.erase(current->first);
00121         }
00123         TM::process_map_collection.erase(ppid);
00124     }
00126     tx.ostm_exit();
00127 }
```

Here is the call graph for this function:

5.2.3.3 std::map< std::thread::id, int > TM::get_thread_Map ( ) [private]

@148 get_thread_Map std::map, returning a map to store all unique ID from all objects from all transactions within the main processes

@150 thread_Map std::map< int, int > create a map to store int key and int value

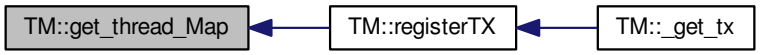
@152 return the map

Definition at line 148 of file TM.cpp.

Referenced by registerTX().

```
00148                                                {
00150     std::map< std::thread::id, int > thread_Map;
00152     return thread_Map;
00153 }
```

Here is the caller graph for this function:



5.2.3.4 TM & TM::Instance ( ) [static]

@31 Instance TM, Scott Meyer's Singleton creation, thread safe Transaction Manager instance creation.

Scott Meyer's Singleton creation, thread safe Transaction Manager instance creation. @33 _instance TM, static class reference to the instance of the Transaction Manager class

@35 _instance ppid, assigning the process id whoever created the Singleton instance

@37 return Singleton instance

Definition at line 31 of file TM.cpp.

References _tm_id.

```
00031                    {
00033     static TM _instance;
00035     _instance._tm_id = getpid();
00037     return _instance;
00038 }
```

5.2.3.5 TM& TM::operator= (const TM &) [private],[delete]

TM copy operator, prevent from copying the Transaction Manager.

5.2.3.6 void TM::print_all ( )

@132 ONLY FOR TESTING print_all void function , print out all object key from txMAP collection associated with the main process.

ONLY FOR TESTING! print_all void function, prints all object in the txMap @134 Locking the print function

@136 Iterate through the txMap to print out the thread id's

@138 Print key (thread number)

@140 Unlocking the print function

Definition at line 132 of file TM.cpp.

References get_Lock, and txMap.

```
00132                   {
00134     get_Lock.lock();
00136     for (auto current = txMap.begin(); current != txMap.end(); ++current) {
00138         std::cout << "KEY : " << current->first << std::endl;
00139     }
00141     get_Lock.unlock();
00142 }
```

5.2.3.7 void TM::registerTX ( ) [private]

@45 registerTX void function, register a new TX Transaction object into ythe txMap/Transaction Map to manage all the transactions within the shared library. TM Transaction managger checking the Process ID existence in the process map collection, If not in the map then register.

@49 guard std::lock_guard, locks the register_Lock mutex, unlock automatically when goes out of the scope

@49 register_Lock std::mutex, used by the lock_guard to protect shared map from race conditions

@51 getppid() return the actual main process thread id, I used it to associate the Transactionas with the main processes

@53 process_map_collection try to find the main process by it's ppid if registred in the library

@55 Check if iterator pointing to the end of the process map then register

@57 Require new map to insert to the process map as a value by the ppid key

@59 Register main process/application to the global map

@63 txMap std::map, collection to store all transaction created by the Transaction Manager

@65 Check if iterator pointing to the end of the txMap then insert

@67 Create a new Transaction Object as a shared pointer

@69 txMap insert the new transaction into the txMap by the threadID key

@71 Get the map if the transaction registered first time

@73 Insert to the GLOBAL MAP as a helper to clean up at end of main process. The value 1 is not used yet

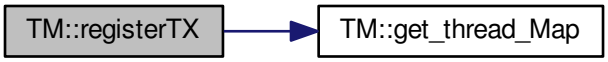
Definition at line 45 of file TM.cpp.

References get_thread_Map(), process_map_collection, register_Lock, and txMap.

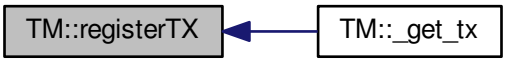
Referenced by _get_tx().

```
{
    std::lock_guard<std::mutex> guard(register_Lock);
    pid_t ppid = getppid();
    std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator = TM::process_map_collection.find(ppid);
    if (process_map_collection_Iterator == TM::process_map_collection.end()) {
        std::map< std::thread::id, int >map = get_thread_Map();
        TM::process_map_collection.insert({ppid, map});

    }
    std::map<std::thread::id, std::shared_ptr < TX>>::iterator it = txMap.find(std::this_thread::get_id());
    if (it == txMap.end()) {
        std::shared_ptr<TX> _transaction_object(new TX(std::this_thread::get_id()));
        txMap.insert({std::this_thread::get_id(), _transaction_object});
        process_map_collection_Iterator = TM::process_map_collection.find(ppid);
        process_map_collection_Iterator->second.insert({std::this_thread::get_id(), 1});
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.4 Member Data Documentation

5.2.4.1 pid_t TM::_tm_id [static], [private]

Definition at line 102 of file TM.h.

Referenced by Instance().

5.2.4.2 std::mutex TM::get_Lock [private]

Definition at line 98 of file TM.h.

Referenced by _get_tx(), and print_all().

5.2.4.3 std::map< pid_t, std::map< std::thread::id, int > > TM::process_map_collection [static], [private]

Definition at line 82 of file TM.h.

Referenced by _TX_EXIT(), and registerTX().

5.2.4.4 std::mutex TM::register_Lock [private]

Definition at line 94 of file TM.h.

Referenced by registerTX().

5.2.4.5 std::map<std::thread::id, std::shared_ptr<TX> > TM::txMap [private]

Definition at line 78 of file TM.h.

Referenced by _get_tx(), _TX_EXIT(), print_all(), and registerTX().

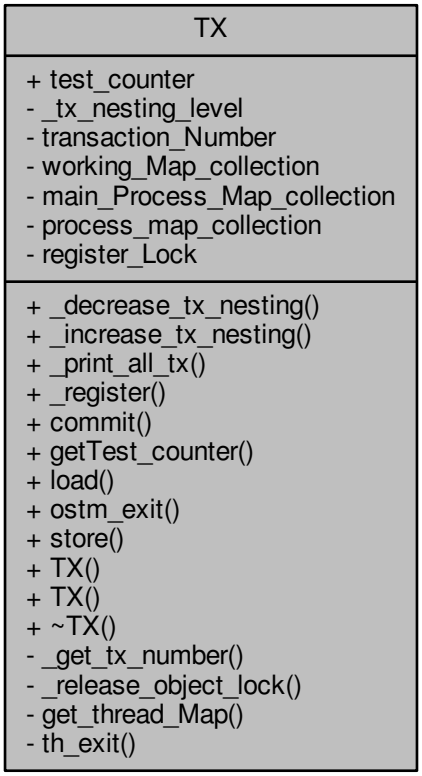
The documentation for this class was generated from the following files:

- TM.h
- TM.cpp

5.3 TX Class Reference

```
#include <TX.h>
```

Collaboration diagram for TX:



Public Member Functions

- void _decrease_tx_nesting ()

  @279 _decrease_tx_nesting decrease the value stored in _tx_nesting_level by one, when outer transactions commit
- void _increase_tx_nesting ()

