Software Transactional Memory

C++14 STM

Technical Manual

Zoltan Fuzesi

April 16, 2018

Student ID: C00197361
Supervisor: Joseph Kehoe
Institute of Technology Carlow
Software Engineering

Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

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 Installation of the Shared library on Linux platform.

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.

Go to the website `Tutorial` and download the library to the required operating system platform. (Linux, Windows, Mac OSX)

1.1.3 Step 2: Unzip in to the target destination.

Unzip the downloaded rar file. You can find the Shared, Static library and the ∗.h files in the unzipped folder. Copy the ∗.h files to the same folder where is the other C++ files are stored.

1.1.4 Step 3: Copy the shared library.

The Shared library is a libostm.so file, that you need copy to the operating system directory 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.

To achieve the required class hierachy between the OSTM library and your own class structure, you need to implement few steps to inherite from the OSTM base class. Go to website `Tutorial` for more details. 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.

You can create an executable file using the provided Makefile as you linking together the library (libostm.so), and the ∗.h files with your own files.

1.1.7 Step 6: Transactional Environment.

Now your application use transactional environment, that guarantees the consistency between object transactions.

1.1.8 Step 7: Run the application.

Go to the directory where the executable was created, and used the following line in the terminal to run the application : ./EXECUTABLE_NAME

2 Class Index

2.1 Class List

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

3 File Index

3.1 File List

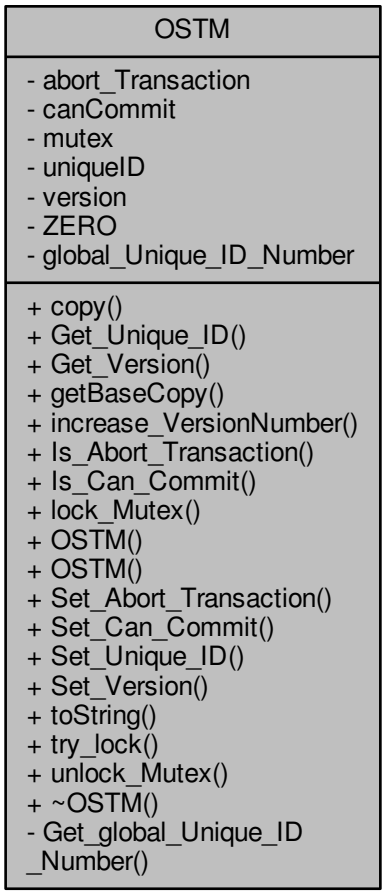
Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 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.
- 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.
- bool try_lock ()

  @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 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

4.1.1 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.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 OSTM::OSTM ( )

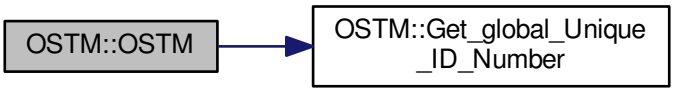
@21 Default constructor

Definition at line 21 of file OSTM.cpp.

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

```
00022 {
00023     /* @24 Integer field <version> indicates the version number of the inherited child object */
00024     this->version = ZERO;
00025     /* @26 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM
      library */
00026     this->uniqueID = Get_global_Unique_ID_Number();
00027     /* @28 Boolean value <canCommit> to determine the object can or cannot commit */
00028     this->canCommit = true;
00029     /* @30 Boolean field <abort_Transaction> to determine the object can or cannot commit */
00030     this->abort_Transaction = false;
00031 }
```

Here is the call graph for this function:



4.1.2.2 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

Definition at line 39 of file OSTM.cpp.

References abort_Transaction, canCommit, uniqueID, and version.

```
00040 {
00041     /* @42 Integer field <version> indicates the version number of the inherited child object */
00042     this->uniqueID = _unique_id_;
00043     /* @44 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM
       library */
00044     this->version = _version_number_;
00045     /* @46 Boolean value <canCommit> to determine the object can or cannot commit */
00046     this->canCommit = true;
00047     /* @48 Boolean value <abort_Transaction> to determine the object can or cannot commit */
00048     this->abort_Transaction = false;
00049 }
```

4.1.2.3 OSTM::~OSTM ( ) [virtual]

@54 Default De-constructor

Definition at line 54 of file OSTM.cpp.

```
00054             {
00055     /* Destroy the object. */
00056 }
```

4.1.3 Member Function Documentation

4.1.3.1 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 46 of file OSTM.h.

```
00046 {};
```

4.1.3.2 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

Definition at line 61 of file OSTM.cpp.

References global_Unique_ID_Number.

Referenced by OSTM().

```
00061                                     {
00062     /* @64 Checking the global_Unique_ID_Number */
00063     if(global_Unique_ID_Number > 10000000)
00064         /* @65 Reset global_Unique_ID_Number to ZERO*/
00065         global_Unique_ID_Number = 0;
00066     /* @67 return static global_Unique_ID_Number */
00067     return ++global_Unique_ID_Number;
00068 }
```

4.1.3.3 int OSTM::Get_Unique_ID ( ) const

@82 Function <Get_Unique_ID> getter for uniqueID private field

Definition at line 82 of file OSTM.cpp.

References uniqueID.

Referenced by getBaseCopy().

```
00083 {
00084     /* @85 return Object uniqueID */
00085     return uniqueID;
00086 }
```

4.1.3.4 int OSTM::Get_Version ( ) const

@100 Function <Get_Version> setter for version private filed

Definition at line 100 of file OSTM.cpp.

References version.

Referenced by getBaseCopy().

```
00101 {
00102     /* return object version number */
00103     return version;
00104 }
```

4.1.3.5 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

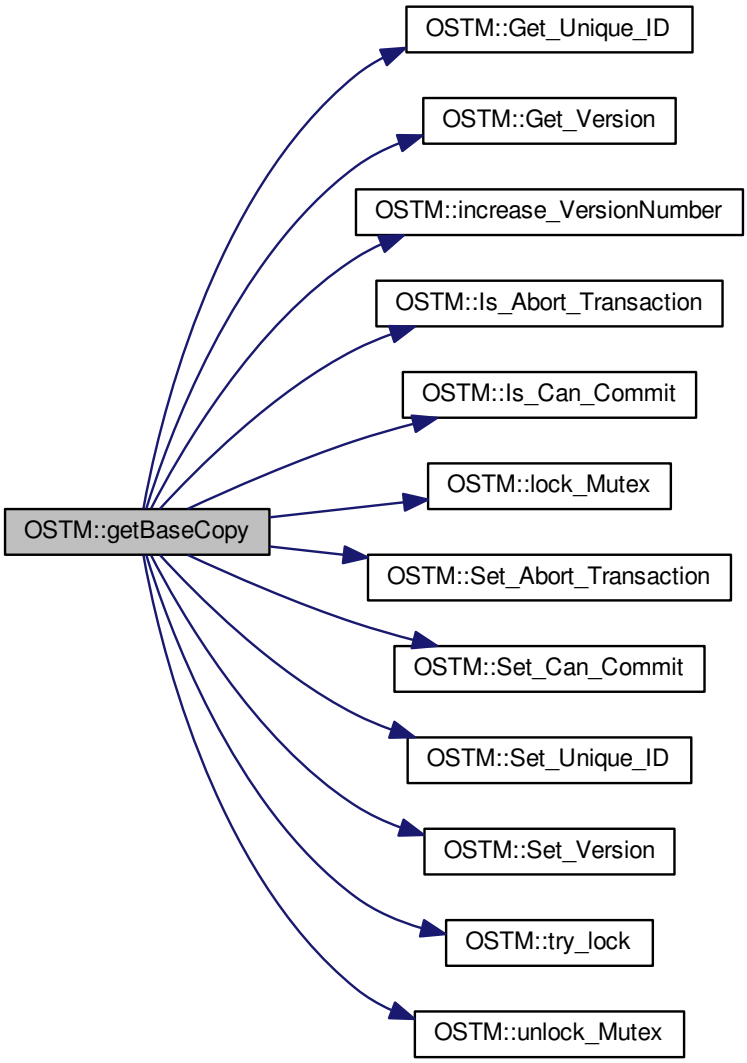
getBaseCopy function implementation in child class

Definition at line 51 of file OSTM.h.

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

```
00051 {};
```

Here is the call graph for this function:



4.1.3.6 void OSTM::increase_VersionNumber ( )

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

Definition at line 108 of file OSTM.cpp.

