AtomicX

Generated by Doxygen 1.9.3

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 thread Namespace Reference

Data Structures

- class atomicx

Chapter 5

Data Structure Documentation

5.1 thread::atomicx::aiterator Class Reference

`#include <atomicx.hpp>`

Public Member Functions

- aiterator ()=delete
- aiterator (atomicx *ptr)

  atomicx based constructor
- atomicx & operator* () const
- atomicx * operator-> ()
- aiterator & operator++ ()

Friends

- bool operator== (const aiterator &a, const aiterator &b)
- bool operator!= (const aiterator &a, const aiterator &b)

5.1.1 Detailed Description

5.1.1.1 ITERATOR FOR THREAD LISTING

Definition at line 145 of file atomicx.hpp.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 aiterator() [1/2]

```
thread::atomicx::aiterator::aiterator ( )  [delete]
```

5.1.2.2 aiterator() [2/2]

```
thread::atomicx::aiterator::aiterator (
            atomicx * ptr )
```

atomicx based constructor

Parameters

ptr	atomicx pointer to iterate

Definition at line 163 of file atomicx.cpp.

5.1.3 Member Function Documentation

5.1.3.1 operator∗()

```
atomicx & thread::atomicx::aiterator::operator* ( ) const
```

Definition at line 166 of file atomicx.cpp.

5.1.3.2 operator++()

```
atomicx::aiterator & thread::atomicx::aiterator::operator++ ( )
```

Definition at line 176 of file atomicx.cpp.

5.1.3.3 operator->()

```
atomicx * thread::atomicx::aiterator::operator-> ( )
```

Definition at line 171 of file atomicx.cpp.

5.1.4 Friends And Related Function Documentation

5.1.4.1 operator"!=

```
bool operator!= (
            const aiterator & a,
            const aiterator & b )  [friend]
```

Definition at line 172 of file atomicx.hpp.

5.1.4.2 operator==

```
bool operator== (
            const aiterator & a,
            const aiterator & b )  [friend]
```

Definition at line 171 of file atomicx.hpp.

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

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.2 thread::atomicx Class Reference

```
#include <atomicx.hpp>
```

Data Structures

- class aiterator
- struct Message
- class mutex
- class queue
- class semaphore
- class smart_ptr
- class smartMutex

  RII compliance lock/shared lock to auto unlock on destruction.
- class smartSemaphore
- class Timeout

  Timeout Check object.

Public Types

- enum class aTypes : uint8_t {
  start =1 , running =5 , now =6 , stop =10 ,
  lock =50 , wait =55 , subscription =60 , sleep =100 ,
  stackOverflow =255 }
- enum class aSubTypes : uint8_t {
  error =10 , ok , look , wait ,
  timeout }
- enum class NotifyType : uint8_t { one = 0 , all = 1 }

Public Member Functions

- aiterator begin (void)

  Get the beggining of the list.
- aiterator end (void)

  Get the end of the list.
- virtual ~atomicx (void)

  virtual destructor of the atomicx
- size_t GetID (void)

  Get the current thread ID.
- size_t GetStackSize (void)

  Get the Max Stack Size for the thread.
- atomicx_time GetNice (void)

  Get the Nice the current thread.
- size_t GetUsedStackSize (void)

  Get the Used Stack Size for the thread since the last context change cycle.
- atomicx_time GetCurrentTick (void)

  Get the Current Tick using the ported tick granularity function.

Static Public Member Functions

- static atomicx * GetCurrent ()

  Get the Current thread in execution.
- static bool Start (void)

  Once it is call the process blocks execution and start all threads.

If GetName was not overloaded by the derived thread implementation

Get the Name object

Returns

const char* name in plain c string

a standard name will be returned.

- virtual const char * GetName (void)
- atomicx_time GetTargetTime (void)

  Get next moment in ported tick granularity the thread will be due to return.
- int GetStatus (void)

  Get the current thread status.
- int GetSubStatus (void)

  Get the current thread sub status.
- size_t GetReferenceLock (void)

  Get the Reference Lock last used to lock the thread.
- size_t GetTagLock (void)

  Get the last tag message posted.
- void SetNice (atomicx_time nice)

  Set the Nice of the thread.
- template<typename T , size_t N>
  atomicx (T(&stack)[N])

  Construct a new atomicx thread.
- atomicx (size_t nStackSize=0, int nStackIncreasePace=1)

  Construct a new atomicx object and set initial auto stack and increase pace.
- virtual void run (void) noexcept=0

  The pure virtual function that runs the thread loop.
- virtual void StackOverflowHandler (void) noexcept=0

  Handles the StackOverflow of the current thread.
- virtual void finish () noexcept

  Called right after run returns, can be used to self-destroy the object and other maintenance actions.
- bool IsStackSelfManaged (void)

  Return if the current thread's stack memory is automatic.
- bool Yield (atomicx_time nSleep=ATOMICX_TIME_MAX)

  Foce the context change explicitly.
- atomicx_time GetLastUserExecTime ()

  Get the Last Execution of User Code.
- size_t GetStackIncreasePace (void)

  Get the Stack Increase Pace value.
- void YieldNow (void)

  Trigger a high priority NOW, caution it will always execute before normal yield.
- void SetDynamicNice (bool status)

  Set the Dynamic Nice on and off.
- bool IsDynamicNiceOn ()

  Get Dynamic Nice status.
- uint32_t GetTopicID (const char *pszTopic, size_t nKeyLenght)

  calculate the Topic ID for a given topic text
- template<typename T >
  bool LookForWaitings (T &refVar, size_t nTag, size_t hasAtleast, atomicx_time waitFor)

  Sync with thread call for a wait (refVar,nTag)

- template<typename T >
  bool LookForWaitings (T &refVar, size_t nTag, atomicx_time waitFor)

  Sync with thread call for a wait (refVar,nTag)
- template<typename T >
  bool IsWaiting (T &refVar, size_t nTag=0, size_t hasAtleast=1, aSubTypes asubType=aSubTypes::wait)

  Check if there are waiting threads for a given reference pointer and tag value.
- template<typename T >
  size_t HasWaitings (T &refVar, size_t nTag=0, aSubTypes asubType=aSubTypes::wait)

  Report how much waiting threads for a given reference pointer and tag value are there.
- template<typename T >
  bool Wait (size_t &nMessage, T &refVar, size_t nTag=0, atomicx_time waitFor=0, aSubTypes asub↩
  Type=aSubTypes::wait)

  Blocks/Waits a notification along with a message and tag from a specific reference pointer.
- template<typename T >
  bool Wait (T &refVar, size_t nTag=0, atomicx_time waitFor=0, aSubTypes asubType=aSubTypes::wait)

  Blocks/Waits a notification along with a tag from a specific reference pointer.
- template<typename T >
  size_t SafeNotify (size_t &nMessage, T &refVar, size_t nTag=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType=aSubTypes::wait)

  Safely notify all Waits from a specific reference pointer along with a message without triggering context change.
- template<typename T >
  size_t Notify (size_t &nMessage, T &refVar, size_t nTag=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType=aSubTypes::wait)

  Notify all Waits from a specific reference pointer along with a message and trigger context change if at least one wait thread got notified.
- template<typename T >
  size_t Notify (size_t &&nMessage, T &refVar, size_t nTag=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType=aSubTypes::wait)
- template<typename T >
  size_t SyncNotify (size_t &nMessage, T &refVar, size_t nTag=0, atomicx_time waitForWaitings=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType=aSubTypes::wait)

  SYNC Waits for at least one Wait call for a given reference pointer along with a message and trigger context change.
- template<typename T >
  size_t SyncNotify (size_t &&nMessage, T &refVar, size_t nTag=0, atomicx_time waitForWaitings=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType=aSubTypes::wait)
- template<typename T >
  size_t SafeNotify (T &refVar, size_t nTag=0, NotifyType notifyAll=NotifyType::one, aSubTypes asub↩
  Type=aSubTypes::wait)

  Safely notify all Waits from a specific reference pointer without triggering context change.
- template<typename T >
  size_t SyncNotify (T &refVar, size_t nTag, atomicx_time waitForWaitings=0, NotifyType notify↩
  All=NotifyType::one, aSubTypes asubType=aSubTypes::wait)

  SYNC Waits for at least one Wait call for a given reference pointer and trigger context change.
- template<typename T >
  size_t Notify (T &refVar, size_t nTag=0, NotifyType notifyAll=NotifyType::one, aSubTypes asub↩
  Type=aSubTypes::wait)

  Notify all Waits from a specific reference pointer and trigger context change if at least one wait thread got notified.
- bool WaitBrokerMessage (const char *pszKey, size_t nKeyLenght, Message &message)

  Block and wait for message from a specific topic string.
- bool WaitBrokerMessage (const char *pszKey, size_t nKeyLenght)

  Block and wait for a notification from a specific topic string.
- bool Publish (const char *pszKey, size_t nKeyLenght, const Message message)

  Publish a message for a specific topic string and trigger a context change if any delivered.
- bool SafePublish (const char *pszKey, size_t nKeyLenght, const Message message)

Safely Publish a message for a specific topic string DO NOT trigger a context change if any delivered.

- bool Publish (const char *pszKey, size_t nKeyLenght)

  Publish a notification for a specific topic string and trigger a context change if any delivered.
- bool SafePublish (const char *pszKey, size_t nKeyLenght)

  Safely Publish a notification for a specific topic string DO NOT trigger a context change if any delivered.
- bool HasSubscriptions (const char *pszTopic, size_t nKeyLenght)

  Check if there is subscryption for a specific Topic String.
- bool HasSubscriptions (uint32_t nKeyID)

  Check if there is subscryption for a specific Topic ID.
- virtual bool BrokerHandler (const char *pszKey, size_t nKeyLenght, Message &message)

  Default broker handler for a subscribed message.
- virtual bool IsSubscribed (const char *pszKey, size_t nKeyLenght)

  Specialize and gives power to decide if a topic is subscrybed on not.
- void SetStackIncreasePace (size_t nIncreasePace)

  Set the Stack Increase Pace object.

5.2.1 Detailed Description

Definition at line 47 of file atomicx.hpp.

5.2.2 Member Enumeration Documentation

5.2.2.1 aSubTypes

```
enum class thread::atomicx::aSubTypes :  uint8_t  [strong]
```

Enumerator

error	
ok	
look	
wait	
timeout	

Definition at line 69 of file atomicx.hpp.

5.2.2.2 aTypes

```
enum class thread::atomicx::aTypes :  uint8_t  [strong]
```

5.2.2.3 STATE MACHINE TYPES

Enumerator

start	
running	
now	
stop	
lock	
wait	
subscription	
sleep	

Enumerator

stackOverflow	

Definition at line 56 of file atomicx.hpp.

5.2.2.4 NotifyType

```
enum class thread::atomicx::NotifyType :  uint8_t  [strong]
```

Enumerator

one	
all	

Definition at line 78 of file atomicx.hpp.

5.2.3 Constructor & Destructor Documentation

5.2.3.1 ~atomicx()

```
thread::atomicx::~atomicx (
            void  )  [virtual]
```

virtual destructor of the atomicx
PUBLIC OBJECT METHOS
Definition at line 465 of file atomicx.cpp.

5.2.3.2 atomicx() [1/2]

```
template<typename T , size_t N>
thread::atomicx::atomicx (
            T(&) stack[N] )  [inline]
```

Construct a new atomicx thread.

Template Parameters

T	Stack memory page type
N	Stack memory page size

Definition at line 920 of file atomicx.hpp.

5.2.3.3 atomicx() [2/2]

```
thread::atomicx::atomicx (
            size_t nStackSize = 0,
            int nStackIncreasePace = 1 )
```

Construct a new atomicx object and set initial auto stack and increase pace.

Parameters

nStackSize	Initial Size of the stack

Parameters

nStackIncreasePace	defalt=1, The increase pace on each resize

Definition at line 456 of file atomicx.cpp.

5.2.4 Member Function Documentation

5.2.4.1 begin()

```
atomicx::aiterator thread::atomicx::begin (
            void  )
```

Get the beggining of the list.

Returns

aiterator

Definition at line 182 of file atomicx.cpp.

5.2.4.2 BrokerHandler()

```
virtual bool thread::atomicx::BrokerHandler (
            const char * pszKey,
            size_t nKeyLenght,
            Message & message )  [inline], [protected], [virtual]
```

Default broker handler for a subscribed message.