  @272 _increase_tx_nesting increase the value stored in _tx_nesting_level by one, indicate that the transaction was nested
- void _print_all_tx ()
- void _register (std::shared_ptr< OSTM > object)

  register void, receives an std::shared_ptr<OSTM> that point to the original memory space to protect from reca conditions
- bool commit ()

  @176 commit function, returns boolean value TRUE/FALSE depends on the action taken within the function. if commit happens return TRUE, otherwise return FALSE, indicate the transaction muist restart.
- int getTest_counter ()

  @287 getTest_counter TESTING ONLY!!! returning the value of the test_counter stored, representing the number of rollbacks
- std::shared_ptr< OSTM > load (std::shared_ptr< OSTM > object)

  @137 load std::shared_ptr<OSTM>, returning an OSTM type shared pointer, that is copy of the original pointer stored in the working map, to work with during transaction life time
- void ostm_exit ()

  @68 ostm_exit void, clear all elements from the shared global collections associated with the main process
- void store (std::shared_ptr< OSTM > object)

  @157 store void, receive an OSTM type shared pointer object to store the changes with the transaction copy object
- TX (std::thread::id id)

  @36 Custom Constructor
- TX (const TX &orig)
- ~TX ()

  @45 De-constructor

Static Public Attributes

- static int test_counter = 0

Private Member Functions

- const std::thread::id _get_tx_number () const

  @294 _get_tx_number, returning the thread id that has assigned the given transaction
- void _release_object_lock ()

  @253 _release_object_lock void function, is get called from commit function, with the purpose to release the locks on all the objects participating in the transaction
- std::map< int, int > get_thread_Map ()

  @301 get_thread_Map, returning a map to store all unique ID from all objects from all transactions within the main process
- void th_exit ()

  @52 th_exit void, delete all std::shared_ptr<OSTM> elements from working_Map_collection, that store pointers to working objects

Private Attributes

- int _tx_nesting_level
- std::thread::id transaction_Number
- std::map< int, std::shared_ptr< OSTM > > working_Map_collection

Static Private Attributes

- static std::map< int, std::shared_ptr< OSTM > > main_Process_Map_collection
- static std::map< pid_t, std::map< int, int > > process_map_collection
- static std::mutex register_Lock

Friends

- class TM

5.3.1 Detailed Description

Definition at line 29 of file TX.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 TX::TX (std::thread::id *id*)

@36 Custom Constructor

Parameters

id	std::thread::id, represent the transaction number when to the TransactionManager

@38 Integer field <transaction_Number> indicates the transaction number to the Transaction manager

@40 Integer field <_tx_nesting_level> indicates the nesting level to the transaction itself

Definition at line 36 of file TX.cpp.

References _tx_nesting_level, and transaction_Number.

```
00036                     {
00038     this->transaction_Number = id;
00040     this->_tx_nesting_level = 0;
00041 }
```

5.3.2.2 TX::~TX ( )

@45 De-constructor

Delete the object.

Definition at line 45 of file TX.cpp.

```
00045         {
00047 }
```

5.3.2.3 TX::TX (const TX & *orig*)

5.3.3 Member Function Documentation

5.3.3.1 void TX::_decrease_tx_nesting ( )

@279 _decrease_tx_nesting decrease the value stored in _tx_nesting_level by one, when outer transactions commit

@281 Decrease transaction nesting level

Definition at line 279 of file TX.cpp.

References _tx_nesting_level.

Referenced by commit().

```
00279                                {
00281     this->_tx_nesting_level -= 1;
00282 ;
00283 }
```

Here is the caller graph for this function:

5.3.3.2 const std::thread::id TX::_get_tx_number ( ) const [private]

@294 _get_tx_number, returning the thread id that has assigned the given transaction

_get_tx_number, returning the transaction uniqe identifier @296 Return the transaction nuber

Definition at line 294 of file TX.cpp.

References transaction_Number.

```
00294                                          {
00296     return transaction_Number;
00297 }
```

5.3.3.3 void TX::_increase_tx_nesting ( )

@272 _increase_tx_nesting increase the value stored in _tx_nesting_level by one, indicate that the transaction was nested

@274 Increase transaction nesting level

Definition at line 272 of file TX.cpp.

References _tx_nesting_level.

```
00272                                {
00274     this->_tx_nesting_level += 1;
00275 }
```

5.3.3.4 void TX::_print_all_tx ( )

@311 _print_all_tx, only for testing! Prints all transaction associated with the main procees.! @313 initialise Iterator

@315 getppid() return the actual main process thread id, I used it to associate the Transactionas with the main processes

'317 initialize and assign Iterator to process_map_collection, by the main process id (ppid)

@319 If there is an entry associated with the process then print out all transactions.

@321 Iterate through process_map_collection

@323 Assign value to iterator

@325 If value found, then print it

@327 print out the transaction number

Definition at line 311 of file TX.cpp.

References process_map_collection, and working_Map_collection.

```
00311                        {
00313     std::map< int, std::shared_ptr<OSTM> >::iterator it;
00315     pid_t ppid = getppid();
00317     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00319     if (process_map_collection_Iterator != TX::process_map_collection.end()) {
00321         for (auto current = process_map_collection_Iterator->second.begin(); current !=
      process_map_collection_Iterator->second.end(); ++current) {
00323             it = working_Map_collection.find(current->first);
00325             if(it != working_Map_collection.end()){
00327                 std::cout << "[Unique number ] : " <<it->second->Get_Unique_ID() << std::endl;
00328             }
00329         }
00330     }
00331 }
```

5.3.3.5 void TX::_register (std::shared_ptr< OSTM > *object*)

register void, receives an std::shared_ptr<OSTM> that point to the original memory space to protect from reca conditions

Parameters

object	std::shared_ptr<OSTM>, is an original shared pointer point to the object memory space

@98 register_Lock(mutex) shared lock between all transaction. MUST USE SHARED LOCK TO PROTECT SH↩ ARED GLOBAL MAP/COLLECTION

@100 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!!

@104 getppid() return the actual main process thread id, I used it to associate the Transactionas with the main processes

@106 Declare and initialize Iterator for process_map_collection, find main process

@108 If iterator cannot find main process, then register

@110 Create new empty map

@112 Register main process/application to the global map

@114 Get the map if registered first time

@117 Declare and initialize Iterator for main_Process_Map_collection, find by original object

@119 If object cannot find, then register

'121 Insert the origin object to the GLOBAL MAP shared between transactions

@123 Insert object ID to the GLOBAL MAP as a helper to clean up at end of main process, Second value (1) not specified yet

@126 Declare and initialize Iterator for working_Map_collection, find copy of the original object

@128 If copy of the object not found, then register

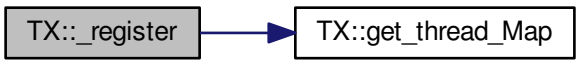
@130 Register transaction own copy of the original object

Definition at line 96 of file TX.cpp.