References version.

Referenced by getBaseCopy().

```
00109 {
00110     /* @111 increase object version number */
00111     this->version += 1;
00112 }
```

4.1.3.7 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

Definition at line 140 of file OSTM.cpp.

References abort_Transaction.

Referenced by getBaseCopy().

```
00140                                       {
00141     /* @142 return abort_Transaction object boolean value */
00142     return abort_Transaction;
00143 }
```

4.1.3.8 bool OSTM::Is_Can_Commit ( ) const

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

Definition at line 124 of file OSTM.cpp.

References canCommit.

Referenced by getBaseCopy().

```
00124                                 {
00125     /* @126 return canCommit boolean value TRUE/FALSE */
00126     return canCommit;
00127 }
```

4.1.3.9 void OSTM::lock_Mutex ( )

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

Definition at line 147 of file OSTM.cpp.

References mutex.

Referenced by getBaseCopy().

```
00147                       {
00148     /* @149 Locking the mutex*/
00149     this->mutex.lock();
00150 }
```

4.1.3.10 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

Definition at line 132 of file OSTM.cpp.

References abort_Transaction.

Referenced by getBaseCopy().

```
00132                                                    {
00133     /* @134 set abort_Transaction object variable to parameter boolean value  */
00134     this->abort_Transaction = abortTransaction;
00135 }
```

4.1.3.11 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

Definition at line 117 of file OSTM.cpp.

References canCommit.

Referenced by getBaseCopy().

```
00117                                        {
00118     /* @119 set canCommit object variable to parameter boolean value*/
00119     this->canCommit = canCommit;
00120 }
```

4.1.3.12 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.

Definition at line 75 of file OSTM.cpp.

References uniqueID.

Referenced by getBaseCopy().

```
00075                                       {
00076     /* @77 set object uniqueID to parameter integer value */
00077     this->uniqueID = uniqueID;
00078 }
```

4.1.3.13 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.

Definition at line 92 of file OSTM.cpp.

References version.

Referenced by getBaseCopy().

```
00093 {
00094     /* @95 set object version to parameter integer value */
00095     this->version = version;
00096 }
```

4.1.3.14 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 41 of file OSTM.h.

```
00041 {};
```

4.1.3.15 bool OSTM::try_lock ( )

@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.

Definition at line 162 of file OSTM.cpp.

References mutex.

Referenced by getBaseCopy().

```
00162                    {
00163     /* @164 Try to unlock the mutex, return TRUE if the lock was acquired successfully, otherwise return
        FALSE */
00164     return this->mutex.try_lock();
00165 }
```

4.1.3.16 void OSTM::unlock_Mutex ( )

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

Definition at line 154 of file OSTM.cpp.

References mutex.

Referenced by getBaseCopy().

```
00154                        {
00155     /* @156 Locking the mutex */
00156     this->mutex.unlock();
00157 }
```

4.1.4 Member Data Documentation

4.1.4.1 bool OSTM::abort_Transaction [private]

Boolean value <abort_Transaction> to determine the object need to abort the transaction

Definition at line 125 of file OSTM.h.

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

4.1.4.2 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 getBaseCopy(), Is_Can_Commit(), OSTM(), and Set_Can_Commit().

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

Unique object number start at ZERO The value stored in 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().

4.1.4.4 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 lock_Mutex(), try_lock(), and unlock_Mutex().

4.1.4.5 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(), getBaseCopy(), OSTM(), and Set_Unique_ID().

4.1.4.6 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(), getBaseCopy(), increase_VersionNumber(), OSTM(), and Set_Version().

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

Integer <ZERO> meaninful string equalient to 0

Definition at line 134 of file OSTM.h.

Referenced by OSTM().

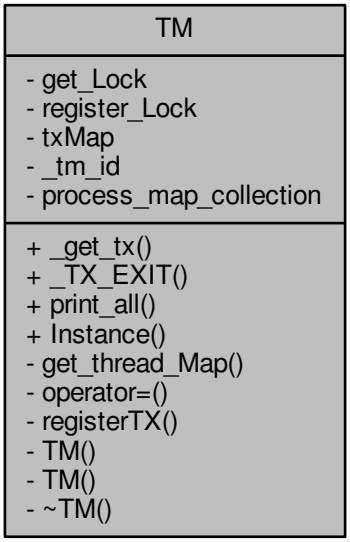
The documentation for this class was generated from the following files:

- OSTM.h
- OSTM.cpp

4.2 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

4.2.1 Detailed Description

Definition at line 70 of file TM.h.

4.2.2 Constructor & Destructor Documentation

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

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

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

TM copy constructor, prevent from copying the Transaction Manager.

4.2.3 Member Function Documentation

4.2.3.1 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.

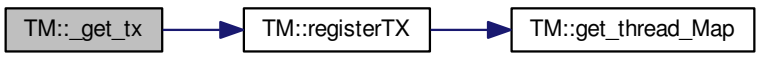
_get_tx std::shared_ptr<TX>, returning a shared pointer transaction object

Definition at line 81 of file TM.cpp.

References get_Lock, registerTX(), and txMap.

```
00082 {
00083
00084     /* @85 guard std::lock_guard, locks the get_Lock mutex, unlock automatically when goes out of the scope
       get_Lock std::mutex, used by the lock_guard to protect txMap from race conditions */
00085     std::lock_guard<std::mutex> guard(get_Lock);
00086     /* @87 txMap try to find the TX Transaction object by it's actual thread ID if registred in the txMap
      */
00087     std::map<std::thread::id, std::shared_ptr<TX>>::iterator it = txMap.find(std::this_thread::get_id(
      ));
00088     /* @89 Check if iterator pointing to the end of the txMap then insert */
00089     if(it == txMap.end())
00090     {
00091       /* @92 If cannot find then call the register function to register the thread with a transaction */
00092        registerTX();
00093        /* @94 If it's registered first time then we need to find it after registration */
00094        it = txMap.find(std::this_thread::get_id());
00095
00096     } else {
00097         /* @98 If transaction already registered, it means the thread participating in nested transactions,
       and increase the nesting */
00098         it->second->_increase_tx_nesting();
00099     }
00100     /* @101 Returning back the transaction (TX) object to the thread */
00101     return it->second;
00102
00103 }
```

Here is the call graph for this function:



4.2.3.2 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

Definition at line 108 of file TM.cpp.

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

```
00108                   {
00109     /*  @110 Transaction manger create a local Transaction Object to access the TX class function without
       nesting any transaction */
00110     TX tx(std::this_thread::get_id());
00111     /* @112 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00112     pid_t ppid = getppid();
00113     /* @114 process_map_collection try to find the main process by it's ppid if registred in the library */
00114     std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
      TM::process_map_collection.find(ppid);
00115     /* @116 Check if iterator NOT pointing to the end of the process map then register */
00116     if (process_map_collection_Iterator != TM::process_map_collection.end()) {
00117         /* @118 Iterate through the process_map_collection to find all transaction associated with main
       process */
00118         for (auto current = process_map_collection_Iterator->second.begin(); current !=
      process_map_collection_Iterator->second.end(); ++current) {
00119             /* @120 Delete all transaction associated with the actual main process */
00120             txMap.erase(current->first);
00121         }
00122         /* @123 When all transaction deleted, delete the main process from the Transacion Manager */
00123         TM::process_map_collection.erase(ppid);
00124     }
00125     /* @126 TX class delete all Global Object shared between the transaction. This function calls only when
       the main process exists to clear out memory */
00126     tx.ostm_exit();
00127 }
```

Here is the call graph for this function:

4.2.3.3 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

Definition at line 148 of file TM.cpp.

Referenced by registerTX().

```
00148                                                   {
00149     /*  @150 thread_Map std::map< int, int > create a map to store int key and int value */
00150     std::map< std::thread::id, int > thread_Map;
00151     /* @152 return the map */
00152     return thread_Map;
00153 }
```

4.2.3.4 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.

Definition at line 31 of file TM.cpp.

References _tm_id.

```
00031                   {
00032    /* @33 _instance TM, static class reference to the instance of the Transaction Manager class */
00033     static TM _instance;
00034     /* @35 _instance ppid, assigning the process id whoever created the Singleton instance */
00035     _instance._tm_id = getpid();
00036     /* @37 return Singleton instance */
00037     return _instance;
00038 }
```

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

TM copy operator, prevent from copying the Transaction Manager.