Parameters

pszKey	The Topic C string
nKeyLenght	The Topic C string size in bytes
message	The atomicx::message payload received

Returns

true signify it was correctly processed

Note

Can be overloaded by the derived by the derived thread implementation and specialized, otherwise a empty function will be called instead

Definition at line 1574 of file atomicx.hpp.

5.2.4.3 end()

```
atomicx::aiterator thread::atomicx::end (
            void  )
```

Get the end of the list.

Returns

aiterator

Definition at line 183 of file atomicx.cpp.

5.2.4.4 finish()

```
virtual void thread::atomicx::finish ( )  [inline], [virtual], [noexcept]
```

Called right after run returns, can be used to self-destroy the object and other maintenance actions.

Note

if not implemented a default "empty" call is used instead

Definition at line 954 of file atomicx.hpp.

5.2.4.5 GetCurrent()

```
atomicx * thread::atomicx::GetCurrent ( )  [static]
```

Get the Current thread in execution.

Returns

atomicx∗ thread

Definition at line 185 of file atomicx.cpp.

5.2.4.6 GetCurrentTick()

```
atomicx_time thread::atomicx::GetCurrentTick (
            void  )
```

Get the Current Tick using the ported tick granularity function.

Returns

atomicx_time based on the ported tick granularity

Definition at line 831 of file atomicx.cpp.

5.2.4.7 GetID()

```
size_t thread::atomicx::GetID (
            void  )
```

Get the current thread ID.

Returns

size_t Thread ID number

Definition at line 485 of file atomicx.cpp.

5.2.4.8 GetLastUserExecTime()

```
atomicx_time thread::atomicx::GetLastUserExecTime ( )
```

Get the Last Execution of User Code.

Returns

atomicx_time

Definition at line 841 of file atomicx.cpp.

5.2.4.9 GetName()

```
const char * thread::atomicx::GetName (
            void  )  [virtual]
```

Definition at line 475 of file atomicx.cpp.

5.2.4.10 GetNice()

```
atomicx_time thread::atomicx::GetNice (
            void  )
```

Get the Nice the current thread.

Returns

atomicx_time the number representing the nice and based on the ported tick granularity.

Definition at line 407 of file atomicx.cpp.

5.2.4.11 GetReferenceLock()

```
size_t thread::atomicx::GetReferenceLock (
            void  )
```

Get the Reference Lock last used to lock the thread.

Returns

size_t the lock_id (used my wait)

Definition at line 505 of file atomicx.cpp.

5.2.4.12 GetStackIncreasePace()

```
size_t thread::atomicx::GetStackIncreasePace (
            void  )
```

Get the Stack Increase Pace value.
Definition at line 851 of file atomicx.cpp.

5.2.4.13 GetStackSize()

```
size_t thread::atomicx::GetStackSize (
            void  )
```

Get the Max Stack Size for the thread.

Returns

size_t size in bytes

Definition at line 412 of file atomicx.cpp.

5.2.4.14 GetStatus()

```
int thread::atomicx::GetStatus (
            void  )
```

Get the current thread status.

Returns

int use atomicx::aTypes

Definition at line 495 of file atomicx.cpp.

5.2.4.15 GetSubStatus()

```
int thread::atomicx::GetSubStatus (
            void  )
```

Get the current thread sub status.

Returns

int use atomicx::aTypes

Definition at line 500 of file atomicx.cpp.

5.2.4.16 GetTagLock()

```
size_t thread::atomicx::GetTagLock (
            void  )
```

Get the last tag message posted.

Returns

size_t atomicx::message::tag value

Definition at line 510 of file atomicx.cpp.

5.2.4.17 GetTargetTime()

```
atomicx_time thread::atomicx::GetTargetTime (
            void  )
```

Get next moment in ported tick granularity the thread will be due to return.

Returns

atomicx_time based on the ported tick granularity

Definition at line 490 of file atomicx.cpp.

5.2.4.18 GetTopicID()

```
uint32_t thread::atomicx::GetTopicID (
            const char * pszTopic,
            size_t nKeyLenght )  [protected]
```

calculate the Topic ID for a given topic text

Parameters

pszTopic	Topic Text in C string
nKeyLenght	Size, in bytes + zero terminated char

Returns

uint32_t The calculated topic ID

Definition at line 667 of file atomicx.cpp.

5.2.4.19 GetUsedStackSize()

```
size_t thread::atomicx::GetUsedStackSize (
            void  )
```

Get the Used Stack Size for the thread since the last context change cycle.

Returns

size_t size in bytes

Definition at line 417 of file atomicx.cpp.

5.2.4.20 HasSubscriptions() [1/2]

```
bool thread::atomicx::HasSubscriptions (
            const char * pszTopic,
            size_t nKeyLenght )  [protected]
```

Check if there is subscryption for a specific Topic String.

Parameters

pszTopic	The Topic string in C string
nKeyLenght	The Topic C string length in bytes

Returns

true if any substriction is found, otherwise false

Definition at line 672 of file atomicx.cpp.

5.2.4.21 HasSubscriptions() [2/2]

```
bool thread::atomicx::HasSubscriptions (
            uint32_t nKeyID ) [protected]
```

Check if there is subscryption for a specific Topic ID.

Parameters

nKeyID	The Topic ID uint32_t

Returns

true if any substriction is found, otherwise false

Definition at line 690 of file atomicx.cpp.

5.2.4.22 HasWaitings()

```
template<typename T >
size_t thread::atomicx::HasWaitings (
            T & refVar,
            size_t nTag = 0,
            aSubTypes asubType = aSubTypes::wait ) [inline], [protected]
```

Report how much waiting threads for a given reference pointer and tag value are there.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer used a a notifier

Parameters

nTag	The size↩ _t tag that will give meaning to the notification, if nTag == 0 mean all bTag for the refVar
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == a↩ Sub↩ Type↩ ::wait

Returns

true

Note

This is a powerful tool since it create layers of waiting within the same reference pointer

Definition at line 1220 of file atomicx.hpp.

5.2.4.23 IsDynamicNiceOn()

```
bool thread::atomicx::IsDynamicNiceOn ( )
```

Get Dynamic Nice status.

Returns

true if dynamic nice is on otherwise off

Definition at line 861 of file atomicx.cpp.

5.2.4.24 IsStackSelfManaged()

```
bool thread::atomicx::IsStackSelfManaged (
            void  )
```

Return if the current thread's stack memory is automatic.
Definition at line 836 of file atomicx.cpp.

5.2.4.25 IsSubscribed()

```
virtual bool thread::atomicx::IsSubscribed (
            const char * pszKey,
            size_t nKeyLenght )  [inline], [protected], [virtual]
```

Specialize and gives power to decide if a topic is subscrybed on not.

Parameters

pszKey	The Topic C String
nKeyLenght	The Topic C String size in bytes

Returns

true if the given topic was subscribed, otherwise false.

Definition at line 1588 of file atomicx.hpp.

5.2.4.26 IsWaiting()

```
template<typename T >
bool thread::atomicx::IsWaiting (
            T & refVar,
            size_t nTag = 0,
            size_t hasAtleast = 1,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Check if there are waiting threads for a given reference pointer and tag value.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer used a a notifier
nTag	The size_t tag that will give meaning to the notification, if nTag == 0 mean all bTag for the refVar
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == aSubType::wait

Returns

true

Note

This is a powerful tool since it create layers of waiting within the same reference pointer

Definition at line 1189 of file atomicx.hpp.

5.2.4.27 LookForWaitings() [1/2]

```
template<typename T >
bool thread::atomicx::LookForWaitings (
            T & refVar,
            size_t nTag,
            atomicx_time waitFor )  [inline], [protected]
```

Sync with thread call for a wait (refVar,nTag)

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer
nTag	The notification meaning, if nTag == 0 means wait all refVar regardless
waitFor	default=0, if 0 wait indefinitely, otherwise wait for custom tick granularity times

Returns

true There is thread waiting for the given refVar/nTag

Definition at line 1157 of file atomicx.hpp.

5.2.4.28 LookForWaitings() [2/2]

```
template<typename T >
bool thread::atomicx::LookForWaitings (
            T & refVar,
            size_t nTag,
            size_t hasAtleast,
            atomicx_time waitFor )  [inline], [protected]
```

Sync with thread call for a wait (refVar,nTag)

5.2.4.29 SMART WAIT/NOTIFY IMPLEMENTATION

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer
nTag	The notification meaning, if nTag == 0 means wait all refVar regardless
waitFor	default=0, if 0 wait indefinitely, otherwise wait for custom tick granularity times

Parameters

hasAtleast	define how minimal Wait calls to report true

Returns

true There is thread waiting for the given refVar/nTag

Definition at line 1120 of file atomicx.hpp.

5.2.4.30 Notify() [1/3]

```
template<typename T >
size_t thread::atomicx::Notify (
            size_t && nMessage,
            T & refVar,
            size_t nTag = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Definition at line 1349 of file atomicx.hpp.

5.2.4.31 Notify() [2/3]

```
template<typename T >
size_t thread::atomicx::Notify (
            size_t & nMessage,
            T & refVar,
            size_t nTag = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Notify all Waits from a specific reference pointer along with a message and trigger context change if at least one wait thread got notified.

Template Parameters

T	Type of the reference pointer

Parameters

nMessage	The size_t message to be sent

Parameters

refVar	The reference pointer used a a notifier
nTag	The size↩ _t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless
notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.

Parameters

asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == aSubType::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1340 of file atomicx.hpp.

5.2.4.32 Notify() [3/3]

```
template<typename T >
size_t thread::atomicx::Notify (
            T & refVar,
            size_t nTag = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Notify all Waits from a specific reference pointer and trigger context change if at least one wait thread got notified.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer used a a notifier

Parameters

nTag	The size↩ _t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless
notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.

Parameters

asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == a↩ Sub↩ Type↩ ::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1461 of file atomicx.hpp.

5.2.4.33 Publish() [1/2]

```
bool thread::atomicx::Publish (
            const char * pszKey,
            size_t nKeyLenght ) [protected]
```

Publish a notification for a specific topic string and trigger a context change if any delivered.

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes

Returns

true if at least one thread has received a message

Definition at line 822 of file atomicx.cpp.

5.2.4.34 Publish() [2/2]

```
bool thread::atomicx::Publish (
            const char * pszKey,
            size_t nKeyLenght,
            const Message message )  [protected]
```

Publish a message for a specific topic string and trigger a context change if any delivered.

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes
message	the atomicx::message structure with message and tag

Returns

true if at least one thread has received a message

Definition at line 777 of file atomicx.cpp.

5.2.4.35 run()

```
virtual void thread::atomicx::run (
            void  )  [pure virtual], [noexcept]
```

The pure virtual function that runs the thread loop.

Note

REQUIRED implementation and once it returns it will execute finish method

5.2.4.36 SafeNotify() [1/2]

```
template<typename T >
size_t thread::atomicx::SafeNotify (
```

```
size_t & nMessage,
T & refVar,
size_t nTag = 0,
NotifyType notifyAll = NotifyType::one,
aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Safely notify all Waits from a specific reference pointer along with a message without triggering context change.

Template Parameters

T	Type of the reference pointer

Parameters

nMessage	The size_t message to be sent
refVar	The reference pointer used a a notifier
nTag	The size_t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless

Parameters

notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == a↩ Sub↩ Type↩ ::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1321 of file atomicx.hpp.

5.2.4.37 SafeNotify() [2/2]

```
template<typename T >
```

```
size_t thread::atomicx::SafeNotify (
            T & refVar,
            size_t nTag = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Safely notify all Waits from a specific reference pointer without triggering context change.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer used a a notifier
nTag	The size_t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless

Parameters

notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == a↩ Sub↩ Type↩ ::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1414 of file atomicx.hpp.

5.2.4.38 SafePublish() [1/2]

```
bool thread::atomicx::SafePublish (
```

```
            const char * pszKey,
            size_t nKeyLenght )  [protected]
```

Safely Publish a notification for a specific topic string DO NOT trigger a context change if any delivered.

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes

Returns

true if at least one thread has received a message

Note

Ideal for been used with interrupt request

Definition at line 786 of file atomicx.cpp.

5.2.4.39 SafePublish() [2/2]

```
bool thread::atomicx::SafePublish (
            const char * pszKey,
            size_t nKeyLenght,
            const Message message )  [protected]
```

Safely Publish a message for a specific topic string DO NOT trigger a context change if any delivered.