References get_thread_Map(), main_Process_Map_collection, process_map_collection, register_Lock, and working_Map_collection.

```
00096                                                {
00098     std::lock_guard<std::mutex> guard(TX::register_Lock);
00100     if(object == nullptr){
00101         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN REGISTER FUNCTION]") );
00102     }
00104     pid_t ppid = getppid();
00106     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00108     if (process_map_collection_Iterator == TX::process_map_collection.end()) {
00110         std::map< int, int >map =  get_thread_Map();
00112         TX::process_map_collection.insert({ppid, map});
00114         process_map_collection_Iterator = TX::process_map_collection.find(ppid);
00115     }
00117     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find(object->Get_Unique_ID());
00119     if (main_Process_Map_collection_Iterator == TX::main_Process_Map_collection
      .end()) {
00121         TX::main_Process_Map_collection.insert({object->Get_Unique_ID(),
      object});
00123         process_map_collection_Iterator->second.insert({object->Get_Unique_ID(), 1});
00124     }
00126     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator
      = working_Map_collection.find(object->Get_Unique_ID());
00128     if (working_Map_collection_Object_Shared_Pointer_Iterator ==
      working_Map_collection.end()) {
00130         working_Map_collection.insert({object->Get_Unique_ID(), object->getBaseCopy(
      object)});
00131     }
00132 }
```

Here is the call graph for this function:



5.3.3.6 void TX::_release_object_lock ( ) [private]

@253 _release_object_lock void function, is get called from commit function, with the purpose to release the locks on all the objects participating in the transaction

_release_object_lock, Release the locks on all Shared global objects used by the transaction @255 Declare Iterator for working_Map_collection

@255 Declare Iterator for working_Map_collection

@260 Find Global shared original object by the transaction object unique ID

@262 If object found, then release lock

@264 Release object lock

Definition at line 253 of file TX.cpp.

References main_Process_Map_collection, and working_Map_collection.

Referenced by commit().

```
00253                                {
00255     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00257     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
00258     for (working_Map_collection_Object_Shared_Pointer_Iterator =
      working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator
       != working_Map_collection.end();
      working_Map_collection_Object_Shared_Pointer_Iterator++) {
00260             main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find((
      working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
00262             if (main_Process_Map_collection_Iterator !=
      TX::main_Process_Map_collection.end()) {
00264                 (main_Process_Map_collection_Iterator)->second->unlock_Mutex();
00265             }
00266         }
00267 }
```

Here is the caller graph for this function:

5.3.3.7 bool TX::commit ( )

@176 commit function, returns boolean value TRUE/FALSE depends on the action taken within the function. if commit happens return TRUE, otherwise return FALSE, indicate the transaction muist restart.

@179 Declare can_Commit boolean variable

@182 Dealing with nested transactions first. if nesting level bigger than ZERO do not commit yet

@183 Decrease nesting level

See also

_decrease_tx_nesting()

@187 Declare and initialize Iterator for working_Map_collection

@189 Declare and initialize Iterator for main_Process_Map_collectio

@191 Iterate through the working_Map_collection, for all associated copy objetcs

@193 Find the Original object in the Shared global colection by the copy object unique ID

@195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!!

@200 Busy waiting, If the object locked by another transaction, then waith until it's get unlockec, then lock it

@203 Compare the original global object version number with the working object version number. If the version number not same, then it cannot coomit

@2005 Set object boolean value to FALSE, cannot commit

@207 Set canCommit false Indicate rollback must happen

@210 If version number are has same value set object boolean value to TRUE

@214 IF can_Commit boolean value setted for FALSE then rollback all copy object in the transaction to the Global object values

@217 iterate through all transaction copy objects one by one

@219 Find the Global shared object by the transaction copy object unique ID

@221 Copy all Global shared original objects changed values by another transaction to the transaction copy objetcs

@224 When the transaction finish to change copying all values from original objects to local copy, then release all Global shared objects.

See also

_release_object_lock()

@226 Return FALSE to indicate the transaction must restart !

@229 Iterate through working_map_collection. If no conflict detected in early stage in the transaction, then commit all the local changes to shared Global objects

@231 Find the Global shared object by the transaction copy object unique ID

@233 If Global shared object found then commit changes

@235 Copy over local transaction object values to original Global object

@237 Increase the version number in the original pointer

@195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!!

@242 When the transaction finish with commit all changes, then release all Global shared objects.

See also

_release_object_lock()

@244 Transaction object clean up all associated values, clean memory.

See also

th_exit()

@246 Return TRUE, indicate the transaction has finished.

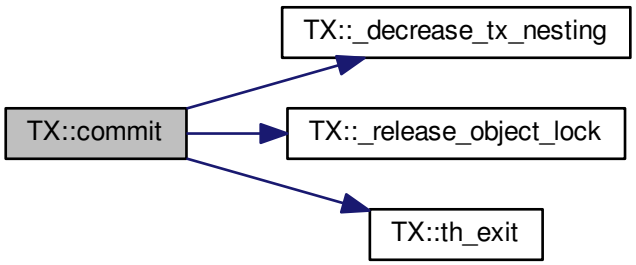
Definition at line 177 of file TX.cpp.

References _decrease_tx_nesting(), _release_object_lock(), _tx_nesting_level, main_Process_Map_collection, th_exit(), and working_Map_collection.

```
00177                 {
00179     bool can_Commit = true;
00181     if (this->_tx_nesting_level > 0) {
00183         _decrease_tx_nesting();
00184         return true;
00185     }
00187     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00189     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
00191     for (working_Map_collection_Object_Shared_Pointer_Iterator =
      working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator
       != working_Map_collection.end();
      working_Map_collection_Object_Shared_Pointer_Iterator++) {
00193             main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find(
      working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
00195             if(main_Process_Map_collection_Iterator ==
      TX::main_Process_Map_collection.end())
00196             {
00197                 throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT FUNCTION]")
      );
00198             }
00199
00201         while(!(main_Process_Map_collection_Iterator->second)->is_Locked());
00203         if (main_Process_Map_collection_Iterator->second->Get_Version() >
      working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Version()) {
00205             working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(false);
00207             can_Commit = false;
00208             break;
```

```
        } else {
            working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(true);
        }
    }
    if (!can_Commit) {
        for (working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end(); working_Map_collection_Object_Shared_Pointer_Iterator++) {
            main_Process_Map_collection_Iterator  = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            (working_Map_collection_Object_Shared_Pointer_Iterator->second)->copy(working_Map_collection_Object_Shared_Pointer_Iterator->second, main_Process_Map_collection_Iterator->second);
        }
        _release_object_lock();
        return false;
    } else {
        for (working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end(); working_Map_collection_Object_Shared_Pointer_Iterator++) {
                main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find((working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
                if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()) {
                    (main_Process_Map_collection_Iterator->second)->copy(main_Process_Map_collection_Iterator->second, working_Map_collection_Object_Shared_Pointer_Iterator->second);
                    main_Process_Map_collection_Iterator->second->increase_VersionNumber();
                } else { throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT FUNCTION]")); }
        }
        _release_object_lock();
        this->th_exit();
        return true;
    }
}//Commit finish
```

Here is the call graph for this function:



5.3.3.8 std::map< int, int > TX::get_thread_Map ( ) [private]

@301 get_thread_Map, returning a map to store all unique ID from all objects from all transactions within the main process

get_thread_Map, returning and map to insert to the process_map_collection as an inner value @303 initialize empty map hold int key and values

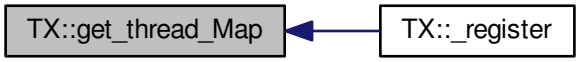
@305 Return the map

Definition at line 301 of file TX.cpp.