4.2.3.6 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

Definition at line 132 of file TM.cpp.

References get_Lock, and txMap.

```
00132                    {
00133     /* @134 Locking the print function */
00134     get_Lock.lock();
00135     /* @136 Iterate through the txMap to print out the thread id's*/
00136     for (auto current = txMap.begin(); current != txMap.end(); ++current) {
00137         /* @138 Print key (thread number)*/
00138         std::cout << "KEY : " << current->first << std::endl;
00139     }
00140     /* @140 Unlocking the print function*/
00141     get_Lock.unlock();
00142 }
```

4.2.3.7 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.

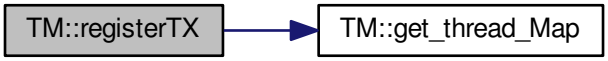
Definition at line 45 of file TM.cpp.

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

Referenced by _get_tx().

```
00046 {
00047   /* @49 guard std::lock_guard, locks the register_Lock mutex, unlock automatically when goes out of the
       scope register_Lock std::mutex, used by the lock_guard to protect shared map from race conditions */
00048     std::lock_guard<std::mutex> guard(register_Lock);
00049     /* @51 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00050     pid_t ppid = getppid();
00051
00052     /* @53 process_map_collection try to find the main process by it's ppid if registred in the library */
00053     std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
      TM::process_map_collection.find(ppid);
00054     /* @55 Check if iterator pointing to the end of the process map then register */
00055     if (process_map_collection_Iterator == TM::process_map_collection.end()) {
00056         /* @57 Require new map to insert to the process map as a value by the ppid key  */
00057         std::map< std::thread::id, int >map = get_thread_Map();
00058         /* @59 Register main process/application to the global map  */
00059         TM::process_map_collection.insert({ppid, map});
00060
00061     }
00062     /* @63 txMap std::map, collection to store all transaction created by the Transaction Manager */
00063     std::map<std::thread::id, std::shared_ptr < TX>>::iterator it = txMap.find(
      std::this_thread::get_id());
00064     /* @65 Check if iterator pointing to the end of the txMap then insert */
00065     if (it == txMap.end()) {
00066         /* @67 Create a new Transaction Object as a shared pointer */
00067         std::shared_ptr<TX> _transaction_object(new TX(std::this_thread::get_id()));
00068         /* @69 txMap insert the new transaction into the txMap by the threadID key */
00069         txMap.insert({std::this_thread::get_id(), _transaction_object});
00070         /* @71 Get the map if the transaction registered first time */
00071         process_map_collection_Iterator = TM::process_map_collection.find(ppid);
00072         /* @73 Insert to the GLOBAL MAP as a helper to clean up at end of main process. The value 1 is not
       used yet */
00073         process_map_collection_Iterator->second.insert({std::this_thread::get_id(), 1});
00074     }
00075 }
```

Here is the call graph for this function:



4.2.4 Member Data Documentation

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

Definition at line 115 of file TM.h.

Referenced by Instance().

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

Definition at line 111 of file TM.h.

Referenced by _get_tx(), and print_all().

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

Definition at line 95 of file TM.h.

Referenced by _TX_EXIT(), and registerTX().

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

Definition at line 107 of file TM.h.

Referenced by registerTX().

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

Definition at line 91 of file TM.h.

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

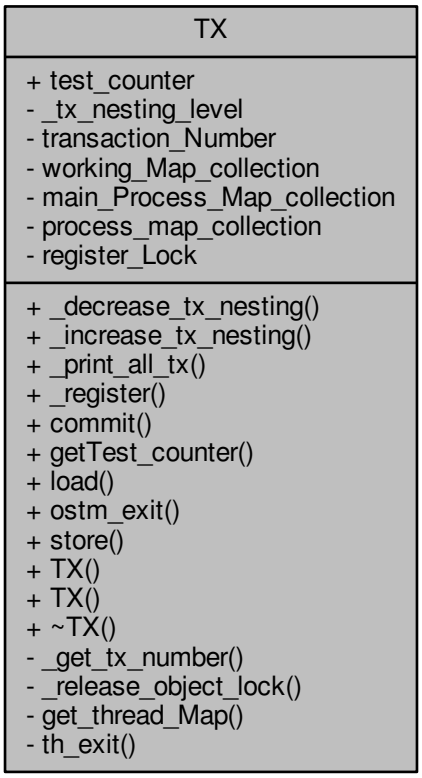
The documentation for this class was generated from the following files:

- TM.h
- TM.cpp

4.3 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

4.3.1 Detailed Description

Definition at line 29 of file TX.h.

4.3.2 Constructor & Destructor Documentation

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

@36 Custom Constructor

Parameters

id	std::thread::id, represent the transaction number to the Transaction Manager

Definition at line 36 of file TX.cpp.

References _tx_nesting_level, and transaction_Number.

```
00036                       {
00037     /* @38 Integer field <transaction_Number> indicates the transaction number to the Transaction manager
      */
00038     this->transaction_Number = id;
00039     /* @40 Integer field <_tx_nesting_level> indicates the nesting level to the transaction itself */
00040     this->_tx_nesting_level = 0;
00041 }
```

4.3.2.2 TX::~TX ( )

@45 De-constructor

Definition at line 45 of file TX.cpp.

```
00045         {
00046     /* Destroy the object. */
00047 }
```

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

4.3.3 Member Function Documentation

4.3.3.1 void TX::_decrease_tx_nesting ( )

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

Definition at line 279 of file TX.cpp.

References _tx_nesting_level.

Referenced by commit().

```
00279                                {
00280     /* @281 Decrease transaction nesting level */
00281     this->_tx_nesting_level -= 1;
00282 ;
00283 }
```

4.3.3.2 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

Definition at line 294 of file TX.cpp.

References transaction_Number.

```
00294                                          {
00295     /* @296 Return the transaction nuber */
00296     return transaction_Number;
00297 }
```

4.3.3.3 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

Definition at line 272 of file TX.cpp.

References _tx_nesting_level.

```
00272                                {
00273     /* @274 Increase transaction nesting level */
00274     this->_tx_nesting_level += 1;
00275 }
```

4.3.3.4 void TX::_print_all_tx ( )

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

Definition at line 311 of file TX.cpp.

References process_map_collection, and working_Map_collection.

```
00311                        {
00312     /* @313 initialise Iterator */
00313     std::map< int, std::shared_ptr<OSTM> >::iterator it;
00314     /* @315 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00315     pid_t ppid = getppid();
00316     /* '317 initialize and assign Iterator to process_map_collection, by the main process id (ppid) */
00317     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00318     /* @319 If there is an entry associated with the process then print out all transactions. */
00319     if (process_map_collection_Iterator != TX::process_map_collection.end()) {
00320         /* @321 Iterate through process_map_collection*/
00321         for (auto current = process_map_collection_Iterator->second.begin(); current !=
      process_map_collection_Iterator->second.end(); ++current) {
00322             /* @323 Assign value to iterator */
00323             it = working_Map_collection.find(current->first);
00324             /* @325 If value found, then print it */
00325             if(it != working_Map_collection.end()){
00326                 /* @327 print out the transaction number */
00327                 std::cout << "[Unique number ] : " <<it->second->Get_Unique_ID() << std::endl;
00328             }
00329         }
00330     }
00331 }
```

4.3.3.5 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

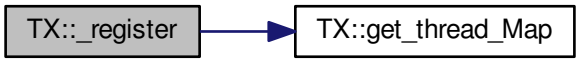
object	std::shared_ptr<OSTM>, is an original shared pointer point to the object memory space