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes
message	the atomicx::message structure with message and tag

Returns

true if at least one thread has received a message

Note

Ideal for been used with interrupt request

Definition at line 744 of file atomicx.cpp.

5.2.4.40 SetDynamicNice()

```
void thread::atomicx::SetDynamicNice (
            bool status )
```

Set the Dynamic Nice on and off.

Parameters

status	True for on otherwsize off

Definition at line 856 of file atomicx.cpp.

5.2.4.41 SetNice()

```
void thread::atomicx::SetNice (
            atomicx_time nice )
```

Set the Nice of the thread.

Parameters

nice	in atomicx_time reference based on the ported tick granularity

Definition at line 480 of file atomicx.cpp.

5.2.4.42 SetStackIncreasePace()

```
void thread::atomicx::SetStackIncreasePace (
            size_t nIncreasePace )  [protected]
```

Set the Stack Increase Pace object.

Parameters

nIncreasePace	The new stack increase pace value

Definition at line 846 of file atomicx.cpp.

5.2.4.43 StackOverflowHandler()

```
virtual void thread::atomicx::StackOverflowHandler (
            void  )  [pure virtual], [noexcept]
```

Handles the StackOverflow of the current thread.

Note

REQUIRED

5.2.4.44 Start()

```
bool thread::atomicx::Start (
            void  )  [static]
```

Once it is call the process blocks execution and start all threads.

Returns

false if it was destried by dead lock (all threads locked)

Definition at line 269 of file atomicx.cpp.

5.2.4.45 SyncNotify() [1/3]

```
template<typename T >
size_t thread::atomicx::SyncNotify (
            size_t && nMessage,
            T & refVar,
            size_t nTag = 0,
            atomicx_time waitForWaitings = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Definition at line 1387 of file atomicx.hpp.

5.2.4.46 SyncNotify() [2/3]

```
template<typename T >
size_t thread::atomicx::SyncNotify (
            size_t & nMessage,
            T & refVar,
            size_t nTag = 0,
            atomicx_time waitForWaitings = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

SYNC Waits for at least one Wait call for a given reference pointer along with a message and trigger context change.

Template Parameters

T	Type of the reference pointer

Parameters

nMessage	The size_t message to be sent
refVar	The reference pointer used a a notifier
nTag	The size_t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless

Parameters

waitForWaitings	default=0 (waiting for Waiting calls) otherSize wait for Wait commands compatible with the paramenters (Sync call).
notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.

Parameters

asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == aSubType::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1373 of file atomicx.hpp.

5.2.4.47 SyncNotify() [3/3]

```
template<typename T >
size_t thread::atomicx::SyncNotify (
            T & refVar,
            size_t nTag,
            atomicx_time waitForWaitings = 0,
            NotifyType notifyAll = NotifyType::one,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

SYNC Waits for at least one Wait call for a given reference pointer and trigger context change.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	The reference pointer used a a notifier

Parameters

nTag	The size↩ _t tag that will give meaning to the notification, if nTag == 0 means notify all refVar regardless
waitForWaitings	default=0 (waiting for Waiting calls) othersize wait for Wait commands compatible with the paramenters (Sync call).

Parameters

notifyAll	default = false, and only the fist available refVar Waiting thread will be notified, if true all available refVar waiting thread will be notified.
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == a↩ Sub↩ Type↩ ::wait

Returns

true if at least one got notified, otherwise false.

Definition at line 1434 of file atomicx.hpp.

5.2.4.48 Wait() [1/2]

```
template<typename T >
```

```
bool thread::atomicx::Wait (
            size_t & nMessage,
            T & refVar,
            size_t nTag = 0,
            atomicx_time waitFor = 0,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Blocks/Waits a notification along with a message and tag from a specific reference pointer.

Template Parameters

T	Type of the reference pointer

Parameters

nMessage	the size_t message to be received
refVar	the reference pointer used as a notifier
nTag	the size_t tag that will give meaning to the the message, if nTag == 0 means wait all refVar regardless

Parameters

waitFor	default==0 (undefinitly), How log to wait for a notification based on atomicx_time
asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == aSubType::wait

Returns

true if it was successfully received.

Definition at line 1247 of file atomicx.hpp.

5.2.4.49 Wait() [2/2]

```
template<typename T >
bool thread::atomicx::Wait (
            T & refVar,
            size_t nTag = 0,
            atomicx_time waitFor = 0,
            aSubTypes asubType = aSubTypes::wait )  [inline], [protected]
```

Blocks/Waits a notification along with a tag from a specific reference pointer.

Template Parameters

T	Type of the reference pointer

Parameters

refVar	the reference pointer used as a notifier
nTag	the size_t tag that will give meaning to the the message, if nTag == 0 means wait all refVar regardless
waitFor	default==0 (undefinitly), How log to wait for a notification based on atomicx_time

Parameters

asubType	Type of the notification, only use it if you know what you are doing, it creates a different type of wait/notify, deafault == aSubType::wait

Returns

true if it was successfully received.

Definition at line 1284 of file atomicx.hpp.

5.2.4.50 WaitBrokerMessage() [1/2]

```
bool thread::atomicx::WaitBrokerMessage (
            const char * pszKey,
            size_t nKeyLenght ) [protected]
```

Block and wait for a notification from a specific topic string.

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes

Returns

true if it was successfully received, otherwise false

Definition at line 725 of file atomicx.cpp.

5.2.4.51 WaitBrokerMessage() [2/2]

```
bool thread::atomicx::WaitBrokerMessage (
            const char * pszKey,
            size_t nKeyLenght,
            Message & message )  [protected]
```

Block and wait for message from a specific topic string.

5.2.4.52 SMART BROKER IMPLEMENTATION

Parameters

pszKey	The Topic string
nKeyLenght	The size of the topic string in bytes
message	the atomicx::message structure with message and tag

Returns

true if it was successfully received, otherwise false

Definition at line 703 of file atomicx.cpp.

5.2.4.53 Yield()

```
bool thread::atomicx::Yield (
            atomicx_time nSleep = ATOMICX_TIME_MAX )
```

Foce the context change explicitly.

Parameters

nSleep	default is ATOMICX↩ _↩ TIME↩ _MAX, otherwise it will override the nice and sleep for n custom tick granularity

Returns

true if the context came back correctly, otherwise false

Definition at line 308 of file atomicx.cpp.

5.2.4.54 YieldNow()

```
void thread::atomicx::YieldNow (
            void  )
```

Trigger a high priority NOW, caution it will always execute before normal yield.
Definition at line 400 of file atomicx.cpp.

5.2.5 Field Documentation

5.2.5.1 autoStack

```
bool thread::atomicx::autoStack
```

Definition at line 1666 of file atomicx.hpp.

5.2.5.2 dynamicNice

```
bool thread::atomicx::dynamicNice
```

Definition at line 1667 of file atomicx.hpp.
The documentation for this class was generated from the following files:

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.3 thread::atomicx::Message Struct Reference

```
#include <atomicx.hpp>
```

Data Fields

- size_t tag
- size_t message

5.3.1 Detailed Description

PROTECTED METHODS, THOSE WILL BE ONLY ACCESSIBLE BY THE THREAD ITSELF
Definition at line 1087 of file atomicx.hpp.

5.3.2 Field Documentation

5.3.2.1 message

`size_t thread::atomicx::Message::message`
Definition at line 1090 of file atomicx.hpp.

5.3.2.2 tag

`size_t thread::atomicx::Message::tag`
Definition at line 1089 of file atomicx.hpp.
The documentation for this struct was generated from the following file:

- atomicx/atomicx.hpp

5.4 thread::atomicx::mutex Class Reference

`#include <atomicx.hpp>`

Public Member Functions

- void Lock ()

  Exclusive/binary lock the smart lock.
- void Unlock ()

  Release the exclusive lock.
- void SharedLock ()

  Shared Lock for the smart Lock.
- void SharedUnlock ()

  Release the current shared lock.
- size_t IsShared ()

  Check how many shared locks are accquired.
- bool IsLocked ()

  Check if a exclusive lock has been already accquired.

5.4.1 Detailed Description

5.4.1.1 SMART LOCK IMPLEMENTATION

Definition at line 696 of file atomicx.hpp.

5.4.2 Member Function Documentation

5.4.2.1 IsLocked()

`bool thread::atomicx::mutex::IsLocked ( )`
Check if a exclusive lock has been already accquired.

Returns

true if yes, otherwise false

Definition at line 580 of file atomicx.cpp.

5.4.2.2 IsShared()

`size_t thread::atomicx::mutex::IsShared ( )`
Check how many shared locks are accquired.

Returns

size_t Number of threads holding shared locks

Definition at line 575 of file atomicx.cpp.

5.4.2.3 Lock()

`void thread::atomicx::mutex::Lock ( )`
Exclusive/binary lock the smart lock.

Note

Once Lock() methos is called, if any thread held a shared lock, the Lock will wait for it to finish in order to acquire the exclusive lock, and all other threads that needs to a shared lock will wait till Lock is accquired and released.

Definition at line 515 of file atomicx.cpp.

5.4.2.4 SharedLock()

`void thread::atomicx::mutex::SharedLock ( )`
Shared Lock for the smart Lock.

Note

Shared lock can only be accquired if no Exclusive lock is waiting or already accquired a exclusive lock, In contrast, if at least one thread holds a shared lock, any exclusive lock can only be accquired once it is released.

Definition at line 546 of file atomicx.cpp.

5.4.2.5 SharedUnlock()

`void thread::atomicx::mutex::SharedUnlock ( )`
Release the current shared lock.
Definition at line 561 of file atomicx.cpp.

5.4.2.6 Unlock()

`void thread::atomicx::mutex::Unlock ( )`
Release the exclusive lock.
Definition at line 530 of file atomicx.cpp.
The documentation for this class was generated from the following files:

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.5 thread::atomicx::queue< T >::QItem Class Reference

Queue Item object.
`#include <atomicx.hpp>`

Public Member Functions

- QItem ()=delete
- QItem (T &qItem)

  Queue Item constructor.
- T & GetItem ()

  Get the current object in the QItem.

Protected Member Functions

- void SetNext (QItem &qItem)

  Set Next Item in the Queue list.
- QItem * GetNext ()

  Get the Next QItem object, if any.

Friends

- class queue

5.5.1 Detailed Description

template<typename T>
class thread::atomicx::queue< T >::QItem

Queue Item object.
Definition at line 496 of file atomicx.hpp.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 QItem() [1/2]

```
template<typename T >
thread::atomicx::queue< T >::QItem::QItem ( )  [delete]
```

5.5.2.2 QItem() [2/2]

```
template<typename T >
thread::atomicx::queue< T >::QItem::QItem (
            T & qItem )  [inline]
```

Queue Item constructor.

Parameters

qItem	Obj tem- plate type T

Definition at line 506 of file atomicx.hpp.

5.5.3 Member Function Documentation

5.5.3.1 GetItem()

```
template<typename T >
T & thread::atomicx::queue< T >::QItem::GetItem ( )  [inline]
```

Get the current object in the QItem.

Returns

T& The template type T object

Definition at line 514 of file atomicx.hpp.

5.5.3.2 GetNext()

```
template<typename T >
QItem * thread::atomicx::queue< T >::QItem::GetNext ( )  [inline], [protected]
```

Get the Next QItem object, if any.

Returns

QItem* A valid QItem pointer otherwise nullptr

Definition at line 537 of file atomicx.hpp.

5.5.3.3 SetNext()

```
template<typename T >
void thread::atomicx::queue< T >::QItem::SetNext (
            QItem & qItem )  [inline], [protected]
```

Set Next Item in the Queue list.

Parameters

qItem	QItem that holds a Queue element

Definition at line 527 of file atomicx.hpp.

5.5.4 Friends And Related Function Documentation

5.5.4.1 queue

```
template<typename T >
friend class queue  [friend]
```

Definition at line 520 of file atomicx.hpp.

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

- atomicx/atomicx.hpp

5.6 thread::atomicx::queue< T > Class Template Reference

`#include <atomicx.hpp>`

Data Structures

- class QItem

  Queue Item object.