Referenced by _register().

```
00301                                       {
00303     std::map< int, int > thread_Map;
00305     return thread_Map;
00306 }
```

Here is the caller graph for this function:



5.3.3.9 int TX::getTest_counter ( )

@287 getTest_counter TESTING ONLY!!! returning the value of the test_counter stored, representing the number of rollbacks

@289 return class level value hold by test_counter variable

Definition at line 287 of file TX.cpp.

References test_counter.

```
00287                           {
00289     return TX::test_counter;
00290 }
```

5.3.3.10 std::shared_ptr< OSTM > TX::load (std::shared_ptr< OSTM > *object*)

@137 load std::shared_ptr<OSTM>, returning an OSTM type shared pointer, that is copy of the original pointer stored in the working map, to work with during transaction life time

Parameters

object	std::shared_ptr<OSTM>, used as a reference to find transaction copy object by the object unique ID

@139 Declare and initialize Iterator for working_Map_collection

@141 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!!

@145 Find copy object in working_Map_collection by the object unique ID

@147 If object found, then return it

@149 Returning a copy of the working copy object

@151 If no object found, throw runtime error

Definition at line 137 of file TX.cpp.

References working_Map_collection.

```
{
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN LOAD FUNCTION]") );
    }
    working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    if (working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end()) {
        return working_Map_collection_Object_Shared_Pointer_Iterator->second->getBaseCopy(working_Map_collection_Object_Shared_Pointer_Iterator->second);
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND LOAD FUNCTION]") );}
}
```

5.3.3.11 void TX::ostm_exit ( )

@68 ostm_exit void, clear all elements from the shared global collections associated with the main process

Parameters

main_Process_Map_collection	std::map, store all std::shared_ptr<OSTM> from all transaction shared between multiple processes
process_map_collection	std::map, store all unique id from all transaction within main process DO NOT CALL THIS METHOD EXPLICITLY!!!!!! WILL DELETE ALL PROCESS ASSOCIATED ELEMENTS!!!!

@70 Declare Iterator main_Process_Map_collection_Iterator

@72 getppid() return the actual main process thread id, I used it to associate the Transactionas with the main processes

@74 process_map_collection try to find the main process by it's ppid if registred in the library

@76 Check if iterator NOT pointing to the end of the process_map_collection then remove all associated elements

@78 Iterate through the process_map_collection to find all transaction associated with main process

@80 Find the OSTM object in the Global shared map

@82 If object found then delete it

@84 Delete element from shared main_Process_Map_collection by object by the unique key, and the shaed_ptr will destroy automatically

@88 Delete main process from Process_map_collection

Definition at line 68 of file TX.cpp.

References main_Process_Map_collection, and process_map_collection.

Referenced by TM::_TX_EXIT().

```
{
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    pid_t ppid = getppid();
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator = TX::process_map_collection.find(ppid);
    if (process_map_collection_Iterator != TX::process_map_collection.end()) {
        for (auto current = process_map_collection_Iterator->second.begin(); current != process_map_collection_Iterator->second.end(); ++current) {
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(current->first);
            if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()){
                TX::main_Process_Map_collection.erase(main_Process_Map_collection_Iterator->first);
            }
        }
        TX::process_map_collection.erase(process_map_collection_Iterator->first);
    }
}
```

Here is the caller graph for this function:

5.3.3.12 void TX::store (std::shared_ptr< OSTM > *object*)

@157 store void, receive an OSTM type shared pointer object to store the changes with the transaction copy object

Parameters

object	std::shared_ptr<OSTM>, receiving a changed shared pointer, that was returned from the load function

@159 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!!

@163 Declare and initialize Iterator for working_Map_collection

@165 Find copy object in working_Map_collection by the object unique ID

@167 If object found, then replace it

@169 Replace copy object in working_Map_collection associated with the unique ID key

@171 If error happes during store procees throw runtime error

Definition at line 157 of file TX.cpp.

References working_Map_collection.

```
00157                                         {
00159     if(object == nullptr){
00160         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN STORE FUNCTION]") );
00161     }
00163     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00165     working_Map_collection_Object_Shared_Pointer_Iterator =
      working_Map_collection.find(object->Get_Unique_ID());
00167     if (working_Map_collection_Object_Shared_Pointer_Iterator !=
      working_Map_collection.end()) {
00169         working_Map_collection_Object_Shared_Pointer_Iterator->second = object;
00171     } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND STORE FUNCTION, CANNOT
       STORE OBJECT]") );}
00172 }
```

5.3.3.13 void TX::th_exit ( ) [private]

@52 th_exit void, delete all std::shared_ptr<OSTM> elements from working_Map_collection, that store pointers to working objects

Clean up all associated values by the thread delete from working_Map_collection, it is an automated function by the transactions

Parameters

working_Map_collection	std::map, store std::shared_ptr<OSTM> transaction pointers

@54 If bigger than ZERO, means active nested transactions running in background, do not delete anything yet

Definition at line 52 of file TX.cpp.

References _tx_nesting_level, and working_Map_collection.

Referenced by commit().

```
00052                   {
00054     if (this->_tx_nesting_level > 0) {
00055         /* Active nested transactions running in background, do not delete anything yet */
00056     } else {
00057         /* Remove all elements map entries from transaction and clear the map */
00058         working_Map_collection.clear();
00059     }
00060 }
```

Here is the caller graph for this function:

5.3.4 Friends And Related Function Documentation

5.3.4.1 friend class TM [friend]

Definition at line 74 of file TX.h.

5.3.5 Member Data Documentation

5.3.5.1 int TX::_tx_nesting_level [private]

_tx_nesting_level, store integer value represent the ttransaction nesting level

Definition at line 101 of file TX.h.

Referenced by _decrease_tx_nesting(), _increase_tx_nesting(), commit(), th_exit(), and TX().

5.3.5.2 std::map< int, std::shared_ptr< OSTM > > TX::main_Process_Map_collection [static], [private]

main_Process_Map_collection, STATIC GLOBAL MAP Collection to store OSTM parent based shared pointers to control/lock and compare objects version number within transactions

Definition at line 105 of file TX.h.

Referenced by _register(), _release_object_lock(), commit(), and ostm_exit().

5.3.5.3 std::map< pid_t, std::map< int, int > > TX::process_map_collection [static], [private]

process_map_collection, STATIC GLOBAL MAP Collection to store all process associated keys to find when deleting transactions

Definition at line 109 of file TX.h.

Referenced by _print_all_tx(), _register(), and ostm_exit().

5.3.5.4 std::mutex TX::register_Lock [static], [private]

register_Lock, std::mutex to control shared access on MAIN MAP

Definition at line 117 of file TX.h.

Referenced by _register().

5.3.5.5 int TX::test_counter = 0 [static]

Definition at line 82 of file TX.h.