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                                                {
00097     /* @98 register_Lock(mutex) shared lock between all transaction. MUST USE SHARED LOCK TO PROTECT SHARED
       GLOBAL MAP/COLLECTION */
00098     std::lock_guard<std::mutex> guard(TX::register_Lock);
00099     /* @100 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
00100     if(object == nullptr){
00101         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN REGISTER FUNCTION]") );
00102     }
00103     /* @104 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00104     pid_t ppid = getppid();
00105     /* @106 Declare and initialize Iterator for process_map_collection, find main process*/
00106     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00107     /* @108 If iterator cannot find main process, then register*/
00108     if (process_map_collection_Iterator == TX::process_map_collection.end()) {
00109         /* @110 Create new empty map */
00110         std::map< int, int >map =  get_thread_Map();
00111         /* @112 Register main process/application to the global map */
00112         TX::process_map_collection.insert({ppid, map});
00113         /* @114 Get the map if registered first time */
00114         process_map_collection_Iterator = TX::process_map_collection.find(ppid);
00115     }
00116     /* @117 Declare and initialize Iterator for main_Process_Map_collection, find by original object */
00117     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find(object->Get_Unique_ID());
00118     /* @119 If object cannot find, then register */
00119     if (main_Process_Map_collection_Iterator == TX::main_Process_Map_collection
      .end()) {
00120         /* '121 Insert the origin object to the GLOBAL MAP shared between transactions */
00121         TX::main_Process_Map_collection.insert({object->Get_Unique_ID(),
      object});
00122         /* @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 */
00123         process_map_collection_Iterator->second.insert({object->Get_Unique_ID(), 1});
00124     }
00125     /* @126 Declare and initialize Iterator for working_Map_collection, find copy of the original object */
00126     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator
      = working_Map_collection.find(object->Get_Unique_ID());
00127     /* @128 If copy of the object not found, then register */
00128     if (working_Map_collection_Object_Shared_Pointer_Iterator ==
      working_Map_collection.end()) {
00129         /* @130 Register transaction own copy of the original object */
00130         working_Map_collection.insert({object->Get_Unique_ID(), object->getBaseCopy(
      object)});
00131     }
00132 }
```

Here is the call graph for this function:



4.3.3.6 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

Definition at line 253 of file TX.cpp.

References main_Process_Map_collection, and working_Map_collection.

Referenced by commit().

```
                                {
    /* @255 Declare Iterator for working_Map_collection */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    /* @255 Declare Iterator for working_Map_collection */
    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++) {
            /* @260 Find Global shared original object by the transaction object unique ID*/
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find((working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
            /* @262 If object found, then release lock*/
            if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()) {
                /* @264 Release object lock */
                (main_Process_Map_collection_Iterator)->second->unlock_Mutex();
            }
        }
}
```

4.3.3.7 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.

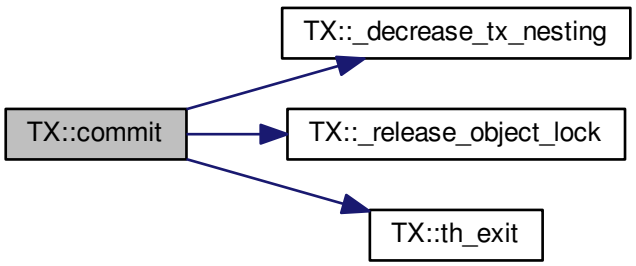
Definition at line 177 of file TX.cpp.

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

```
                {
    /* @179 Declare can_Commit boolean variable */
    bool can_Commit = true;
    /* @182 Dealing with nested transactions first. if nesting level bigger than ZERO do not commit yet */
    if (this->_tx_nesting_level > 0) {
        /* @183 Decrease nesting level @see _decrease_tx_nesting() */
        _decrease_tx_nesting();
        return true;
    }
    /* @187 Declare and initialize Iterator for working_Map_collection */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    /* @189 Declare and initialize Iterator for main_Process_Map_collectio */
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    /* @191 Iterate through the working_Map_collection, for all associated copy objetcs */
    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++) {
            /* @193 Find the Original object in the Shared global colection by the copy object unique ID */
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            /* @195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!! */
            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]"));
            }

        /* @200 Busy waiting WHILE try_lock function return false, If the object locked by another transaction, then waith until it's get unlocked, then lock it */
        while(!(main_Process_Map_collection_Iterator->second)->try_lock());
        /* @203 Compare the original global object version number with the working object version number. If the version number not same, then it cannot coomit*/
        if (main_Process_Map_collection_Iterator->second->Get_Version() > working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Version()) {
            /* @2005 Set object boolean value to FALSE, cannot commit */
            working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(false);
            /* @207 Set canCommit false Indicate rollback must happen */
            can_Commit = false;
            break;
        } else {
            /* @210 If version number are has same value set object boolean value to TRUE*/
            working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(true);
        }
```

```
    }
    /* @214 IF can_Commit boolean value setted for FALSE then rollback all copy object in the transaction to the Global object values*/
    if (!can_Commit) {
        /* @217 iterate through all transaction copy objects one by one */
        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++) {
            /* @219 Find the Global shared object by the transaction copy object unique ID */
            main_Process_Map_collection_Iterator  = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            /* @221 Copy all Global shared original objects changed values by another transaction to the transaction copy objetcs */
            (working_Map_collection_Object_Shared_Pointer_Iterator->second)->copy(working_Map_collection_Object_Shared_Pointer_Iterator->second, main_Process_Map_collection_Iterator->second);
        }
        /* @224 When the transaction finish to change copying all values from original objects to local copy, then release all Global shared objects. @see _release_object_lock() */
        _release_object_lock();
        /* @226 Return FALSE to indicate the transaction must restart !*/
        return false;
    } else {
        /* @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 */
        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++) {
                /* @231 Find the Global shared object by the transaction copy object unique ID */
                main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find((working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
                /* @233 If Global shared object found then commit changes */
                if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()) {
                    /* @235 Copy over local transaction object values to original Global object*/
                    (main_Process_Map_collection_Iterator->second)->copy(main_Process_Map_collection_Iterator->second, working_Map_collection_Object_Shared_Pointer_Iterator->second);
                    /* @237 Increase the version number in the original pointer*/
                    main_Process_Map_collection_Iterator->second->increase_VersionNumber();
                /* @195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!! */
                } else { throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT FUNCTION]")); }
        }
        /* @242 When the transaction finish with commit all changes, then release all Global shared objects. @see _release_object_lock() */
        _release_object_lock();
        /* @244 Transaction object clean up all associated values, clean memory. @see th_exit()*/
        this->th_exit();
        /* @246 Return TRUE, indicate the transaction has finished. */
        return true;
    }
}
```

Here is the call graph for this function:



4.3.3.8 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

Definition at line 301 of file TX.cpp.

Referenced by _register().

```
00301                                       {
00302     /* @303 initialize empty map hold int key and values*/
00303     std::map< int, int > thread_Map;
00304     /* @305 Return the map*/
00305     return thread_Map;
00306 }
```

4.3.3.9 int TX::getTest_counter ( )

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

Definition at line 287 of file TX.cpp.

References test_counter.

```
00287                           {
00288     /* @289 return class level value hold by test_counter variable */
00289     return TX::test_counter;
00290 }
```

4.3.3.10 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

Definition at line 137 of file TX.cpp.

References working_Map_collection.

```
00137                                                       {
00138     /* @139 Declare and initialize Iterator for working_Map_collection */
00139     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00140     /* @141 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
00141     if(object == nullptr){
00142         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN LOAD FUNCTION]") );
00143     }
00144     /* @145 Find copy object in working_Map_collection by the object unique ID*/
00145     working_Map_collection_Object_Shared_Pointer_Iterator =
      working_Map_collection.find(object->Get_Unique_ID());
00146     /* @147 If object found, then return it */
00147     if (working_Map_collection_Object_Shared_Pointer_Iterator !=
      working_Map_collection.end()) {
```

```
        /* @149 Returning a copy of the working copy object */
        return working_Map_collection_Object_Shared_Pointer_Iterator->second->getBaseCopy(working_Map_collection_Object_Shared_Pointer_Iterator->second);
    /* @151 If no object found, throw runtime error */
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND LOAD FUNCTION]") );}
}
```

4.3.3.11 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!!!!

Definition at line 68 of file TX.cpp.

References main_Process_Map_collection, and process_map_collection.

Referenced by TM::_TX_EXIT().