Public Member Functions

- queue ()=delete
- queue (size_t nQSize)

  Thread Safe Queue constructor.
- bool PushBack (T item)

  Push an object to the end of the queue, if the queue is full, it waits till there is a space.
- bool PushFront (T item)

  Push an object to the beggining of the queue, if the queue is full, it waits till there is a space.
- T Pop ()

  Pop an Item from the beggining of queue. Is no object there is no object in the queue, it waits for it.
- size_t GetSize ()

  Get the number of the objects in the queue.
- size_t GetMaxSize ()

  Get the Max number of object in the queue can hold.
- bool IsFull ()

  Check if the queue is full.

5.6.1 Detailed Description

template<typename T>
class thread::atomicx::queue< T >

5.6.1.1 QUEUE FOR IPC IMPLEMENTATION

Definition at line 327 of file atomicx.hpp.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 queue() [1/2]

```
template<typename T >
thread::atomicx::queue< T >::queue ( )  [delete]
```

5.6.2.2 queue() [2/2]

```
template<typename T >
thread::atomicx::queue< T >::queue (
            size_t nQSize )  [inline]
```

Thread Safe Queue constructor.

Parameters

nQSize	Max number of objects to hold

Definition at line 338 of file atomicx.hpp.

5.6.3 Member Function Documentation

5.6.3.1 GetMaxSize()

```
template<typename T >
size_t thread::atomicx::queue< T >::GetMaxSize ( )  [inline]
```

Get the Max number of object in the queue can hold.

Returns

size_t The max number of object

Definition at line 476 of file atomicx.hpp.

5.6.3.2 GetSize()

```
template<typename T >
size_t thread::atomicx::queue< T >::GetSize ( )  [inline]
```

Get the number of the objects in the queue.

Returns

size_t Number of the objects in the queue

Definition at line 466 of file atomicx.hpp.

5.6.3.3 IsFull()

```
template<typename T >
bool thread::atomicx::queue< T >::IsFull ( )  [inline]
```

Check if the queue is full.

Returns

true for yes, otherwise false

Definition at line 486 of file atomicx.hpp.

5.6.3.4 Pop()

```
template<typename T >
T thread::atomicx::queue< T >::Pop ( )  [inline]
```

Pop an Item from the beggining of queue. Is no object there is no object in the queue, it waits for it.

Returns

T return the object stored.

Definition at line 436 of file atomicx.hpp.

5.6.3.5 PushBack()

```
template<typename T >
bool thread::atomicx::queue< T >::PushBack (
            T item ) [inline]
```

Push an object to the end of the queue, if the queue is full, it waits till there is a space.

Parameters

item	The object to be pushed into the queue

Returns

true if it was able to push a object in the queue, false otherwise

Definition at line 349 of file atomicx.hpp.

5.6.3.6 PushFront()

```
template<typename T >
bool thread::atomicx::queue< T >::PushFront (
            T item ) [inline]
```

Push an object to the beggining of the queue, if the queue is full, it waits till there is a space.

Parameters

item	The object to be pushed into the queue

Returns

true if it was able to push a object in the queue, false otherwise

Definition at line 394 of file atomicx.hpp.
The documentation for this class was generated from the following file:

- atomicx/atomicx.hpp

5.7 thread::atomicx::semaphore Class Reference

```
#include <atomicx.hpp>
```

Public Member Functions

- semaphore (size_t nMaxShared)

  Construct a new semaphore with MaxShared allowed.

- bool acquire (atomicx_time nTimeout=0)

  Acquire a shared lock context, if already on max shared allowed, wait till one is release or timeout.
- void release ()

  Releases one shared lock.
- size_t GetCount ()

  Get How many shared locks at a given moment.
- size_t GetWaitCount ()

  Get how many waiting threads for accquiring context.
- size_t GetMaxAcquired ()

  Get the Max Acquired Number.

Static Public Member Functions

- static size_t GetMax ()

  Get the maximun accquired context possible.

5.7.1 Detailed Description

5.7.1.1 SEMAPHORES IMPLEMENTATION

Definition at line 563 of file atomicx.hpp.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 semaphore()

```
thread::atomicx::semaphore::semaphore (
            size_t nMaxShared )
```

Construct a new semaphore with MaxShared allowed.

Parameters

nMaxShred	Max shared lock

Definition at line 34 of file atomicx.cpp.

5.7.3 Member Function Documentation

5.7.3.1 acquire()

```
bool thread::atomicx::semaphore::acquire (
            atomicx_time nTimeout = 0 )
```

Acquire a shared lock context, if already on max shared allowed, wait till one is release or timeout.

Parameters

nTimeout	default = 0 (indefinitely), How long to wait of acc-quiring

Returns

true if it acquired the context, otherwise timeout returns false

Definition at line 43 of file atomicx.cpp.

5.7.3.2 GetCount()

`size_t thread::atomicx::semaphore::GetCount ( )`

Get How many shared locks at a given moment.

Returns

size_t Number of shared locks

Definition at line 70 of file atomicx.cpp.

5.7.3.3 GetMax()

`size_t thread::atomicx::semaphore::GetMax ( ) [static]`

Get the maximun accquired context possible.

Returns

size_t

Definition at line 38 of file atomicx.cpp.

5.7.3.4 GetMaxAcquired()

`size_t thread::atomicx::semaphore::GetMaxAcquired ( )`

Get the Max Acquired Number.

Returns

size_t The max acquired context number

Definition at line 75 of file atomicx.cpp.

5.7.3.5 GetWaitCount()

`size_t thread::atomicx::semaphore::GetWaitCount ( )`

Get how many waiting threads for accquiring context.

Returns

size_t Number of waiting threads

Definition at line 80 of file atomicx.cpp.

5.7.3.6 release()

```
void thread::atomicx::semaphore::release ( )
```

Releases one shared lock.
Definition at line 60 of file atomicx.cpp.
The documentation for this class was generated from the following files:

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.8 thread::atomicx::smart_ptr< T > Class Template Reference

```
#include <atomicx.hpp>
```

Public Member Functions

- smart_ptr (T *p)
  smart pointer constructor
- smart_ptr (const smart_ptr< T > &sa)
  smart pointer overload constructor
- smart_ptr< T > & operator= (const smart_ptr< T > &sa)
  Smart pointer Assignment operator.
- ~smart_ptr (void)
  Smart pointer destructor.
- T * operator-> (void)
  Smart pointer access operator.
- T & operator& (void)
  Smart pointer access operator.
- bool IsValid (void)
  Check if the referece still valid.
- size_t GetRefCounter (void)
  Get the Ref Counter of the managed pointer.

5.8.1 Detailed Description

template<typename T>
class thread::atomicx::smart_ptr< T >

5.8.1.1 SUPLEMENTAR SMART_PTR IMPLEMENTATION

Definition at line 197 of file atomicx.hpp.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 smart_ptr() [1/2]

```
template<typename T >
thread::atomicx::smart_ptr< T >::smart_ptr (
            T * p )  [inline]
```

smart pointer constructor

Parameters

p	pointer type T to be managed

Definition at line 206 of file atomicx.hpp.

5.8.2.2 smart_ptr() [2/2]

```
template<typename T >
thread::atomicx::smart_ptr< T >::smart_ptr (
            const smart_ptr< T > & sa )  [inline]
```

smart pointer overload constructor

Parameters

sa	Smart pointer reference

Definition at line 214 of file atomicx.hpp.

5.8.2.3 ~smart_ptr()

```
template<typename T >
thread::atomicx::smart_ptr< T >::~smart_ptr (
            void  )  [inline]
```

Smart pointer destructor.
Definition at line 247 of file atomicx.hpp.

5.8.3 Member Function Documentation

5.8.3.1 GetRefCounter()

```
template<typename T >
size_t thread::atomicx::smart_ptr< T >::GetRefCounter (
            void  )  [inline]
```

Get the Ref Counter of the managed pointer.

Returns

size_t How much active references

Definition at line 298 of file atomicx.hpp.

5.8.3.2 IsValid()

```
template<typename T >
bool thread::atomicx::smart_ptr< T >::IsValid (
            void  )  [inline]
```

Check if the referece still valid.

Returns

true if the reference still not null, otherwise false

Definition at line 288 of file atomicx.hpp.

5.8.3.3 operator&()

```
template<typename T >
T & thread::atomicx::smart_ptr< T >::operator& (
            void ) [inline]
```

Smart pointer access operator.

Returns

T* Reference for the managed object T

Definition at line 278 of file atomicx.hpp.

5.8.3.4 operator->()

```
template<typename T >
T * thread::atomicx::smart_ptr< T >::operator-> (
            void ) [inline]
```

Smart pointer access operator.

Returns

T* Pointer for the managed object T

Definition at line 268 of file atomicx.hpp.

5.8.3.5 operator=()

```
template<typename T >
smart_ptr< T > & thread::atomicx::smart_ptr< T >::operator= (
            const smart_ptr< T > & sa ) [inline]
```

Smart pointer Assignment operator.

Parameters

sa	Smart poiter refer- ence

Returns

smart_ptr<T>& smart pointer this reference.

Definition at line 227 of file atomicx.hpp.
The documentation for this class was generated from the following file:

- atomicx/atomicx.hpp

5.9 thread::atomicx::smartMutex Class Reference

RII compliance lock/shared lock to auto unlock on destruction.
`#include <atomicx.hpp>`

Public Member Functions

- smartMutex ()=delete
- smartMutex (mutex &lockObj)

  Construct a new Smart Lock object based a existing lock.
- ~smartMutex ()

  Destroy and release the smart lock taken.
- bool SharedLock ()

  Accquire a SharedLock.
- bool Lock ()

  Accquire a exclusive Lock.
- size_t IsShared ()

  Check how many shared locks are accquired.
- bool IsLocked ()

  Check if a exclusive lock has been already accquired.

5.9.1 Detailed Description

RII compliance lock/shared lock to auto unlock on destruction.
Definition at line 752 of file atomicx.hpp.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 smartMutex() [1/2]

```
thread::atomicx::smartMutex::smartMutex ( )  [delete]
```

5.9.2.2 smartMutex() [2/2]

```
thread::atomicx::smartMutex::smartMutex (
            mutex & lockObj )
```

Construct a new Smart Lock object based a existing lock.

Parameters

lockObj	the existing lock object

Definition at line 585 of file atomicx.cpp.

5.9.2.3 ~smartMutex()

```
thread::atomicx::smartMutex::~smartMutex ( )
```

Destroy and release the smart lock taken.
Definition at line 588 of file atomicx.cpp.

5.9.3 Member Function Documentation

5.9.3.1 IsLocked()

`bool thread::atomicx::smartMutex::IsLocked ( )`
Check if a exclusive lock has been already accquired.

Returns

true if yes, otherwise false

Definition at line 634 of file atomicx.cpp.

5.9.3.2 IsShared()

`size_t thread::atomicx::smartMutex::IsShared ( )`
Check how many shared locks are accquired.

Returns

size_t Number of threads holding shared locks

Definition at line 629 of file atomicx.cpp.

5.9.3.3 Lock()

`bool thread::atomicx::smartMutex::Lock ( )`
Accquire a exclusive Lock.

Returns

true if accquired, false if another accquisition was already done

Definition at line 615 of file atomicx.cpp.

5.9.3.4 SharedLock()

`bool thread::atomicx::smartMutex::SharedLock ( )`
Accquire a SharedLock.

Returns

true if accquired, false if another accquisition was already done

Definition at line 601 of file atomicx.cpp.
The documentation for this class was generated from the following files:

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.10 thread::atomicx::smartSemaphore Class Reference

`#include <atomicx.hpp>`

Public Member Functions

- smartSemaphore (atomicx::semaphore &sem)

  Acquire and managed the semaphore.
- smartSemaphore ()=delete
- ~smartSemaphore ()

  Destroy the smart Semaphore while releasing it.
- bool acquire (atomicx_time nTimeout=0)

Acquire a shared lock context, if already on max shared allowed, wait till one is release or timeout.

- void release ()

  Releases one shared lock.

- size_t GetCount ()

  Get How many shared locks at a given moment.

- size_t GetWaitCount ()

  Get how many waiting threads for accquiring context.

- size_t GetMaxAcquired ()

  Get the Max Acquired Number.

- bool IsAcquired ()

  Report if the smartSemaphore has acquired a shared context.

Static Public Member Functions

- static size_t GetMax ()

  Get the maximun accquired context possible.