Referenced by getTest_counter().

5.3.5.6 std::thread::id TX::transaction_Number [private]

transaction_Number, Returning the transaction number what is a registered thread number associated with the transaction

Definition at line 97 of file TX.h.

Referenced by _get_tx_number(), and TX().

5.3.5.7 std::map< int, std::shared_ptr<OSTM> > TX::working_Map_collection [private]

working_Map_collection, Collection to store copy of OSTM parent based original Global shared pointers to make invisible changes during isolated transaction

Definition at line 93 of file TX.h.

Referenced by _print_all_tx(), _register(), _release_object_lock(), commit(), load(), store(), and th_exit().

The documentation for this class was generated from the following files:

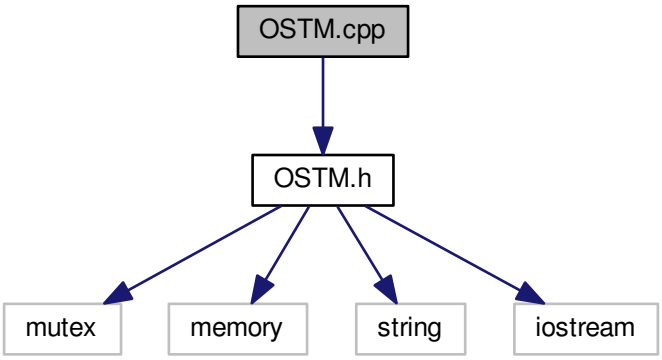
- TX.h
- TX.cpp

6 File Documentation

6.1 OSTM.cpp File Reference

`#include "OSTM.h"`

Include dependency graph for OSTM.cpp:



6.2 OSTM.cpp

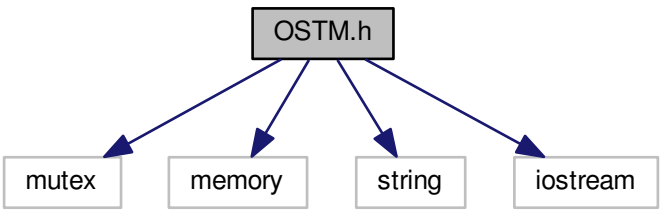
```
00001 /*
00002  * File:   OSTM.cpp
00003  * Author: Zoltan Fuzesi C00197361,
00004  * IT Carlow, Software Engineering,
00005  *
00006  * Supervisor : Joe Kehoe,
00007  *
00008  * C++ Software Transactional Memory,
00009  *
00010  * Created on December 18, 2017, 2:09 PM
00011  * OSTM base class function declarations.
00012  */
00013
00014 #include "OSTM.h"
00015
00016 int OSTM::global_Unique_ID_Number = 0;
00017
00021 OSTM::OSTM()
00022 {
00024     this->version = ZERO;
00026     this->uniqueID = Get_global_Unique_ID_Number();
00028     this->canCommit = true;
00030     this->abort_Transaction = false;
00031 }
00032
00033
00039 OSTM::OSTM(int _version_number_, int _unique_id_)
00040 {
00042     this->uniqueID = _unique_id_;
00044     this->version = _version_number_;
00046     this->canCommit = true;
00048     this->abort_Transaction = false;
00049 }
00050
00054 OSTM::~OSTM() {
00056 }
00061 int OSTM::Get_global_Unique_ID_Number() {
00063     if(global_Unique_ID_Number > 10000000)
00065         global_Unique_ID_Number = 0;
```

```
    return ++global_Unique_ID_Number;
}

void OSTM::Set_Unique_ID(int uniqueID) {
    this->uniqueID = uniqueID;
}
int OSTM::Get_Unique_ID() const
{
    return uniqueID;
}
void OSTM::Set_Version(int version)
{
    this->version = version;
}
int OSTM::Get_Version() const
{
    return version;
}
void OSTM::increase_VersionNumber()
{
    this->version += 1;
}
void OSTM::Set_Can_Commit(bool canCommit) {
    this->canCommit = canCommit;
}
bool OSTM::Is_Can_Commit() const {
    return canCommit;
}
void OSTM::Set_Abort_Transaction(bool abortTransaction) {
    this->abort_Transaction = abortTransaction;
}
bool OSTM::Is_Abort_Transaction() const {
    return abort_Transaction;
}
void OSTM::lock_Mutex() {
    this->mutex.lock();
}
void OSTM::unlock_Mutex() {
    this->mutex.unlock();
}
bool OSTM::is_Locked(){
    return this->mutex.try_lock();
}
```

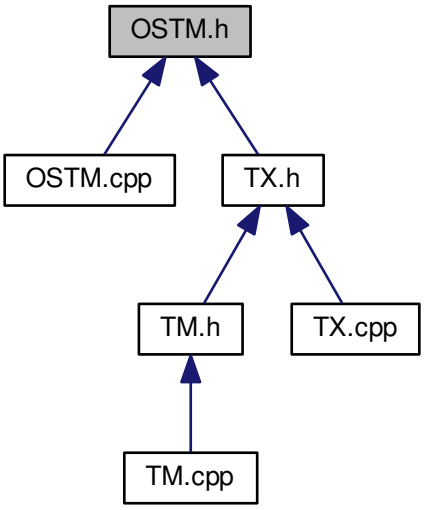
6.3 OSTM.h File Reference

```
#include <mutex>
#include <memory>
#include <string>
#include <iostream>
```

Include dependency graph for OSTM.h:



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



Classes

- class OSTM

6.4 OSTM.h

```
00001
00015 #ifndef OSTM_H
00016 #define OSTM_H
00017 #include <mutex>
00018 #include <memory>
00019 #include <string>
00020 #include <iostream>
00021 #include <string>
00022
00023 class OSTM {
00024 public:
00025     /*
00026      * Default Constructor
00027      */
00028     OSTM();
00029     /*
00030      * Custom Constructor
00031      */
00032     OSTM(int _version_number_, int _unique_id_);
00033     /*
00034      * De-constructor
00035      */
00036     virtual ~OSTM();
00041     virtual void copy(std::shared_ptr<OSTM> from, std::shared_ptr<OSTM> to){};
00046     virtual std::shared_ptr<OSTM> getBaseCopy(std::shared_ptr<OSTM> object){};//std::cout <<
       "[OSTM GETBASECOPY]" << std::endl;};
00051     virtual void toString(){};
00052     /*
00053      * Setter for object unique id
00054      * @param uniqueID Integer to set the uniqueId
00055      */
00056     void Set_Unique_ID(int uniqueID);
00057     /*
00058      * Getter for object unique id
```

```
     */
    int Get_Unique_ID() const;
    /*
     * Setter for object version number
     * @param version Integer to set the version number
     */
    void Set_Version(int version);
    /*
     * Getter for object version number
     */
    int Get_Version() const;
    /*
     * When transacion make changes on object at commit time increase the version number on the object.
     */
    void increase_VersionNumber();
    /*
     * Determin if the object can commit or not. Return boolean TRUE/FALSE
     */
    bool Is_Can_Commit() const;
    /*
     * Setter for canCommit boolean filed
     * @param canCommit Boolean to set the canCommit variable
     */
    void Set_Can_Commit(bool canCommit);
    /*
     * set boolean
     * @param abortTransaction boolean to set the abort_Transaction TRUE or FALSE
     */
    void Set_Abort_Transaction(bool abortTransaction);
    /*
     * Determin if the object need to abort the transaction or not. Return boolean TRUE/FALSE
     */
    bool Is_Abort_Transaction() const;
    /*
     * Function to lock the object itself
     */
    void lock_Mutex();
    /*
     * Function to unlock the object itself
     */
    void unlock_Mutex();
    /*
     * Function to try lock the object itself if it is not locked. Return boolean value TRUE/FALSE depending if it is can lock or not.
     */
    bool is_Locked();

private:
    int version;
    int uniqueID;
    bool canCommit;
    bool abort_Transaction;
    static int global_Unique_ID_Number;
    const int ZERO = 0;
    std::mutex mutex;
    int Get_global_Unique_ID_Number();

};

#endif /* OSTM_H */
```