```
                    {
    /* @70 Declare Iterator main_Process_Map_collection_Iterator */
    std::map<int, std::shared_ptr<OSTM>>::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 */
    pid_t ppid = getppid();
    /* @74 process_map_collection try to find the main process by it's ppid if registred in the library */
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator = TX::process_map_collection.find(ppid);
    /* @76 Check if iterator NOT pointing to the end of the process_map_collection then remove all associated elements */
    if (process_map_collection_Iterator != TX::process_map_collection.end()) {
        /* @78 Iterate through the process_map_collection to find all transaction associated with main process */
        for (auto current = process_map_collection_Iterator->second.begin(); current != process_map_collection_Iterator->second.end(); ++current) {
            /* @80 Find the OSTM object in the Global shared map */
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(current->first);
            /* @82 If object found then delete it*/
            if (main_Process_Map_collection_Iterator != TX::main_Process_Map_collection.end()){
                /* @84 Delete element from shared main_Process_Map_collection by object by the unique key, and the shaed_ptr will destroy automatically */
                TX::main_Process_Map_collection.erase(main_Process_Map_collection_Iterator->first);
            }
        }
        /* @88 Delete main process from Process_map_collection */
        TX::process_map_collection.erase(process_map_collection_Iterator->first);
    }
}
```

4.3.3.12 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

Definition at line 157 of file TX.cpp.

References working_Map_collection.

```
00157                                                {
00158     /* @159 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
00159     if(object == nullptr){
00160         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN STORE FUNCTION]") );
00161     }
00162     /* @163 Declare and initialize Iterator for working_Map_collection */
00163     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00164     /* @165 Find copy object in working_Map_collection by the object unique ID*/
00165     working_Map_collection_Object_Shared_Pointer_Iterator =
      working_Map_collection.find(object->Get_Unique_ID());
00166     /* @167 If object found, then replace it */
00167     if (working_Map_collection_Object_Shared_Pointer_Iterator !=
      working_Map_collection.end()) {
00168         /* @169 Replace copy object in working_Map_collection associated with the unique ID key*/
00169         working_Map_collection_Object_Shared_Pointer_Iterator->second = object;
00170     /* @171 If error happes during store procees throw runtime error */
00171     } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND STORE FUNCTION, CANNOT
       STORE OBJECT]") );}
00172 }
```

4.3.3.13 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

Definition at line 52 of file TX.cpp.

References _tx_nesting_level, and working_Map_collection.

Referenced by commit().

```
00052                 {
00053     /* @54 If bigger than ZERO, means active nested transactions running in background, do not delete
       anything yet */
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 }
```

4.3.4 Friends And Related Function Documentation

4.3.4.1 friend class TM [friend]

Definition at line 74 of file TX.h.

4.3.5 Member Data Documentation

4.3.5.1 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().

4.3.5.2 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().

4.3.5.3 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().

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

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

Definition at line 117 of file TX.h.

Referenced by _register().

4.3.5.5 int TX::test_counter = 0 [static]

Definition at line 82 of file TX.h.

Referenced by getTest_counter().

4.3.5.6 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().

4.3.5.7 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().

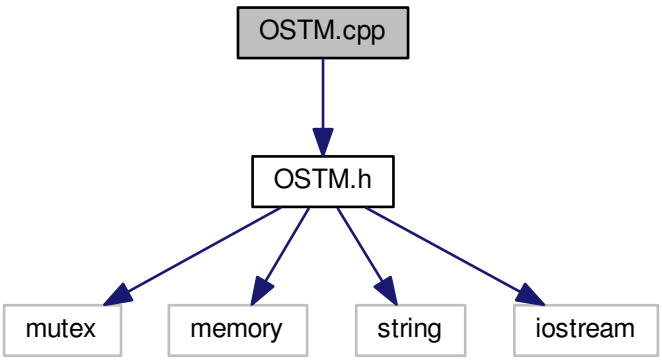
The documentation for this class was generated from the following files:

- TX.h
- TX.cpp

5 File Documentation

5.1 OSTM.cpp File Reference

`#include "OSTM.h"`
Include dependency graph for OSTM.cpp:



5.2 OSTM.cpp

```
/*
 * File:   OSTM.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 "OSTM.h"
```

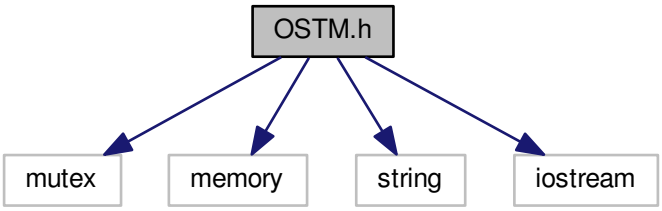
```
00015
00016 int OSTM::global_Unique_ID_Number = 0;
00017
00021 OSTM::OSTM()
00022 {
00023     /* @24 Integer field <version> indicates the version number of the inherited child object */
00024     this->version = ZERO;
00025     /* @26 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM
       library */
00026     this->uniqueID = Get_global_Unique_ID_Number();
00027     /* @28 Boolean value <canCommit> to determine the object can or cannot commit */
00028     this->canCommit = true;
00029     /* @30 Boolean field <abort_Transaction> to determine the object can or cannot commit */
00030     this->abort_Transaction = false;
00031 }
00032
00033
00039 OSTM::OSTM(int _version_number_, int _unique_id_)
00040 {
00041     /* @42 Integer field <version> indicates the version number of the inherited child object */
00042     this->uniqueID = _unique_id_;
00043     /* @44 Integer field <uniqueID> is a unique identifier assigned to every object registered in OSTM
       library */
00044     this->version = _version_number_;
00045     /* @46 Boolean value <canCommit> to determine the object can or cannot commit */
00046     this->canCommit = true;
00047     /* @48 Boolean value <abort_Transaction> to determine the object can or cannot commit */
00048     this->abort_Transaction = false;
00049 }
00050
00054 OSTM::~OSTM() {
00055     /* Destroy the object. */
00056 }
00061 int OSTM::Get_global_Unique_ID_Number() {
00062     /* @64 Checking the global_Unique_ID_Number */
00063     if(global_Unique_ID_Number > 10000000)
00064         /* @65 Reset global_Unique_ID_Number to ZERO*/
00065         global_Unique_ID_Number = 0;
00066     /* @67 return static global_Unique_ID_Number */
00067     return ++global_Unique_ID_Number;
00068 }
00069
00075 void OSTM::Set_Unique_ID(int uniqueID) {
00076     /* @77 set object uniqueID to parameter integer value */
00077     this->uniqueID = uniqueID;
00078 }
00082 int OSTM::Get_Unique_ID() const
00083 {
00084     /* @85 return Object uniqueID */
00085     return uniqueID;
00086 }
00092 void OSTM::Set_Version(int version)
00093 {
00094     /* @95 set object version to parameter integer value */
00095     this->version = version;
00096 }
00100 int OSTM::Get_Version() const
00101 {
00102     /* return object version number */
00103     return version;
00104 }
00108 void OSTM::increase_VersionNumber()
00109 {
00110     /* @111 increase object version number */
00111     this->version += 1;
00112 }
00117 void OSTM::Set_Can_Commit(bool canCommit) {
00118     /* @119 set canCommit object variable to parameter boolean value*/
00119     this->canCommit = canCommit;
00120 }
00124 bool OSTM::Is_Can_Commit() const {
00125     /* @126 return canCommit boolean value TRUE/FALSE */
00126     return canCommit;
00127 }
00132 void OSTM::Set_Abort_Transaction(bool abortTransaction) {
00133     /* @134 set abort_Transaction object variable to parameter boolean value  */
00134     this->abort_Transaction = abortTransaction;
00135 }
00140 bool OSTM::Is_Abort_Transaction() const {
00141     /* @142 return abort_Transaction object boolean value */
00142     return abort_Transaction;
00143 }
00147 void OSTM::lock_Mutex() {
00148     /* @149 Locking the mutex*/
00149     this->mutex.lock();
00150 }
00154 void OSTM::unlock_Mutex() {
```

```
    /* @156 Locking the mutex */
    this->mutex.unlock();
}
bool OSTM::try_lock(){
    /* @164 Try to unlock the mutex, return TRUE if the lock was acquired successfully, otherwise return
     FALSE */
    return this->mutex.try_lock();
}
```

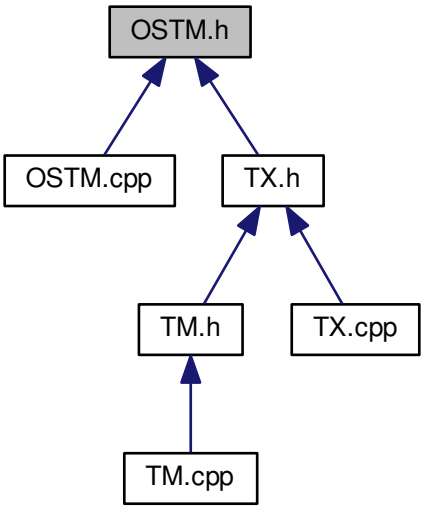
5.3 OSTM.h File Reference