5.10.1 Detailed Description

Definition at line 620 of file atomicx.hpp.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 smartSemaphore() [1/2]

```
thread::atomicx::smartSemaphore::smartSemaphore (
            atomicx::semaphore & sem )
```

Acquire and managed the semaphore.

Parameters

sem	base semaphore

Definition at line 86 of file atomicx.cpp.

5.10.2.2 smartSemaphore() [2/2]

```
thread::atomicx::smartSemaphore::smartSemaphore ( )  [delete]
```

5.10.2.3 ~smartSemaphore()

```
thread::atomicx::smartSemaphore::~smartSemaphore ( )
```

Destroy the smart Semaphore while releasing it.
Definition at line 89 of file atomicx.cpp.

5.10.3 Member Function Documentation

5.10.3.1 acquire()

```
bool thread::atomicx::smartSemaphore::acquire (
            atomicx_time nTimeout = 0 )
```

Acquire a shared lock context, if already on max shared allowed, wait till one is release or timeout.

Parameters

nTimeout	default = 0 (indefinitely), How long to wait of acquiring

Returns

true if it acquired the context, otherwise timeout returns false

Definition at line 94 of file atomicx.cpp.

5.10.3.2 GetCount()

```
size_t thread::atomicx::smartSemaphore::GetCount ( )
```

Get How many shared locks at a given moment.

Returns

size_t Number of shared locks

Definition at line 116 of file atomicx.cpp.

5.10.3.3 GetMax()

```
static size_t thread::atomicx::smartSemaphore::GetMax ( )  [static]
```

Get the maximun accquired context possible.

Returns

size_t

5.10.3.4 GetMaxAcquired()

```
size_t thread::atomicx::smartSemaphore::GetMaxAcquired ( )
```

Get the Max Acquired Number.

Returns

size_t The max acquired context number

Definition at line 121 of file atomicx.cpp.

5.10.3.5 GetWaitCount()

`size_t thread::atomicx::smartSemaphore::GetWaitCount ( )`
Get how many waiting threads for accquiring context.

Returns

size_t Number of waiting threads

Definition at line 126 of file atomicx.cpp.

5.10.3.6 IsAcquired()

`bool thread::atomicx::smartSemaphore::IsAcquired ( )`
Report if the smartSemaphore has acquired a shared context.

Returns

true if it has successfully acquired a shared context otherwise false

Definition at line 131 of file atomicx.cpp.

5.10.3.7 release()

`void thread::atomicx::smartSemaphore::release ( )`
Releases one shared lock.
Definition at line 108 of file atomicx.cpp.
The documentation for this class was generated from the following files:

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

5.11 thread::atomicx::Timeout Class Reference

Timeout Check object.
`#include <atomicx.hpp>`

Public Member Functions

- Timeout ()=delete
- Timeout (atomicx_time nTimoutValue)

  Construct a new Timeout object.
- void Set (atomicx_time nTimoutValue)

  Set a timeout from now.
- bool IsTimedout ()

  Check wether it has timeout.
- atomicx_time GetRemaining ()

  Get the remaining time till timeout.
- atomicx_time GetDurationSince (atomicx_time startTime)

  Get the Time Since specific point in time.

5.11.1 Detailed Description

Timeout Check object.
Definition at line 88 of file atomicx.hpp.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Timeout() [1/2]

```
thread::atomicx::Timeout::Timeout ( )  [delete]
```

5.11.2.2 Timeout() [2/2]

```
thread::atomicx::Timeout::Timeout (
            atomicx_time nTimoutValue )
```

Construct a new Timeout object.

Parameters

nTimoutValue	Timeout value to be calculated

Note

To decrease the amount of memory, Timeout does not save the start time. Special use case: if nTimeoutValue == 0, IsTimedout is always false.

Definition at line 136 of file atomicx.cpp.

5.11.3 Member Function Documentation

5.11.3.1 GetDurationSince()

```
atomicx_time thread::atomicx::Timeout::GetDurationSince (
            atomicx_time startTime )
```

Get the Time Since specific point in time.

Parameters

startTime	The specific point in time

Returns

atomicx_time How long since the point in time

Note

To decrease the amount of memory, Timeout does not save the start time.

Definition at line 158 of file atomicx.cpp.

5.11.3.2 GetRemaining()

```
atomicx_time thread::atomicx::Timeout::GetRemaining ( )
```

Get the remaining time till timeout.

Returns

atomicx_time Remaining time till timeout, otherwise 0;

Definition at line 151 of file atomicx.cpp.

5.11.3.3 IsTimedout()

```
bool thread::atomicx::Timeout::IsTimedout ( )
```

Check wether it has timeout.

Returns

true if it timeout otherwise 0

Definition at line 146 of file atomicx.cpp.

5.11.3.4 Set()

```
void thread::atomicx::Timeout::Set (
            atomicx_time nTimoutValue )
```

Set a timeout from now.

Parameters

nTimoutValue	timeout in atomicx_time

Definition at line 141 of file atomicx.cpp.

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

- atomicx/atomicx.hpp
- atomicx/atomicx.cpp

Chapter 6

File Documentation

6.1 atomicx/atomicx.cpp File Reference

```
#include "atomicx.hpp"
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdint.h>
#include <setjmp.h>
#include <stdlib.h>
```

Namespaces

- namespace thread

Macros

- #define POLY 0x8408

Functions

- void yield (void)

6.1.1 Macro Definition Documentation

6.1.1.1 POLY

```
#define POLY 0x8408
```

6.1.2 Function Documentation

6.1.2.1 yield()

```
void yield (
            void  )
```

Definition at line 22 of file atomicx.cpp.