6.5 README.md File Reference

6.6 README.md

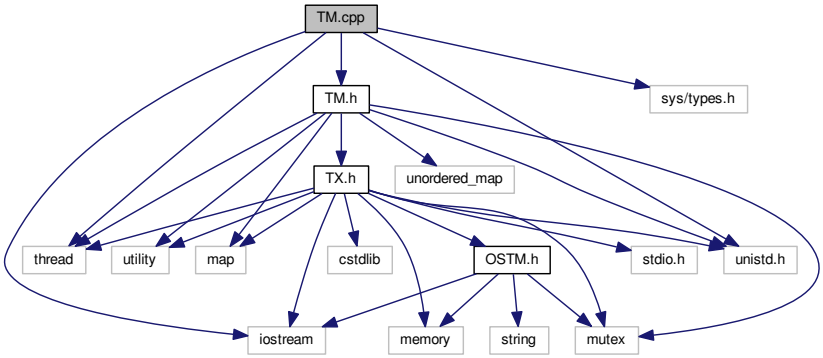
```
Usage of the STM library on Linux.<br>
In order to use the O_STM library with any C++ application, it need to be placed to the operation system /usr/lib directory.<br>
1. Copy lib_o_stm.so file to /usr/lib: sudo cp lib_o_stm.so /usr/lib
2. Include the TM.h TX.h and the OSTM.h files in your application.<br>
3. Create Makefile :
<br><br>
### Makefile.mk Documentation<br><br>
EXE =Test<br>
CC = g++<br>
PROGRAM = app<br>
CFLAGS =-std=c++14 -pthread <br>
CFILES = main.cpp AIB.cpp ULSTER.cpp BOA.cpp UNBL.cpp SWBPLC.cpp<br>
HFILES = TM.h TX.h OSTM.h AIB.h ULSTER.h BOA.h UNBL.h SWBPLC.h<br>
<br><br>
all:$(PROGRAM)<br>
```

```
<br><br>
###Rule for SHARED linking<br>
$(PROGRAM):$(CFILES) $(HFILES)<br>
   $(CC) $(CFLAGS) *.cpp -I -L /usr/lib/lib_o_stm.so  -o $(EXE)    <br>
clean:<br>
   rm -f *.o<br>
<br>
3. Run the application/executable file : ./Test
```

6.7 TM.cpp File Reference

```
#include "TM.h"
#include <thread>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
```

Include dependency graph for TM.cpp:



6.8 TM.cpp

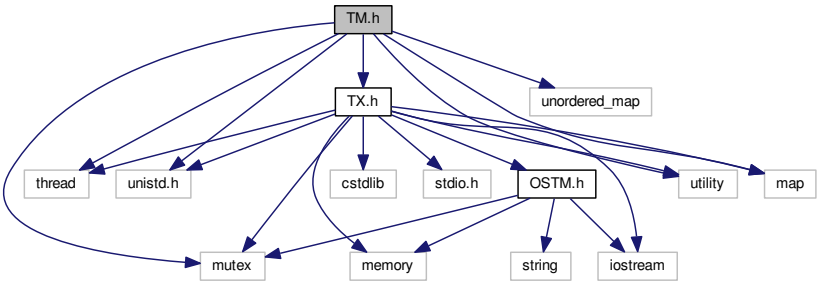
```
/*
 * File:   TM.cpp
 * Author: Zoltan Fuzesi C00197361,
 * IT Carlow, Software Engineering,
 *
 * Supervisor : Joe Kehoe,
 *
 * C++ Software Transactional Memory,
 *
 * Created on December 18, 2017, 2:09 PM
 * OSTM base class function declarations.
 */

#include "TM.h"
#include <thread>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>

/*
 * @23 _tm_id pid_t, process id determine the actual process between process in the STM library
 */
pid_t TM::_tm_id;
/*
 * @27 static Global std::map process_map_collection store all transactional objects/pointers
 */
std::map<pid_t, std::map< std::thread::id, int >> TM::process_map_collection;
TM& TM::Instance() {
```

```
    static TM _instance;
    _instance._tm_id = getpid();
    return _instance;
}

void TM::registerTX()
{
    std::lock_guard<std::mutex> guard(register_Lock);
    pid_t ppid = getppid();
    std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
TM::process_map_collection.find(ppid);
    if (process_map_collection_Iterator == TM::process_map_collection.end()) {
        std::map< std::thread::id, int >map = get_thread_Map();
        TM::process_map_collection.insert({ppid, map});

    }
    std::map<std::thread::id, std::shared_ptr < TX>>::iterator it = txMap.find(
std::this_thread::get_id());
    if (it == txMap.end()) {
        std::shared_ptr<TX> _transaction_object(new TX(std::this_thread::get_id()));
        txMap.insert({std::this_thread::get_id(), _transaction_object});
        process_map_collection_Iterator = TM::process_map_collection.find(ppid);
        process_map_collection_Iterator->second.insert({std::this_thread::get_id(), 1});
    }
}

std::shared_ptr<TX>const TM::_get_tx()
{
    std::lock_guard<std::mutex> guard(get_Lock);
    std::map<std::thread::id, std::shared_ptr<TX>>::iterator it = txMap.find(std::this_thread::get_id(
));
    if(it == txMap.end())
    {
       registerTX();
       it = txMap.find(std::this_thread::get_id());

    } else {
        it->second->_increase_tx_nesting();
    }
    return it->second;

}
void TM::_TX_EXIT(){
    TX tx(std::this_thread::get_id());
    pid_t ppid = getppid();
    std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
TM::process_map_collection.find(ppid);
    if (process_map_collection_Iterator != TM::process_map_collection.end()) {
        for (auto current = process_map_collection_Iterator->second.begin(); current !=
process_map_collection_Iterator->second.end(); ++current) {
            txMap.erase(current->first);
        }
        TM::process_map_collection.erase(ppid);
    }
    tx.ostm_exit();
}
void TM::print_all(){
    get_Lock.lock();
    for (auto current = txMap.begin(); current != txMap.end(); ++current) {
        std::cout << "KEY : " << current->first << std::endl;
    }
    get_Lock.unlock();
}

std::map< std::thread::id, int > TM::get_thread_Map() {
    std::map< std::thread::id, int > thread_Map;
    return thread_Map;
}
```