```
#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

5.4 OSTM.h

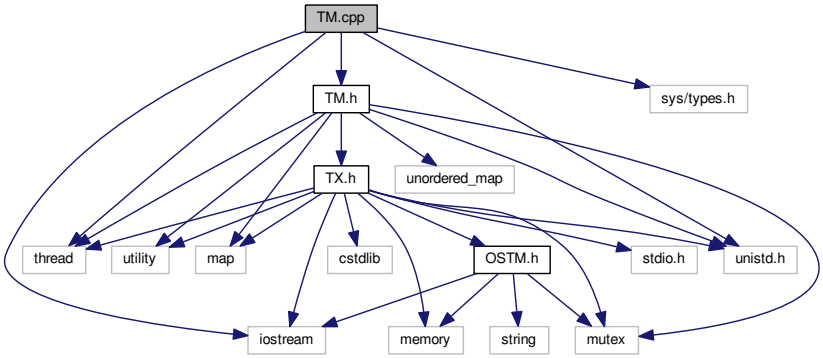
```
#ifndef OSTM_H
#define OSTM_H
#include <mutex>
#include <memory>
#include <string>
#include <iostream>
#include <string>

class OSTM {
public:
    /*
     * Default Constructor
     */
    OSTM();
    /*
     * Custom Constructor
     */
    OSTM(int _version_number_, int _unique_id_);
    /*
     * De-constructor
     */
    virtual ~OSTM();
    virtual void toString(){};
    virtual void copy(std::shared_ptr<OSTM> from, std::shared_ptr<OSTM> to){};
    virtual std::shared_ptr<OSTM> getBaseCopy(std::shared_ptr<OSTM> object){};
    /*
     * Setter for object unique id
     * @param uniqueID Integer to set the uniqueId
     */
    void Set_Unique_ID(int uniqueID);
    /*
     * 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.
00102      */
00103     bool try_lock();
00104
00105 private:
00111     int version;
00117     int uniqueID;
00121     bool canCommit;
00125     bool abort_Transaction;
00130     static int global_Unique_ID_Number;
00134     const int ZERO = 0;
00139     std::mutex mutex;
00143     int Get_global_Unique_ID_Number();
00144
00145 };
00146
00147 #endif /* OSTM_H */
```

5.5 TM.cpp File Reference

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

Include dependency graph for TM.cpp:



5.6 TM.cpp

```
00001 /*
00002  * File:   TM.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 "TM.h"
00015 #include <thread>
00016 #include <unistd.h>
00017 #include <sys/types.h>
00018 #include <iostream>
00019
00020 /*
00021    @23 _tm_id pid_t, process id determine the actual process between process in the STM library
```

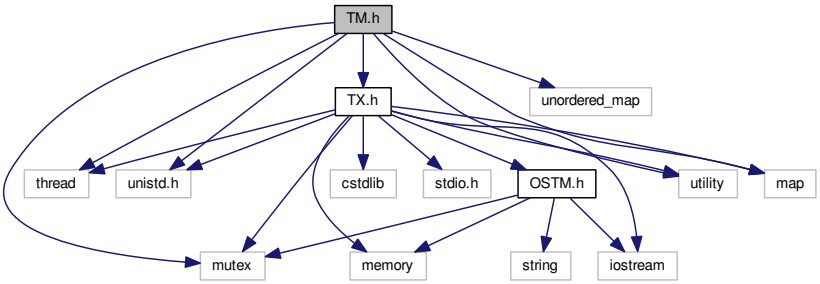
```
00022  */
00023 pid_t TM::_tm_id;
00024 /*
00025    @27 static Global std::map process_map_collection store all transactional objects/pointers
00026  */
00027 std::map<pid_t, std::map< std::thread::id, int >> TM::process_map_collection;
00031 TM& TM::Instance() {
00032    /* @33 _instance TM, static class reference to the instance of the Transaction Manager class */
00033     static TM _instance;
00034     /* @35 _instance ppid, assigning the process id whoever created the Singleton instance */
00035     _instance._tm_id = getpid();
00036     /* @37 return Singleton instance */
00037     return _instance;
00038 }
00039
00045 void TM::registerTX()
00046 {
00047   /* @49 guard std::lock_guard, locks the register_Lock mutex, unlock automatically when goes out of the
       scope register_Lock std::mutex, used by the lock_guard to protect shared map from race conditions */
00048     std::lock_guard<std::mutex> guard(register_Lock);
00049     /* @51 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00050     pid_t ppid = getppid();
00051
00052     /* @53 process_map_collection try to find the main process by it's ppid if registred in the library */
00053     std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator =
      TM::process_map_collection.find(ppid);
00054     /* @55 Check if iterator pointing to the end of the process map then register */
00055     if (process_map_collection_Iterator == TM::process_map_collection.end()) {
00056         /* @57 Require new map to insert to the process map as a value by the ppid key  */
00057         std::map< std::thread::id, int >map = get_thread_Map();
00058         /* @59 Register main process/application to the global map  */
00059         TM::process_map_collection.insert({ppid, map});
00060
00061     }
00062     /* @63 txMap std::map, collection to store all transaction created by the Transaction Manager */
00063     std::map<std::thread::id, std::shared_ptr < TX>>::iterator it = txMap.find(
      std::this_thread::get_id());
00064     /* @65 Check if iterator pointing to the end of the txMap then insert */
00065     if (it == txMap.end()) {
00066         /* @67 Create a new Transaction Object as a shared pointer */
00067         std::shared_ptr<TX> _transaction_object(new TX(std::this_thread::get_id()));
00068         /* @69 txMap insert the new transaction into the txMap by the threadID key */
00069         txMap.insert({std::this_thread::get_id(), _transaction_object});
00070         /* @71 Get the map if the transaction registered first time */
00071         process_map_collection_Iterator = TM::process_map_collection.find(ppid);
00072         /* @73 Insert to the GLOBAL MAP as a helper to clean up at end of main process. The value 1 is not
       used yet */
00073         process_map_collection_Iterator->second.insert({std::this_thread::get_id(), 1});
00074     }
00075 }
00076
00081 std::shared_ptr<TX>const TM::_get_tx()
00082 {
00083
00084     /* @85 guard std::lock_guard, locks the get_Lock mutex, unlock automatically when goes out of the scope
       get_Lock std::mutex, used by the lock_guard to protect txMap from race conditions */
00085     std::lock_guard<std::mutex> guard(get_Lock);
00086     /* @87 txMap try to find the TX Transaction object by it's actual thread ID if registred in the txMap
      */
00087     std::map<std::thread::id, std::shared_ptr<TX>>::iterator it = txMap.find(std::this_thread::get_id(
      ));
00088     /* @89 Check if iterator pointing to the end of the txMap then insert */
00089     if(it == txMap.end())
00090     {
00091       /* @92 If cannot find then call the register function to register the thread with a transaction */
00092        registerTX();
00093        /* @94 If it's registered first time then we need to find it after registration */
00094        it = txMap.find(std::this_thread::get_id());
00095
00096     } else {
00097         /* @98 If transaction already registered, it means the thread participating in nested transactions,
       and increase the nesting */
00098         it->second->_increase_tx_nesting();
00099     }
00100     /* @101 Returning back the transaction (TX) object to the thread */
00101     return it->second;
00102
00103 }
00108 void TM::_TX_EXIT(){
00109     /*  @110 Transaction manger create a local Transaction Object to access the TX class function without
       nesting any transaction */
00110     TX tx(std::this_thread::get_id());
00111     /* @112 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00112     pid_t ppid = getppid();
00113     /* @114 process_map_collection try to find the main process by it's ppid if registred in the library */
```

```
    std::map<pid_t, std::map< std::thread::id, int >>::iterator process_map_collection_Iterator = TM::process_map_collection.find(ppid);
    /* @116 Check if iterator NOT pointing to the end of the process map then register */
    if (process_map_collection_Iterator != TM::process_map_collection.end()) {
        /* @118 Iterate through the process_map_collection to find all transaction associated with main process */
        for (auto current = process_map_collection_Iterator->second.begin(); current != process_map_collection_Iterator->second.end(); ++current) {
            /* @120 Delete all transaction associated with the actual main process */
            txMap.erase(current->first);
        }
        /* @123 When all transaction deleted, delete the main process from the Transacion Manager */
        TM::process_map_collection.erase(ppid);
    }
    /* @126 TX class delete all Global Object shared between the transaction. This function calls only when the main process exists to clear out memory */
    tx.ostm_exit();
}
void TM::print_all(){
    /* @134 Locking the print function */
    get_Lock.lock();
    /* @136 Iterate through the txMap to print out the thread id's*/
    for (auto current = txMap.begin(); current != txMap.end(); ++current) {
        /* @138 Print key (thread number)*/
        std::cout << "KEY : " << current->first << std::endl;
    }
    /* @140 Unlocking the print function*/
    get_Lock.unlock();
}

std::map< std::thread::id, int > TM::get_thread_Map() {
    /*  @150 thread_Map std::map< int, int > create a map to store int key and int value */
    std::map< std::thread::id, int > thread_Map;
    /* @152 return the map */
    return thread_Map;
}
```