6.2 atomicx.cpp

Go to the documentation of this file.

```
00001 //
00002 //  atomic.cpp
00003 //  atomic
00004 //
00005 //  Created by GUSTAVO CAMPOS on 29/08/2021.
00006 //
00007
00008 #include "atomicx.hpp"
00009
00010 #include <stdio.h>
00011 #include <unistd.h>
00012
00013 #include <string.h>
00014 #include <stdint.h>
00015 #include <setjmp.h>
00016
00017 #include <stdlib.h>
00018
00019 extern "C"
00020 {
00021     #pragma weak yield
00022     void yield(void)
00023     {}
00024 }
00025
00026 namespace thread
00027 {
00028     // Static initializations
00029     static atomicx* ms_paFirst=nullptr;
00030     static atomicx* ms_paLast=nullptr;
00031     static jmp_buf ms_joinContext{};
00032     static atomicx* ms_pCurrent=nullptr;
00033
00034     atomicx::semaphore::semaphore(size_t nMaxShared) : m_maxShared(nMaxShared)
00035     {
00036     }
00037
00038     size_t atomicx::semaphore::GetMax ()
00039     {
00040         return (size_t) ~0;
00041     }
00042
00043     bool atomicx::semaphore::acquire(atomicx_time nTimeout)
00044     {
00045         Timeout timeout(nTimeout);
00046
00047         while (m_counter >= m_maxShared)
00048         {
00049             if (timeout.IsTimedout () || GetCurrent()->Wait (*this, 1, timeout.GetRemaining()) ==
      false)
00050             {
00051                 return false;
00052             }
00053         }
00054
00055         m_counter++;
00056
00057         return true;
00058     }
00059
00060     void atomicx::semaphore::release()
00061     {
00062         if (m_counter)
00063         {
00064             m_counter --;
00065
00066             GetCurrent()->Notify (*this, 1, NotifyType::one);
00067         }
00068     }
00069
00070     size_t atomicx::semaphore::GetCount ()
00071     {
00072         return m_counter;
00073     }
00074
00075     size_t atomicx::semaphore::GetMaxAcquired ()
00076     {
00077         return m_maxShared;
00078     }
00079
00080     size_t atomicx::semaphore::GetWaitCount ()
00081     {
00082         return GetCurrent()->HasWaitings (*this, 1);
```

```
00083     }
00084
00085     // Smart Semaphore, manages the semaphore com comply with RII
00086     atomicx::smartSemaphore::smartSemaphore(semaphore& sem) : m_sem(sem)
00087     {}
00088
00089     atomicx::smartSemaphore::~smartSemaphore()
00090     {
00091         release ();
00092     }
00093
00094     bool atomicx::smartSemaphore::acquire(atomicx_time nTimeout)
00095     {
00096         if (bAcquired == false && m_sem.acquire (nTimeout))
00097         {
00098             bAcquired = true;
00099         }
00100         else
00101         {
00102             return false;
00103         }
00104
00105         return true;
00106     }
00107
00108     void atomicx::smartSemaphore::release()
00109     {
00110         if (bAcquired)
00111         {
00112             m_sem.release ();
00113         }
00114     }
00115
00116     size_t atomicx::smartSemaphore::GetCount ()
00117     {
00118         return m_sem.GetCount ();
00119     }
00120
00121     size_t atomicx::smartSemaphore::GetMaxAcquired ()
00122     {
00123         return m_sem.GetMaxAcquired();
00124     }
00125
00126     size_t atomicx::smartSemaphore::GetWaitCount ()
00127     {
00128         return m_sem.GetWaitCount ();
00129     }
00130
00131     bool atomicx::smartSemaphore::IsAcquired ()
00132     {
00133         return bAcquired;
00134     }
00135
00136     atomicx::Timeout::Timeout (atomicx_time nTimeoutValue) : m_timeoutValue (0)
00137     {
00138         Set (nTimeoutValue);
00139     }
00140
00141     void atomicx::Timeout::Set(atomicx_time nTimeoutValue)
00142     {
00143         m_timeoutValue = nTimeoutValue ? nTimeoutValue + Atomicx_GetTick () : 0;
00144     }
00145
00146     bool atomicx::Timeout::IsTimedout()
00147     {
00148         return (m_timeoutValue == 0 || Atomicx_GetTick () < m_timeoutValue) ? false : true;
00149     }
00150
00151     atomicx_time atomicx::Timeout::GetRemaining()
00152     {
00153         auto nNow = Atomicx_GetTick ();
00154
00155         return (nNow < m_timeoutValue) ? m_timeoutValue - nNow : 0;
00156     }
00157
00158     atomicx_time atomicx::Timeout::GetDurationSince(atomicx_time startTime)
00159     {
00160         return startTime - GetRemaining ();
00161     }
00162
00163     atomicx::aiterator::aiterator(atomicx* ptr) : m_ptr(ptr)
00164     {}
00165
00166     atomicx& atomicx::aiterator::operator*() const
00167     {
00168         return *m_ptr;
00169     }
```

```
00170
00171     atomicx* atomicx::aiterator::operator->()
00172     {
00173         return m_ptr;
00174     }
00175
00176     atomicx::aiterator& atomicx::aiterator::operator++()
00177     {
00178         if (m_ptr != nullptr) m_ptr = m_ptr->m_paNext;
00179         return *this;
00180     }
00181
00182     atomicx::aiterator atomicx::begin() { return aiterator(ms_paFirst); }
00183     atomicx::aiterator atomicx::end()   { return aiterator(nullptr); }
00184
00185     atomicx* atomicx::GetCurrent()
00186     {
00187         return ms_pCurrent;
00188     }
00189
00190     void atomicx::AddThisThread()
00191     {
00192         if (ms_paFirst == nullptr)
00193         {
00194             ms_paFirst = this;
00195             ms_paLast = ms_paFirst;
00196         }
00197         else
00198         {
00199             this->m_paPrev = ms_paLast;
00200             ms_paLast->m_paNext = this;
00201             ms_paLast = this;
00202         }
00203     }
00204
00205     bool atomicx::SelectNextThread()
00206     {
00207         atomicx* pItem = ms_paFirst;
00208
00209         do
00210         {
00211             if (ms_pCurrent->m_nTargetTime == 0 && pItem->m_nTargetTime > 0)
00212             {
00213                 ms_pCurrent = pItem;
00214             }
00215
00216             if (pItem->m_aStatus == aTypes::start || pItem->m_aStatus == aTypes::now)
00217             {
00218                 ms_pCurrent = pItem;
00219                 break;
00220             }
00221             else if ((pItem->m_aStatus == aTypes::sleep || (pItem->m_aStatus == aTypes::wait &&
    pItem->m_nTargetTime > 0)) && ms_pCurrent->m_nTargetTime >= pItem->m_nTargetTime)
00222             {
00223                 ms_pCurrent = pItem;
00224             }
00225
00226         } while ((pItem = pItem->m_paNext));
00227
00228         if (ms_pCurrent == nullptr)
00229         {
00230             return false;
00231         }
00232         else
00233         {
00234             switch (ms_pCurrent->m_aStatus)
00235             {
00236                 case aTypes::wait:
00237                     if (ms_pCurrent->m_nTargetTime > 0)
00238                     {
00239                         ms_pCurrent->m_aSubStatus = aSubTypes::timeout;
00240                     }
00241                     else
00242                     {
00243                         // A blocked wait should never come here if the system was not in deadlock.
00244                         return false;
00245                     }
00246
00247                 case aTypes::sleep:
00248                 {
00249                     atomicx_time nCurrent = Atomicx_GetTick();
00250
00251                     (void) Atomicx_SleepTick(nCurrent < ms_pCurrent->m_nTargetTime ?
    ms_pCurrent->m_nTargetTime - nCurrent : 0);
00252
00253                     break;
00254                 }
```

```
00255
00256                 case aTypes::running:
00257                 case aTypes::start:
00258                 case aTypes::now:
00259                     break;
00260
00261                 default:
00262                     return false;
00263             }
00264         }
00265
00266         return true;
00267     }
00268
00269     bool atomicx::Start(void)
00270     {
00271         if (ms_paFirst != nullptr)
00272         {
00273             static bool nRunning = true;
00274
00275             ms_pCurrent = ms_paFirst;
00276
00277             while (nRunning && SelectNextThread ())
00278             {
00279                 if (setjmp(ms_joinContext) == 0)
00280                 {
00281                     if (ms_pCurrent->m_aStatus == aTypes::start)
00282                     {
00283                         volatile uint8_t nStackStart=0;
00284
00285                         ms_pCurrent->m_aStatus = aTypes::running;
00286
00287                         ms_pCurrent->m_pStaskStart = &nStackStart;
00288
00289                         ms_pCurrent->m_lastResumeUserTime = Atomicx_GetTick ();
00290
00291                         ms_pCurrent->run();
00292
00293                         ms_pCurrent->m_aStatus = aTypes::start;
00294
00295                         ms_pCurrent->finish ();
00296                     }
00297                     else
00298                     {
00299                         longjmp(ms_pCurrent->m_context, 1);
00300                     }
00301                 }
00302             }
00303         }
00304
00305         return false;
00306     }
00307
00308     bool atomicx::Yield(atomicx_time nSleep)
00309     {
00310         m_LastUserExecTime = GetCurrentTick () - m_lastResumeUserTime;
00311
00312         if (m_aStatus == aTypes::running)
00313         {
00314             m_aStatus = aTypes::sleep;
00315             m_aSubStatus = aSubTypes::ok;
00316             m_nTargetTime=Atomicx_GetTick() + (nSleep == ATOMICX_TIME_MAX ? m_nice : nSleep);
00317         }
00318         else if (m_aStatus == aTypes::wait)
00319         {
00320             m_nTargetTime=nSleep > 0 ? nSleep + Atomicx_GetTick() : 0;
00321         }
00322         else
00323         {
00324             m_nTargetTime = (atomicx_time)~0;
00325         }
00326
00327         volatile uint8_t nStackEnd=0;
00328         m_pStaskEnd = &nStackEnd;
00329         m_stacUsedkSize = (size_t) (m_pStaskStart - m_pStaskEnd);
00330
00331         if (m_stacUsedkSize > m_stackSize || m_stack == nullptr)
00332         {
00333             /*
00334             * Controll the auto-stack memory
00335             * Note: Due to some small microcontroller
00336             *   does not have try/catch/throw by default
00337             *   I decicde to use malloc/free instead
00338             *   to control errors
00339             */
00340
00341             if (m_flags.autoStack == true)
```

```
00342             {
00343                 if (m_stack != nullptr)
00344                 {
00345                     free ((void*) m_stack);
00346                 }
00347 
00348                 if (m_stacUsedkSize > m_stackSize)
00349                 {
00350                     m_stackSize = m_stacUsedkSize + m_stackIncreasePace;
00351                 }
00352 
00353                 if ((m_stack = (uint8_t*) malloc (m_stacUsedkSize)) == nullptr)
00354                 {
00355                     m_aStatus = aTypes::stackOverflow;
00356                 }
00357             }
00358             else
00359             {
00360                 m_aStatus = aTypes::stackOverflow;
00361             }
00362 
00363             if (m_aStatus == aTypes::stackOverflow)
00364             {
00365                 (void) StackOverflowHandler();
00366                 abort();
00367             }
00368         }
00369 
00370         if (m_aStatus != aTypes::stackOverflow && memcpy((void*)m_stack, (const void*) m_pStaskEnd,
      m_stacUsedkSize) != (void*) m_stack)
00371         {
00372             return false;
00373         }
00374 
00375         if (setjmp(m_context) == 0)
00376         {
00377             longjmp(ms_joinContext, 1);
00378         }
00379         else
00380         {
00381             ms_pCurrent->m_stacUsedkSize = (size_t) (ms_pCurrent->m_pStaskStart - &nStackEnd);
00382             if (memcpy((void*) ms_pCurrent->m_pStaskEnd, (const void*) ms_pCurrent->m_stack,
      ms_pCurrent->m_stacUsedkSize) != (void*) ms_pCurrent->m_pStaskEnd)
00383             {
00384                 return false;
00385             }
00386         }
00387 
00388         ms_pCurrent->m_lastResumeUserTime = GetCurrentTick ();
00389 
00390         ms_pCurrent->m_aStatus = aTypes::running;
00391 
00392         if (ms_pCurrent->m_flags.dynamicNice == true)
00393         {
00394             ms_pCurrent->m_nice = ((ms_pCurrent->m_LastUserExecTime) + ms_pCurrent->m_nice) / 2;
00395         }
00396 
00397         return true;
00398     }
00399 
00400     void atomicx::YieldNow ()
00401     {
00402         m_aStatus = aTypes::now;
00403         m_aSubStatus = aSubTypes::ok;
00404         Yield ();
00405     }
00406 
00407     atomicx_time atomicx::GetNice(void)
00408     {
00409         return m_nice;
00410     }
00411 
00412     size_t atomicx::GetStackSize(void)
00413     {
00414         return m_stackSize;
00415     }
00416 
00417     size_t atomicx::GetUsedStackSize(void)
00418     {
00419         return m_stacUsedkSize;
00420     }
00421 
00422     void atomicx::RemoveThisThread()
00423     {
00424         if (m_paNext == nullptr && m_paPrev == nullptr)
00425         {
00426             ms_paFirst = nullptr;
```

```
00427             m_paPrev = nullptr;
00428             ms_pCurrent = nullptr;
00429         }
00430         else if (m_paPrev == nullptr)
00431         {
00432             m_paNext->m_paPrev = nullptr;
00433             ms_paFirst = m_paNext;
00434             ms_pCurrent = ms_paFirst;
00435         }
00436         else if (m_paNext == nullptr)
00437         {
00438             m_paPrev->m_paNext = nullptr;
00439             ms_paLast = m_paPrev;
00440             ms_pCurrent = ms_paFirst;
00441         }
00442         else
00443         {
00444             m_paPrev->m_paNext = m_paNext;
00445             m_paNext->m_paPrev = m_paPrev;
00446             ms_pCurrent = m_paNext->m_paPrev;
00447         }
00448     }
00449
00450     void atomicx::SetDefaultParameters ()
00451     {
00452         m_flags.autoStack = false;
00453         m_flags.dynamicNice = false;
00454     }
00455
00456     atomicx::atomicx(size_t nStackSize, int nStackIncreasePace) : m_context{},
    m_stackSize(nStackSize), m_stackIncreasePace(nStackIncreasePace), m_stack(nullptr)
00457     {
00458         SetDefaultParameters ();
00459
00460         m_flags.autoStack = true;
00461
00462         AddThisThread();
00463     }
00464
00465     atomicx::~atomicx()
00466     {
00467         RemoveThisThread();
00468
00469         if (m_flags.autoStack == true && m_stack != nullptr)
00470         {
00471             free((void*)m_stack);
00472         }
00473     }
00474
00475     const char* atomicx::GetName(void)
00476     {
00477         return "thread";
00478     }
00479
00480     void atomicx::SetNice (atomicx_time nice)
00481     {
00482         m_nice = nice;
00483     }
00484
00485     size_t atomicx::GetID(void)
00486     {
00487         return (size_t) this;
00488     }
00489
00490     atomicx_time atomicx::GetTargetTime(void)
00491     {
00492         return m_nTargetTime;
00493     }
00494
00495     int atomicx::GetStatus(void)
00496     {
00497         return static_cast<int>(m_aStatus);
00498     }
00499
00500     int atomicx::GetSubStatus(void)
00501     {
00502         return static_cast<int>(m_aSubStatus);
00503     }
00504
00505     size_t atomicx::GetReferenceLock(void)
00506     {
00507         return (size_t) m_pLockId;
00508     }
00509
00510     size_t atomicx::GetTagLock(void)
00511     {
00512         return (size_t) m_lockMessage.tag;
```

```
00513     }
00514
00515     void atomicx::mutex::Lock()
00516     {
00517         auto pAtomic = atomicx::GetCurrent();
00518
00519         if(pAtomic == nullptr) return;
00520
00521         // Get exclusive mutex
00522         while (bExclusiveLock && pAtomic->Wait(bExclusiveLock, 1));
00523
00524         bExclusiveLock = true;
00525
00526         // Wait all shared locks to be done
00527         while (nSharedLockCount && pAtomic->Wait(nSharedLockCount,2));
00528     }
00529
00530     void atomicx::mutex::Unlock()
00531     {
00532         auto pAtomic = atomicx::GetCurrent();
00533
00534         if(pAtomic == nullptr) return;
00535
00536         if (bExclusiveLock == true)
00537         {
00538             bExclusiveLock = false;
00539
00540             // Notify Other locks procedures
00541             pAtomic->Notify(nSharedLockCount, 2, NotifyType::all);
00542             pAtomic->Notify(bExclusiveLock, 1, NotifyType::one);
00543         }
00544     }
00545
00546     void atomicx::mutex::SharedLock()
00547     {
00548         auto pAtomic = atomicx::GetCurrent();
00549
00550         if(pAtomic == nullptr) return;
00551
00552         // Wait for exclusive mutex
00553         while (bExclusiveLock > 0 && pAtomic->Wait(bExclusiveLock, 1));
00554
00555         nSharedLockCount++;
00556
00557         // Notify Other locks procedures
00558         pAtomic->Notify (nSharedLockCount, 2, NotifyType::one);
00559     }
00560
00561     void atomicx::mutex::SharedUnlock()
00562     {
00563         auto pAtomic = atomicx::GetCurrent();
00564
00565         if(pAtomic == nullptr) return;
00566
00567         if (nSharedLockCount)
00568         {
00569             nSharedLockCount--;
00570
00571             pAtomic->Notify(nSharedLockCount, 2, NotifyType::one);
00572         }
00573     }
00574
00575     size_t atomicx::mutex::IsShared()
00576     {
00577         return nSharedLockCount;
00578     }
00579
00580     bool atomicx::mutex::IsLocked()
00581     {
00582         return bExclusiveLock;
00583     }
00584
00585     atomicx::smartMutex::smartMutex (mutex& lockObj) : m_lock(lockObj)
00586     {}
00587
00588     atomicx::smartMutex::~smartMutex()
00589     {
00590         switch (m_lockType)
00591         {
00592             case 'L':
00593                 m_lock.Unlock();
00594                 break;
00595             case 'S':
00596                 m_lock.SharedUnlock();
00597                 break;
00598         }
00599     }
```

```
00600
00601     bool atomicx::smartMutex::SharedLock()
00602     {
00603         bool bRet = false;
00604
00605         if (m_lockType == '\0')
00606         {
00607             m_lock.SharedLock();
00608             m_lockType = 'S';
00609             bRet = true;
00610         }
00611
00612         return bRet;
00613     }
00614
00615     bool atomicx::smartMutex::Lock()
00616     {
00617         bool bRet = false;
00618
00619         if (m_lockType == '\0')
00620         {
00621             m_lock.Lock();
00622             m_lockType = 'L';
00623             bRet = true;
00624         }
00625
00626         return bRet;
00627     }
00628
00629     size_t atomicx::smartMutex::IsShared()
00630     {
00631         return m_lock.IsShared();
00632     }
00633
00634     bool atomicx::smartMutex::IsLocked()
00635     {
00636         return m_lock.IsLocked();
00637     }
00638
00639     uint16_t atomicx::crc16(const uint8_t* pData, size_t nSize, uint16_t nCRC)
00640     {
00641         #define POLY 0x8408
00642         unsigned char nCount;
00643         unsigned int data;
00644
00645         nCRC = ~nCRC;
00646
00647         if (nSize == 0) return (~nCRC);
00648
00649         do
00650         {
00651             for (nCount = 0, data = (unsigned int)0xff & *pData++; nCount < 8; nCount++, data >>= 1)
00652             {
00653                 if ((nCRC & 0x0001) ^ (data & 0x0001))
00654                     nCRC = (nCRC >> 1) ^ POLY;
00655                 else
00656                     nCRC >>= 1;
00657             }
00658         } while (--nSize);
00659
00660         nCRC = ~nCRC;
00661         data = nCRC;
00662         nCRC = (nCRC << 8) | (data >> 8 & 0xff);
00663
00664         return (nCRC);
00665     }
00666
00667     uint32_t atomicx::GetTopicID (const char* pszTopic, size_t nKeyLenght)
00668     {
00669         return ((uint32_t) ((crc16 ((const uint8_t*)pszTopic, nKeyLenght, 0) << 15) | crc16 ((const
      uint8_t*)pszTopic, nKeyLenght, 0x8408)));
00670     }
00671
00672     bool atomicx::HasSubscriptions (const char* pszKey, size_t nKeyLenght)
00673     {
00674         if (pszKey != nullptr && nKeyLenght > 0)
00675         {
00676             uint32_t nKeyID = GetTopicID(pszKey, nKeyLenght);
00677
00678             for (auto& thr : *this)
00679             {
00680                 if (nKeyID == thr.m_TopicId || IsSubscribed (pszKey, nKeyLenght))
00681                 {
00682                     return true;
00683                 }
00684             }
00685         }
```

```
00686
00687         return false;
00688     }
00689
00690     bool atomicx::HasSubscriptions (uint32_t nKeyID)
00691     {
00692         for (auto& thr : *this)
00693         {
00694             if (nKeyID == thr.m_TopicId)
00695             {
00696                 return true;
00697             }
00698         }
00699
00700         return false;
00701     }
00702
00703     bool atomicx::WaitBrokerMessage (const char* pszKey, size_t nKeyLenght, Message& message)
00704     {
00705         if (pszKey != nullptr && nKeyLenght > 0)
00706         {
00707             m_aStatus = aTypes::subscription;
00708
00709             m_TopicId = GetTopicID(pszKey, nKeyLenght);
00710             m_nTargetTime = Atomicx_GetTick();
00711
00712             m_lockMessage = {0,0};
00713
00714             Yield();
00715
00716             message.message = m_lockMessage.message;
00717             message.tag = m_lockMessage.tag;
00718
00719             return true;
00720         }
00721
00722         return false;
00723     }
00724
00725     bool atomicx::WaitBrokerMessage (const char* pszKey, size_t nKeyLenght)
00726     {
00727         if (pszKey != nullptr && nKeyLenght > 0)
00728         {
00729             m_aStatus = aTypes::subscription;
00730
00731             m_TopicId = GetTopicID(pszKey, nKeyLenght);
00732             m_nTargetTime = Atomicx_GetTick();
00733
00734             Yield();
00735
00736             m_lockMessage = {0,0};
00737
00738             return true;
00739         }
00740
00741         return false;
00742     }
00743
00744     bool atomicx::SafePublish (const char* pszKey, size_t nKeyLenght, const Message message)
00745     {
00746         size_t nCounter=0;
00747
00748         if (pszKey != nullptr && nKeyLenght > 0)
00749         {
00750             uint32_t nTagId = GetTopicID(pszKey, nKeyLenght);
00751
00752             for (auto& thr : *this)
00753             {
00754                 if (nTagId == thr.m_TopicId)
00755                 {
00756                     nCounter++;
00757
00758                     thr.m_aStatus = aTypes::now;
00759                     thr.m_TopicId = 0;
00760                     thr.m_nTargetTime = Atomicx_GetTick();
00761                     thr.m_lockMessage.message = message.message;
00762                     thr.m_lockMessage.tag = message.tag;
00763                 }
00764
00765                 if (thr.IsSubscribed(pszKey, nKeyLenght))
00766                 {
00767                     nCounter++;
00768
00769                     thr.BrokerHandler(pszKey, nKeyLenght, thr.m_lockMessage);
00770                 }
00771             }
00772         }
```

```
00773
00774         return nCounter ? true : false;
00775     }
00776
00777     bool atomicx::Publish (const char* pszKey, size_t nKeyLenght, const Message message)
00778     {
00779         bool nReturn = SafePublish(pszKey, nKeyLenght, message);
00780
00781         if (nReturn) Yield();
00782
00783         return nReturn;
00784     }
00785
00786     bool atomicx::SafePublish (const char* pszKey, size_t nKeyLenght)
00787     {
00788         size_t nCounter=0;
00789
00790         if (pszKey != nullptr && nKeyLenght > 0)
00791         {
00792             uint32_t nTagId = GetTopicID(pszKey, nKeyLenght);
00793
00794             for (auto& thr : *this)
00795             {
00796                 if (nTagId == thr.m_TopicId)
00797                 {
00798                     nCounter++;
00799
00800                     thr.m_aStatus = aTypes::now;
00801                     thr.m_TopicId = 0;
00802                     thr.m_nTargetTime = Atomicx_GetTick();
00803                     thr.m_lockMessage = {0,0};
00804                 }
00805
00806                 if (thr.IsSubscribed(pszKey, nKeyLenght))
00807                 {
00808                     nCounter++;
00809
00810                     thr.m_lockMessage = {0,0};
00811                     thr.BrokerHandler(pszKey, nKeyLenght, thr.m_lockMessage);
00812                 }
00813
00814             }
00815
00816             if (nCounter) Yield();
00817         }
00818
00819         return nCounter ? true : false;
00820     }
00821
00822     bool atomicx::Publish (const char* pszKey, size_t nKeyLenght)
00823     {
00824         bool nReturn = SafePublish(pszKey, nKeyLenght);
00825
00826         if (nReturn) Yield();
00827
00828         return nReturn;
00829     }
00830
00831     atomicx_time atomicx::GetCurrentTick(void)
00832     {
00833         return Atomicx_GetTick ();
00834     }
00835
00836     bool atomicx::IsStackSelfManaged(void)
00837     {
00838         return m_flags.autoStack;
00839     }
00840
00841     atomicx_time atomicx::GetLastUserExecTime()
00842     {
00843         return m_LastUserExecTime;
00844     }
00845
00846     void atomicx::SetStackIncreasePace(size_t nIncreasePace)
00847     {
00848         m_stackIncreasePace = nIncreasePace;
00849     }
00850
00851     size_t atomicx::GetStackIncreasePace(void)
00852     {
00853         return m_stackIncreasePace;
00854     }
00855
00856     void atomicx::SetDynamicNice(bool status)
00857     {
00858         m_flags.dynamicNice = status;
00859     }
```

```
00860
00861     bool atomicx::IsDynamicNiceOn()
00862     {
00863         return m_flags.dynamicNice;
00864     }
00865 }
```