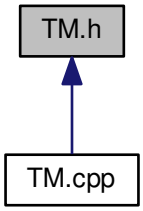
6.9 TM.h File Reference

```
#include <thread>
#include <unistd.h>
#include <mutex>
#include <unordered_map>
#include <utility>
#include <map>
#include "TX.h"
```

Include dependency graph for TM.h:



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



Classes

- class TM

6.10 TM.h

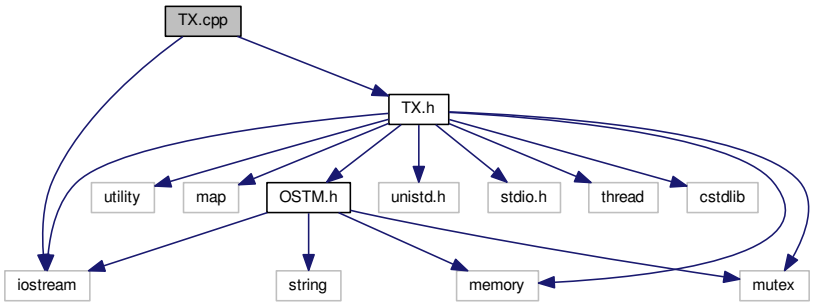
```
00001
00046 #ifndef TM_H
00047 #define TM_H
00048
00049 #include <thread>
00050 #include <unistd.h>//used for pid_t
00051 #include <mutex>
00052 #include <unordered_map>
00053 #include <utility>
00054 #include <map>
00055 #include "TX.h"
00056
00057 class TM {
00058 private:
00059     /*
00060      * TM constructor, prevent from multiple instantiation
00061      */
00062     TM() = default;
00063     /*
00064      * TM de-constructor, prevent from deletion
00065      */
00066     ~TM() = default;
```

```
    TM(const TM&) = delete;
    TM& operator=(const TM&) = delete;
    /*
     * txMap std::map, store all transactional objects created with Transaction Manager
     */
    std::map<std::thread::id, std::shared_ptr<TX>>txMap;
    /*
     * STATIC GLOBAL MAP Collection to store all process associated keys to find when deleting transactions

     */
    static std::map<pid_t, std::map< std::thread::id, int >>
process_map_collection;
    /*
     * get_thread_Map returning and map to insert to the process_map_collection as an inner value
     */
    std::map< std::thread::id, int > get_thread_Map();
    /*
     * registerTX void, register transaction into txMap
     */
    void registerTX();
    /*
     * register_Lock std::mutex, used in the registerTX function
     */
    std::mutex register_Lock;
    /*
     * register_Lock std::mutex, used in the _get_tx function
     */
    std::mutex get_Lock;
    /*
     * _tm_id pid_t, process id determine the actual process between process in the shared OSTM library
     */
    static pid_t _tm_id;

public:
    static TM& Instance();
    std::shared_ptr<TX>const _get_tx();
    void _TX_EXIT();
    void print_all();
};


#endif // TM_H
```

6.11 TX.cpp File Reference

```
#include "TX.h"
#include <iostream>
```

Include dependency graph for TX.cpp:



6.12 TX.cpp

```
/*
 * File:   TX.cpp
 * Author: Zoltan Fuzesi C00197361,
```

```
 * IT Carlow, Software Engineering,
 *
 * Supervisor : Joe Kehoe,
 *
 * C++ Software Transactional Memory,
 *
 * Created on December 18, 2017, 2:09 PM
 * OSTM base class function declarations.
 */

#include "TX.h"
#include <iostream>
/*
 * @19 main_Process_Map_collection, register static Global class level map to store all transactional objects/pointers
 */
std::map<int, std::shared_ptr<OSTM> >TX::main_Process_Map_collection;
/*
 * @23 process_map_collection, register static Global class level map to store all transaction number associated with the main process
 */
std::map<pid_t, std::map< int, int >> TX::process_map_collection;
/*
 * @27 egister_Lock, register static class level shared std:mutex to protect shared map during transaction registration
 */
std::mutex TX::register_Lock;
/*
 * @31 test_counter, register class level Integer variable to store the umber of rollback happens, for testing purposes
 */
int TX::test_counter = 0;
TX::TX(std::thread::id id) {
    this->transaction_Number = id;
    this->_tx_nesting_level = 0;
}
TX::~TX() {
}
void TX::th_exit() {
    if (this->_tx_nesting_level > 0) {
        /* Active nested transactions running in background, do not delete anything yet */
    } else {
        /* Remove all elements map entries from transaction and clear the map */
        working_Map_collection.clear();
    }
}

void TX::ostm_exit() {
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    pid_t ppid = getppid();
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator = TX::process_map_collection.find(ppid);
    if (process_map_collection_Iterator != TX::process_map_collection.end()) {
        for (auto current = process_map_collection_Iterator->second.begin(); current != process_map_collection_Iterator->second.end(); ++current) {
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(current->first);
            if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()){
                TX::main_Process_Map_collection.erase(main_Process_Map_collection_Iterator->first);
            }
        }
        TX::process_map_collection.erase(process_map_collection_Iterator->first);
    }
}

void TX::_register(std::shared_ptr<OSTM> object) {
    std::lock_guard<std::mutex> guard(TX::register_Lock);
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN REGISTER FUNCTION]") );
    }
    pid_t ppid = getppid();
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator = TX::process_map_collection.find(ppid);
    if (process_map_collection_Iterator == TX::process_map_collection.end()) {
        std::map< int, int >map =  get_thread_Map();
        TX::process_map_collection.insert({ppid, map});
        process_map_collection_Iterator = TX::process_map_collection.find(ppid);
    }
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(object->Get_Unique_ID());
    if (main_Process_Map_collection_Iterator == TX::main_Process_Map_collection.end()) {
        TX::main_Process_Map_collection.insert({object->Get_Unique_ID(), object});
        process_map_collection_Iterator->second.insert({object->Get_Unique_ID(), 1});
```