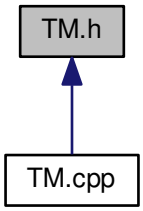
5.7 TM.h File Reference

```
#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

5.8 TM.h

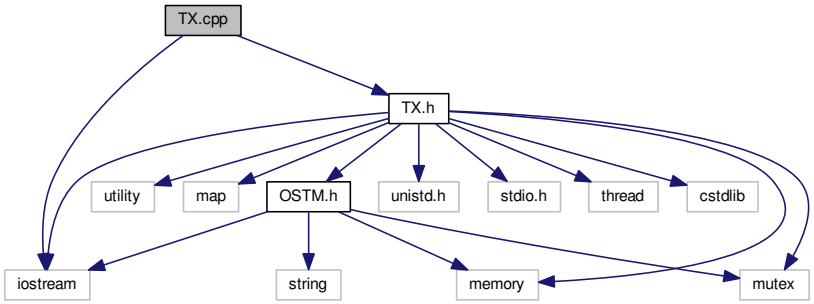
```
#ifndef TM_H
#define TM_H

#include <thread>
#include <unistd.h>//used for pid_t
#include <mutex>
#include <unordered_map>
#include <utility>
#include <map>
#include "TX.h"

class TM {
private:
    /*
     * TM constructor, prevent from multiple instantiation
     */
    TM() = default;
    /*
     * TM de-constructor, prevent from deletion
     */
    ~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
```

5.9 TX.cpp File Reference

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

Include dependency graph for TX.cpp:



5.10 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
 */
```

```
00027 std::mutex TX::register_Lock;
00028 /*
00029   @31 test_counter, register class level Integer variable to store the umber of rollback happens, for
       testing purposes
00030  */
00031 int TX::test_counter = 0;
00036 TX::TX(std::thread::id id) {
00037     /* @38 Integer field <transaction_Number> indicates the transaction number to the Transaction manager
      */
00038     this->transaction_Number = id;
00039     /* @40 Integer field <_tx_nesting_level> indicates the nesting level to the transaction itself */
00040     this->_tx_nesting_level = 0;
00041 }
00045 TX::~TX() {
00046     /* Destroy the object. */
00047 }
00052 void TX::th_exit() {
00053     /* @54 If bigger than ZERO, means active nested transactions running in background, do not delete
       anything yet */
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 }
00061
00068 void TX::ostm_exit() {
00069     /* @70 Declare Iterator main_Process_Map_collection_Iterator */
00070     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
00071     /* @72 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00072     pid_t ppid = getppid();
00073     /* @74 process_map_collection try to find the main process by it's ppid if registred in the library */
00074     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00075     /* @76 Check if iterator NOT pointing to the end of the process_map_collection then remove all
       associated elements */
00076     if (process_map_collection_Iterator != TX::process_map_collection.end()) {
00077         /* @78 Iterate through the process_map_collection to find all transaction associated with main
       process */
00078         for (auto current = process_map_collection_Iterator->second.begin(); current !=
      process_map_collection_Iterator->second.end(); ++current) {
00079             /* @80 Find the OSTM object in the Global shared map */
00080             main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find(current->first);
00081             /* @82 If object found then delete it*/
00082             if (main_Process_Map_collection_Iterator !=
      TX::main_Process_Map_collection.end()){
00083                 /* @84 Delete element from shared main_Process_Map_collection by object by the unique key,
       and the shaed_ptr will destroy automatically */
00084                 TX::main_Process_Map_collection.erase(
      main_Process_Map_collection_Iterator->first);
00085             }
00086         }
00087         /* @88 Delete main process from Process_map_collection */
00088         TX::process_map_collection.erase(process_map_collection_Iterator->first);
00089     }
00090 }
00091
00096 void TX::_register(std::shared_ptr<OSTM> object) {
00097     /* @98 register_Lock(mutex) shared lock between all transaction. MUST USE SHARED LOCK TO PROTECT SHARED
       GLOBAL MAP/COLLECTION */
00098     std::lock_guard<std::mutex> guard(TX::register_Lock);
00099     /* @100 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
00100     if(object == nullptr){
00101         throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN REGISTER FUNCTION]") );
00102     }
00103     /* @104 getppid() return the actual main process thread id, I used it to associate the Transactionas
       with the main processes */
00104     pid_t ppid = getppid();
00105     /* @106 Declare and initialize Iterator for process_map_collection, find main process*/
00106     std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
      TX::process_map_collection.find(ppid);
00107     /* @108 If iterator cannot find main process, then register*/
00108     if (process_map_collection_Iterator == TX::process_map_collection.end()) {
00109         /* @110 Create new empty map */
00110         std::map< int, int >map =  get_thread_Map();
00111         /* @112 Register main process/application to the global map */
00112         TX::process_map_collection.insert({ppid, map});
00113         /* @114 Get the map if registered first time */
00114         process_map_collection_Iterator = TX::process_map_collection.find(ppid);
00115     }
00116     /* @117 Declare and initialize Iterator for main_Process_Map_collection, find by original object */
00117     std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find(object->Get_Unique_ID());
00118     /* @119 If object cannot find, then register */
```

```
    if (main_Process_Map_collection_Iterator == TX::main_Process_Map_collection.end()) {
        /* '121 Insert the origin object to the GLOBAL MAP shared between transactions */
        TX::main_Process_Map_collection.insert({object->Get_Unique_ID(), object});
        /* @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 */
        process_map_collection_Iterator->second.insert({object->Get_Unique_ID(), 1});
    }
    /* @126 Declare and initialize Iterator for working_Map_collection, find copy of the original object */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    /* @128 If copy of the object not found, then register */
    if (working_Map_collection_Object_Shared_Pointer_Iterator == working_Map_collection.end()) {
        /* @130 Register transaction own copy of the original object */
        working_Map_collection.insert({object->Get_Unique_ID(), object->getBaseCopy(object)});
    }
}
std::shared_ptr<OSTM> TX::load(std::shared_ptr<OSTM> object) {
    /* @139 Declare and initialize Iterator for working_Map_collection */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    /* @141 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN LOAD FUNCTION]") );
    }
    /* @145 Find copy object in working_Map_collection by the object unique ID*/
    working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    /* @147 If object found, then return it */
    if (working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end()) {
        /* @149 Returning a copy of the working copy object */
        return working_Map_collection_Object_Shared_Pointer_Iterator->second->getBaseCopy(working_Map_collection_Object_Shared_Pointer_Iterator->second);
    /* @151 If no object found, throw runtime error */
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND LOAD FUNCTION]") );}
}
void TX::store(std::shared_ptr<OSTM> object) {
    /* @159 RUNTIME ERROR. Check for null pointer ! Null pointer can cause segmentation fault!!! */
    if(object == nullptr){
        throw std::runtime_error(std::string("[RUNTIME ERROR : NULL POINTER IN STORE FUNCTION]") );
    }
    /* @163 Declare and initialize Iterator for working_Map_collection */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    /* @165 Find copy object in working_Map_collection by the object unique ID*/
    working_Map_collection_Object_Shared_Pointer_Iterator = working_Map_collection.find(object->Get_Unique_ID());
    /* @167 If object found, then replace it */
    if (working_Map_collection_Object_Shared_Pointer_Iterator != working_Map_collection.end()) {
        /* @169 Replace copy object in working_Map_collection associated with the unique ID key*/
        working_Map_collection_Object_Shared_Pointer_Iterator->second = object;
    /* @171 If error happes during store procees throw runtime error */
    } else { throw std::runtime_error(std::string("[RUNTIME ERROR : NO OBJECT FOUND STORE FUNCTION, CANNOT STORE OBJECT]") );}
}
bool TX::commit() {
    /* @179 Declare can_Commit boolean variable */
    bool can_Commit = true;
    /* @182 Dealing with nested transactions first. if nesting level bigger than ZERO do not commit yet */
    if (this->_tx_nesting_level > 0) {
        /* @183 Decrease nesting level @see _decrease_tx_nesting() */
        _decrease_tx_nesting();
        return true;
    }
    /* @187 Declare and initialize Iterator for working_Map_collection */
    std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
    /* @189 Declare and initialize Iterator for main_Process_Map_collectio */
    std::map<int, std::shared_ptr<OSTM>>::iterator main_Process_Map_collection_Iterator;
    /* @191 Iterate through the working_Map_collection, for all associated copy objetcs */
    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++) {
            /* @193 Find the Original object in the Shared global colection by the copy object unique ID */
            main_Process_Map_collection_Iterator = TX::main_Process_Map_collection.find(working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
            /* @195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!! */
            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]") );
            }
```