6.3 atomicx/atomicx.hpp File Reference

```
#include <stdint.h>
#include <stdlib.h>
#include <setjmp.h>
```

Data Structures

- class thread::atomicx
- class thread::atomicx::Timeout

  Timeout Check object.
- class thread::atomicx::aiterator
- class thread::atomicx::smart_ptr< T >
- class thread::atomicx::queue< T >
- class thread::atomicx::queue< T >::QItem

  Queue Item object.
- class thread::atomicx::semaphore
- class thread::atomicx::smartSemaphore
- class thread::atomicx::mutex
- class thread::atomicx::smartMutex

  RII compliance lock/shared lock to auto unlock on destruction.
- struct thread::atomicx::Message

Namespaces

- namespace thread

Macros

- #define ATOMICX_VERSION "1.2.1"
- #define ATOMIC_VERSION_LABEL "AtomicX v" ATOMICX_VERSION " built at " __TIMESTAMP__
- #define ATOMICX_TIME_MAX ((atomicx_time) ~0)

Typedefs

- using atomicx_time = uint32_t

Functions

- void yield (void)
- atomicx_time Atomicx_GetTick (void)

  Implement the custom Tick acquisition.
- void Atomicx_SleepTick (atomicx_time nSleep)

  Implement a custom sleep, usually based in the same GetTick granularity.

6.3.1 Macro Definition Documentation

6.3.1.1 ATOMIC_VERSION_LABEL

```
#define ATOMIC_VERSION_LABEL "AtomicX v" ATOMICX_VERSION " built at " __TIMESTAMP__
```
Definition at line 17 of file atomicx.hpp.

6.3.1.2 ATOMICX_TIME_MAX

```
#define ATOMICX_TIME_MAX ((atomicx_time) ~0)
```
Definition at line 21 of file atomicx.hpp.

6.3.1.3 ATOMICX_VERSION

```
#define ATOMICX_VERSION "1.2.1"
```
Definition at line 16 of file atomicx.hpp.

6.3.2 Typedef Documentation

6.3.2.1 atomicx_time

```
using atomicx_time = uint32_t
```
Definition at line 19 of file atomicx.hpp.

6.3.3 Function Documentation

6.3.3.1 Atomicx_GetTick()

```
atomicx_time Atomicx_GetTick (
            void  )
```
Implement the custom Tick acquisition.

Returns

atomicx_time

6.3.3.2 Atomicx_SleepTick()

```
void Atomicx_SleepTick (
            atomicx_time nSleep )
```
Implement a custom sleep, usually based in the same GetTick granularity.

Parameters

nSleep	How long custom tick to wait

Note

This function is particularly special, since it give freedom to tweak the processor power consuption if necessary

6.3.3.3 yield()

```
void yield (
            void  )
```

Definition at line 22 of file atomicx.cpp.