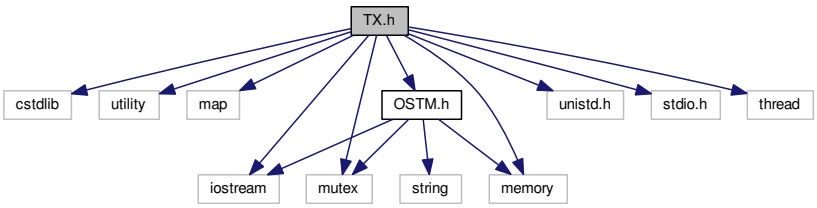
```
    }
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    if (working_Map_collection_Object_Shared_Pointer_Iterator == working_Map_collection.end()) {
        working_Map_collection.insert({object->Get_Unique_ID(), object->getBaseCopy(object)});
    }
}
std::shared_ptr<OSTM> TX::load(std::shared_ptr<OSTM> object) {
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN LOAD FUNCTION]") );
    }
    working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    if (working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end()) {
        return working_Map_collection_Object_Shared_Pointer_Iterator->second->getBaseCopy(working_Map_collection_Object_Shared_Pointer_Iterator->second);
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND LOAD FUNCTION]") );}
}
void TX::store(std::shared_ptr<OSTM> object) {
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN STORE FUNCTION]") );
    }
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    if (working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end()) {
        working_Map_collection_Object_Shared_Pointer_Iterator->second = object;
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND STORE FUNCTION, CANNOT STORE OBJECT]") );}
}
bool TX::commit() {
    bool can_Commit = true;
    if (this->_tx_nesting_level > 0) {
        _decrease_tx_nesting();
        return true;
    }
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    for (working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end(); working_Map_collection_Object_Shared_Pointer_Iterator++) {
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            if(main_Process_Map_collection_Iterator == TX::main_Process_Map_collection.end())
            {
                throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT FUNCTION]") );
            }

        while(!(main_Process_Map_collection_Iterator->second)->is_Locked());
        if (main_Process_Map_collection_Iterator->second->Get_Version() > working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Version()) {
            working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(false);
            can_Commit = false;
            break;
        } else {
            working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(true);
        }
    }
    if (!can_Commit) {
        for (working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end(); working_Map_collection_Object_Shared_Pointer_Iterator++) {
            main_Process_Map_collection_Iterator  = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            (working_Map_collection_Object_Shared_Pointer_Iterator->second)->copy(working_Map_collection_Object_Shared_Pointer_Iterator->second, main_Process_Map_collection_Iterator->second);
        }
        _release_object_lock();
        return false;
    } else {
        for (working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end(); working_Map_collection_Object_Shared_Pointer_Iterator++) {
                main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find((
```

```
working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
                if (main_Process_Map_collection_Iterator !=
TX::main_Process_Map_collection.end()) {
                    (main_Process_Map_collection_Iterator->second)->copy(
main_Process_Map_collection_Iterator->second, working_Map_collection_Object_Shared_Pointer_Iterator->second);
                    main_Process_Map_collection_Iterator->second->increase_VersionNumber();
                } else { throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT
 FUNCTION]")); }
        }
        _release_object_lock();
        this->th_exit();
        return true;
    }
}//Commit finish

void TX::_release_object_lock(){
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    for (working_Map_collection_Object_Shared_Pointer_Iterator =
working_Map_collection.begin(); working_Map_collection_Object_Shared_Pointer_Iterator
 != working_Map_collection.end();
working_Map_collection_Object_Shared_Pointer_Iterator++) {
            main_Process_Map_collection_Iterator =
TX::main_Process_Map_collection.find((
working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
            if (main_Process_Map_collection_Iterator !=
TX::main_Process_Map_collection.end()) {
                (main_Process_Map_collection_Iterator)->second->unlock_Mutex();
            }
        }
}

void TX::_increase_tx_nesting() {
    this->_tx_nesting_level += 1;
}
void TX::_decrease_tx_nesting() {
    this->_tx_nesting_level -= 1;
;
}
int TX::getTest_counter() {
    return TX::test_counter;
}
const std::thread::id TX::_get_tx_number() const {
    return transaction_Number;
}
std::map< int, int > TX::get_thread_Map() {
    std::map< int, int > thread_Map;
    return thread_Map;
}

void TX::_print_all_tx() {
    std::map< int, std::shared_ptr<OSTM> >::iterator it;
    pid_t ppid = getppid();
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
TX::process_map_collection.find(ppid);
    if (process_map_collection_Iterator != TX::process_map_collection.end()) {
        for (auto current = process_map_collection_Iterator->second.begin(); current !=
process_map_collection_Iterator->second.end(); ++current) {
            it = working_Map_collection.find(current->first);
            if(it != working_Map_collection.end()){
                std::cout << "[Unique number ] : " <<it->second->Get_Unique_ID() << std::endl;
            }
        }
    }
}
```

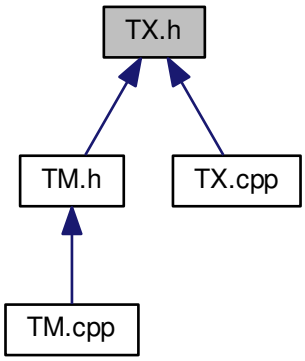
6.13 TX.h File Reference

```
#include <cstdlib>
#include <utility>
#include <map>
#include <iostream>
#include <mutex>
#include <unistd.h>
#include <memory>
#include <stdio.h>
#include <thread>
#include "OSTM.h"
```

Include dependency graph for TX.h:



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



Classes

- class TX

6.14 TX.h

```
00001 /*
00002  * File:   TX.h
00003  * Author: Zoltan Fuzesi C00197361,
00004  * IT Carlow, Software Engineering,
00005  *
00006  * Supervisor : Joe Kehoe,
00007  *
00008  * C++ Software Transactional Memory,
00009  *
00010  * Created on December 18, 2017, 2:09 PM
00011  * OSTM base class function declarations.
00012  */
00013
00014 #ifndef TX_H
00015 #define TX_H
00016 #include <cstdlib>
00017 #include <utility>
00018 #include <map>
00019 #include <iostream>
```

```
#include <mutex>
#include <unistd.h>
#include <memory>
#include <stdio.h>
#include <thread>
#include "OSTM.h"

class TM;

class TX {
public:
    /*
     * Custom Constructor
     */
    TX(std::thread::id id);
    /*
     * De-constructor
     */
    ~TX();
    /*
     * Default copy constructor
     */
    TX(const TX& orig);
    /*
     * Delete all map entries associated with the main process
     */
    void ostm_exit();
    /*
     *  Register OSTM pointer into STM library
     */
    void _register(std::shared_ptr<OSTM>  object);
    /*
     * Load a copy of OSTM shared pointer to main process
     */
    std::shared_ptr<OSTM>  load(std::shared_ptr<OSTM>  object);
    /*
     * Store transactional changes
     */
    void store(std::shared_ptr<OSTM>  object);
    /*
     * Commit transactional changes
     */
    bool commit();
    /*
     * Increase TX (Transaction) nesting level by one
     */
    void _increase_tx_nesting();
    /*
     * Decrease TX (transaction) nesting level by one
     */
    void _decrease_tx_nesting();
    /*
     * Only TM Transaction Manager can create instance of TX Transaction
     */
    friend class TM;
    /*
     * ONLY FOR TESTING!!! returning the number of rollback happened during transactions
     */
    int getTest_counter();
    /*
     * test_counter int ONLY FOR TESTING!!! store number of rollbacks
     */
    static int test_counter;
    /*
     * TESTING ONLY print all transactions
     */
    void _print_all_tx() ;


private:
    std::map< int, std::shared_ptr<OSTM>  > working_Map_collection;
    std::thread::id transaction_Number;
    int _tx_nesting_level;
    static std::map<int, std::shared_ptr<OSTM>  >main_Process_Map_collection;
    static std::map<pid_t, std::map< int, int >> process_map_collection;
    std::map< int , int > get_thread_Map();
    static std::mutex register_Lock;
    const std::thread::id _get_tx_number() const;
    void _release_object_lock();
    void th_exit();

};
#endif // _TX_H_
```