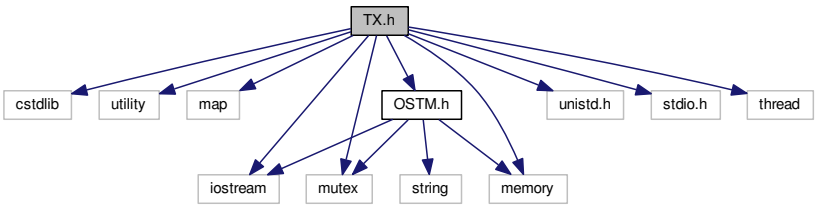
```
00199
00200         /* @200 Busy waiting WHILE try_lock function return false, If the object locked by another
       transaction, then waith until it's get unlocked, then lock it */
00201         while(!(main_Process_Map_collection_Iterator->second)->try_lock());
00202         /* @203 Compare the original global object version number with the working object version number.
       If the version number not same, then it cannot coomit*/
00203         if (main_Process_Map_collection_Iterator->second->Get_Version() >
      working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Version()) {
00204             /* @2005 Set object boolean value to FALSE, cannot commit */
00205             working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(false);
00206             /* @207 Set canCommit false Indicate rollback must happen */
00207             can_Commit = false;
00208             break;
00209         } else {
00210             /* @210 If version number are has same value set object boolean value to TRUE*/
00211             working_Map_collection_Object_Shared_Pointer_Iterator->second->Set_Can_Commit(true);
00212         }
00213     }
00214     /* @214 IF can_Commit boolean value setted for FALSE then rollback all copy object in the transaction
       to the Global object values*/
00215     if (!can_Commit) {
00216         /* @217 iterate through all transaction copy objects one by one */
00217         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++) {
00218             /* @219 Find the Global shared object by the transaction copy object unique ID */
00219             main_Process_Map_collection_Iterator  =
      TX::main_Process_Map_collection.find(
      working_Map_collection_Object_Shared_Pointer_Iterator->second->Get_Unique_ID());
00220             /* @221 Copy all Global shared original objects changed values by another transaction to the
       transaction copy objetcs */
00221             (working_Map_collection_Object_Shared_Pointer_Iterator->second)->copy(
      working_Map_collection_Object_Shared_Pointer_Iterator->second, main_Process_Map_collection_Iterator->second);
00222         }
00223         /* @224 When the transaction finish to change copying all values from original objects to local
       copy, then release all Global shared objects. @see _release_object_lock() */
00224         _release_object_lock();
00225         /* @226 Return FALSE to indicate the transaction must restart !*/
00226         return false;
00227     } else {
00228         /* @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 */
00229         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++) {
00230                 /* @231 Find the Global shared object by the transaction copy object unique ID */
00231                 main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find((
      working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
00232                 /* @233 If Global shared object found then commit changes */
00233                 if (main_Process_Map_collection_Iterator !=
      TX::main_Process_Map_collection.end()) {
00234                     /* @235 Copy over local transaction object values to original Global object*/
00235                     (main_Process_Map_collection_Iterator->second)->copy(
      main_Process_Map_collection_Iterator->second, working_Map_collection_Object_Shared_Pointer_Iterator->second);
00236                     /* @237 Increase the version number in the original pointer*/
00237                     main_Process_Map_collection_Iterator->second->increase_VersionNumber();
00238                 /* @195 RUNTIME ERROR. If no object found ! Null pointer can cause segmentation fault!!! */
00239                 } else { throw std::runtime_error(std::string("[RUNTIME ERROR : CAN'T FIND OBJECT COMMIT
       FUNCTION]")); }
00240         }
00241         /* @242 When the transaction finish with commit all changes, then release all Global shared
       objects. @see _release_object_lock() */
00242         _release_object_lock();
00243         /* @244 Transaction object clean up all associated values, clean memory. @see th_exit()*/
00244         this->th_exit();
00245         /* @246 Return TRUE, indicate the transaction has finished. */
00246         return true;
00247     }
00248 }
00249
00253 void TX::_release_object_lock(){
00254     /* @255 Declare Iterator for working_Map_collection */
00255     std::map< int, std::shared_ptr<OSTM> >::iterator working_Map_collection_Object_Shared_Pointer_Iterator;
00256     /* @255 Declare Iterator for working_Map_collection */
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++) {
00259             /* @260 Find Global shared original object by the transaction object unique ID*/
00260             main_Process_Map_collection_Iterator =
      TX::main_Process_Map_collection.find((
      working_Map_collection_Object_Shared_Pointer_Iterator->second)->Get_Unique_ID());
00261             /* @262 If object found, then release lock*/
```

```
            if (main_Process_Map_collection_Iterator !=
    TX::main_Process_Map_collection.end()) {
                /* @264 Release object lock */
                (main_Process_Map_collection_Iterator)->second->unlock_Mutex();
            }
        }
}

void TX::_increase_tx_nesting() {
    /* @274 Increase transaction nesting level */
    this->_tx_nesting_level += 1;
}
void TX::_decrease_tx_nesting() {
    /* @281 Decrease transaction nesting level */
    this->_tx_nesting_level -= 1;
;
}
int TX::getTest_counter() {
    /* @289 return class level value hold by test_counter variable */
    return TX::test_counter;
}
const std::thread::id TX::_get_tx_number() const {
    /* @296 Return the transaction nuber */
    return transaction_Number;
}
std::map< int, int > TX::get_thread_Map() {
    /* @303 initialize empty map hold int key and values*/
    std::map< int, int > thread_Map;
    /* @305 Return the map*/
    return thread_Map;
}

void TX::_print_all_tx() {
    /* @313 initialise Iterator */
    std::map< int, std::shared_ptr<OSTM> >::iterator it;
    /* @315 getppid() return the actual main process thread id, I used it to associate the Transactionas
     with the main processes */
    pid_t ppid = getppid();
    /* '317 initialize and assign Iterator to process_map_collection, by the main process id (ppid) */
    std::map<pid_t, std::map< int, int >>::iterator process_map_collection_Iterator =
    TX::process_map_collection.find(ppid);
    /* @319 If there is an entry associated with the process then print out all transactions. */
    if (process_map_collection_Iterator != TX::process_map_collection.end()) {
        /* @321 Iterate through process_map_collection*/
        for (auto current = process_map_collection_Iterator->second.begin(); current !=
    process_map_collection_Iterator->second.end(); ++current) {
            /* @323 Assign value to iterator */
            it = working_Map_collection.find(current->first);
            /* @325 If value found, then print it */
            if(it != working_Map_collection.end()){
                /* @327 print out the transaction number */
                std::cout << "[Unique number ] : " <<it->second->Get_Unique_ID() << std::endl;
            }
        }
    }
}
```

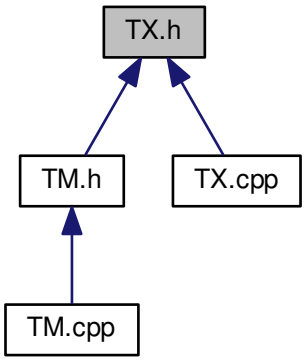
5.11 TX.h File Reference

```
#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

5.12 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_
```