6.4 atomicx.hpp

Go to the documentation of this file.

```
00001 //
00002 //  atomic.hpp
00003 //  atomic
00004 //
00005 //  Created by GUSTAVO CAMPOS on 29/08/2021.
00006 //
00007
00008 #ifndef atomic_hpp
00009 #define atomic_hpp
00010
00011 #include <stdint.h>
00012 #include <stdlib.h>
00013 #include <setjmp.h>
00014
00015 /* Official version */
00016 #define ATOMICX_VERSION "1.2.1"
00017 #define ATOMIC_VERSION_LABEL "AtomicX v" ATOMICX_VERSION " built at " __TIMESTAMP__
00018
00019 using atomicx_time = uint32_t;
00020
00021 #define ATOMICX_TIME_MAX ((atomicx_time) ~0)
00022
00023 extern "C"
00024 {
00025     extern void yield(void);
00026 }
00027
00033 extern atomicx_time Atomicx_GetTick(void);
00034
00043 extern void Atomicx_SleepTick(atomicx_time nSleep);
00044
00045 namespace thread
00046 {
00047     class atomicx
00048     {
00049     public:
00050
00056         enum class aTypes : uint8_t
00057         {
00058             start=1,
00059             running=5,
00060             now=6,
00061             stop=10,
00062             lock=50,
00063             wait=55,
00064             subscription=60,
00065             sleep=100,
00066             stackOverflow=255
00067         };
00068
00069         enum class aSubTypes : uint8_t
00070         {
00071             error=10,
00072             ok,
00073             look,
00074             wait,
00075             timeout
00076         };
00077
00078         enum class NotifyType : uint8_t
00079         {
00080             one = 0,
00081             all = 1
00082         };
00083
00088         class Timeout
```

```
00089         {
00090             public:
00091                 Timeout () = delete;
00092 
00102                 Timeout (atomicx_time nTimoutValue);
00103 
00109                 void Set(atomicx_time nTimoutValue);
00110 
00116                 bool IsTimedout();
00117 
00123                 atomicx_time GetRemaining();
00124 
00135                 atomicx_time GetDurationSince(atomicx_time startTime);
00136 
00137             private:
00138                 atomicx_time m_timeoutValue = 0;
00139         };
00145         class aiterator
00146         {
00147         public:
00148             aiterator() = delete;
00149 
00155             aiterator(atomicx* ptr);
00156 
00157             /*
00158              * Access operator
00159              */
00160             atomicx& operator*() const;
00161             atomicx* operator->();
00162 
00163             /*
00164              * Movement operator
00165              */
00166             aiterator& operator++();
00167 
00168             /*
00169              * Binary operators
00170              */
00171             friend bool operator== (const aiterator& a, const aiterator& b){ return a.m_ptr ==
      b.m_ptr;};
00172             friend bool operator!= (const aiterator& a, const aiterator& b){ return a.m_ptr !=
      b.m_ptr;};
00173 
00174         private:
00175             atomicx* m_ptr;
00176         };
00177 
00183         aiterator begin(void);
00184 
00190         aiterator end(void);
00191 
00197         template <typename T> class smart_ptr
00198         {
00199         public:
00200 
00206             smart_ptr(T* p) : pRef (new reference {p, 1})
00207             { }
00208 
00214             smart_ptr(const smart_ptr<T>& sa)
00215             {
00216                 pRef = sa.pRef;
00217                 pRef->nRC++;
00218             }
00219 
00227             smart_ptr<T>& operator=(const smart_ptr<T>& sa)
00228             {
00229                 if (pRef != nullptr && pRef->nRC > 0)
00230                 {
00231                     pRef->nRC--;
00232                 }
00233 
00234                 pRef = sa.pRef;
00235 
00236                 if (pRef != nullptr)
00237                 {
00238                     pRef->nRC++;
00239                 }
00240 
00241                 return *this;
00242             }
00243 
00247             ~smart_ptr(void)
00248             {
00249                 if (pRef != nullptr)
00250                 {
00251                     if (--pRef->nRC == 0)
00252                     {
```

```
00253                       delete pRef->pReference;
00254                       delete pRef;
00255                   }
00256                   else
00257                   {
00258                       pRef->nRC--;
00259                   }
00260               }
00261           }
00262
00268           T* operator-> (void)
00269           {
00270               return pRef->pReference;
00271           }
00272
00278           T& operator& (void)
00279           {
00280               return *pRef->pReference;
00281           }
00282
00288           bool IsValid(void)
00289           {
00290               return pRef == nullptr ? false : pRef->pReference == nullptr ? false : true;
00291           }
00292
00298           size_t GetRefCounter(void)
00299           {
00300               if (pRef != nullptr)
00301               {
00302                   return pRef->nRC;
00303               }
00304
00305               return 0;
00306           }
00307
00308       private:
00309
00310           smart_ptr(void) = delete;
00311           struct reference
00312           {
00313               T* pReference ;
00314               size_t nRC;
00315           };
00316
00317           reference* pRef=nullptr;
00318       };
00319
00326       template<typename T>
00327       class queue
00328       {
00329       public:
00330
00331           queue() = delete;
00332
00338           queue(size_t nQSize):m_nQSize(nQSize), m_nItens{0}
00339           {}
00340
00349           bool PushBack(T item)
00350           {
00351               if (m_nItens >= m_nQSize)
00352               {
00353                   if (atomicx::GetCurrent() != nullptr)
00354                   {
00355                       atomicx::GetCurrent()->Wait(*this,1);
00356                   }
00357                   else
00358                   {
00359                       return false;
00360                   }
00361               }
00362
00363               QItem* pQItem = new QItem(item);
00364
00365               if (m_pQIStart == nullptr)
00366               {
00367                   m_pQIStart = m_pQIEnd = pQItem;
00368               }
00369               else
00370               {
00371                   m_pQIEnd->SetNext(*pQItem);
00372                   m_pQIEnd = pQItem;
00373               }
00374
00375               m_nItens++;
00376
00377               if (atomicx::GetCurrent() != nullptr)
00378               {
```

```
00379                   atomicx::GetCurrent()->Notify(*this,0);
00380               }
00381
00382               return true;
00383           }
00384
00385
00394           bool PushFront(T item)
00395           {
00396               if (m_nItens >= m_nQSize)
00397               {
00398                   if (atomicx::GetCurrent() != nullptr)
00399                   {
00400                       atomicx::GetCurrent()->Wait(*this,1);
00401                   }
00402                   else
00403                   {
00404                       return false;
00405                   }
00406               }
00407
00408               QItem* pQItem = new QItem(item);
00409
00410               if (m_pQIStart == nullptr)
00411               {
00412                   m_pQIStart = m_pQIEnd = pQItem;
00413               }
00414               else
00415               {
00416                   pQItem->SetNext(*m_pQIStart);
00417                   m_pQIStart = pQItem;
00418               }
00419
00420               m_nItens++;
00421
00422               if (atomicx::GetCurrent() != nullptr)
00423               {
00424                   atomicx::GetCurrent()->Notify    (*this,0);
00425               }
00426
00427               return true;
00428           }
00429
00436           T Pop()
00437           {
00438               if (m_nItens == 0)
00439               {
00440                   atomicx::GetCurrent()->Wait(*this,0);
00441               }
00442
00443               T pItem = m_pQIStart->GetItem();
00444
00445               QItem* p_tmpQItem = m_pQIStart;
00446
00447               m_pQIStart = m_pQIStart->GetNext();
00448
00449               delete p_tmpQItem;
00450
00451               m_nItens--;
00452
00453               if (atomicx::GetCurrent() != nullptr)
00454               {
00455                   atomicx::GetCurrent()->Notify(*this,1);
00456               }
00457
00458               return pItem;
00459           }
00460
00466           size_t GetSize()
00467           {
00468               return m_nItens;
00469           }
00470
00476           size_t GetMaxSize()
00477           {
00478               return m_nQSize;
00479           }
00480
00486           bool IsFull()
00487           {
00488               return m_nItens >= m_nQSize;
00489           }
00490
00491       protected:
00492
00496           class QItem
00497           {
```

```
00498             public:
00499                 QItem () = delete;
00500 
00506                 QItem(T& qItem) : m_qItem(qItem), m_pNext(nullptr)
00507                 {}
00508 
00514                 T& GetItem()
00515                 {
00516                     return m_qItem;
00517                 }
00518 
00519             protected:
00520                 friend class queue;
00521 
00527                 void SetNext (QItem& qItem)
00528                 {
00529                     m_pNext = &qItem;
00530                 }
00531 
00537                 QItem* GetNext ()
00538                 {
00539                     return m_pNext;
00540                 }
00541 
00542             private:
00543 
00544                 T m_qItem;
00545                 QItem* m_pNext;
00546             };
00547 
00548         private:
00549             size_t m_nQSize;
00550             size_t m_nItens;
00551 
00552             QItem* m_pQIEnd = nullptr;
00553             QItem* m_pQIStart = nullptr;
00554 
00555         };
00556 
00563         class semaphore
00564         {
00565             public:
00571                 semaphore(size_t nMaxShared);
00572 
00580                 bool acquire(atomicx_time nTimeout = 0);
00581 
00585                 void release();
00586 
00592                 size_t GetCount();
00593 
00599                 size_t GetWaitCount();
00600 
00606                 size_t GetMaxAcquired();
00607 
00613                 static size_t GetMax ();
00614 
00615             private:
00616                 size_t m_counter=0;
00617                 size_t m_maxShared;
00618         };
00619 
00620         class smartSemaphore
00621         {
00622             public:
00628                 smartSemaphore (atomicx::semaphore& sem);
00629                 smartSemaphore () = delete;
00633                 ~smartSemaphore();
00634 
00642                 bool acquire(atomicx_time nTimeout = 0);
00643 
00647                 void release();
00648 
00654                 size_t GetCount();
00655 
00661                 size_t GetWaitCount();
00662 
00668                 size_t GetMaxAcquired();
00669 
00675                 static size_t GetMax ();
00676 
00682                 bool IsAcquired ();
00683 
00684             private:
00685             semaphore& m_sem;
00686             bool bAcquired = false;
00687         };
00688 
```

```
00695         /* The stamart mutex implementation */
00696         class mutex
00697         {
00698         public:
00707             void Lock();
00708 
00712             void Unlock();
00713 
00721             void SharedLock();
00722 
00726             void SharedUnlock();
00727 
00733             size_t IsShared();
00734 
00740             bool IsLocked();
00741 
00742         protected:
00743         private:
00744             size_t nSharedLockCount=0;
00745             bool bExclusiveLock=false;
00746         };
00747 
00752         class smartMutex
00753         {
00754             public:
00755                 smartMutex() = delete;
00756 
00762                 smartMutex (mutex& lockObj);
00763 
00767                 ~smartMutex();
00768 
00774                 bool SharedLock();
00775 
00781                 bool Lock();
00782 
00788                 size_t IsShared();
00789 
00795                 bool IsLocked();
00796 
00797             private:
00798 
00799             mutex& m_lock;
00800             uint8_t m_lockType = '\0';
00801         };
00802 
00810         virtual ~atomicx(void);
00811 
00817         static atomicx* GetCurrent();
00818 
00824         static bool Start(void);
00825 
00831         size_t GetID(void);
00832 
00838         size_t GetStackSize(void);
00839 
00846         atomicx_time GetNice(void);
00847 
00853         size_t GetUsedStackSize(void);
00854 
00860         atomicx_time GetCurrentTick(void);
00861 
00870         virtual const char* GetName(void);
00871 
00877         atomicx_time GetTargetTime(void);
00878 
00884         int GetStatus(void);
00885 
00891         int GetSubStatus(void);
00892 
00898         size_t GetReferenceLock(void);
00899 
00905         size_t GetTagLock(void);
00906 
00912         void SetNice (atomicx_time nice);
00913 
00920         template<typename T, size_t N> atomicx(T (&stack)[N]) : m_context{}, m_stackSize{N},
      m_stack((volatile uint8_t*) stack)
00921         {
00922            SetDefaultParameters();
00923 
00924             AddThisThread();
00925         }
00926 
00933         atomicx(size_t nStackSize=0, int nStackIncreasePace=1);
00934 
00940         virtual void run(void) noexcept = 0;
00941 
```

```
00947         virtual void StackOverflowHandler(void) noexcept = 0;
00948
00954         virtual void finish() noexcept
00955         {
00956             return;
00957         }
00958
00962         bool IsStackSelfManaged(void);
00963
00971         bool Yield(atomicx_time nSleep=ATOMICX_TIME_MAX);
00972
00978         atomicx_time GetLastUserExecTime();
00979
00983         size_t GetStackIncreasePace(void);
00984
00988         void YieldNow (void);
00989
00995         void SetDynamicNice(bool status);
00996
01002         bool IsDynamicNiceOn();
01003
01007     private:
01008
01013         void SetDefaultParameters ();
01014
01015         template<typename T> void SetWaitParammeters (T& refVar, size_t nTag=0, aSubTypes asubType =
      aSubTypes::wait)
01016         {
01017             m_TopicId = 0;
01018             m_pLockId = (uint8_t*)&refVar;
01019             m_aStatus = aTypes::wait;
01020             m_aSubStatus = asubType;
01021
01022             m_lockMessage.tag = nTag;
01023             m_lockMessage.message = 0;
01024         }
01025
01038         template<typename T> size_t SafeNotifier(size_t& nMessage, T& refVar, size_t nTag, aSubTypes
      subType, NotifyType notifyAll=NotifyType::one)
01039         {
01040             size_t nRet = 0;
01041
01042             for (auto& thr : *this)
01043             {
01044                 if (thr.m_aSubStatus == subType && thr.m_aStatus == aTypes::wait && thr.m_pLockId ==
      (void*) &refVar && nTag == thr.m_lockMessage.tag)
01045                 {
01046                     thr.m_TopicId = 0;
01047                     thr.m_aStatus = aTypes::now;
01048                     thr.m_nTargetTime = 0;
01049                     thr.m_pLockId = nullptr;
01050
01051                     thr.m_lockMessage.message = nMessage;
01052                     thr.m_lockMessage.tag = nTag;
01053
01054                     nRet++;
01055
01056                     if (notifyAll == NotifyType::one)
01057                     {
01058                         break;
01059                     }
01060                 }
01061             }
01062
01063             return nRet;
01064         }
01065
01075         template<typename T> size_t SafeNotifyLookWaitings(T& refVar, size_t nTag)
01076         {
01077             size_t message=0;
01078
01079             return SafeNotifier(message, refVar, nTag, aSubTypes::look, NotifyType::all);
01080         }
01081
01085     protected:
01086
01087         struct Message
01088         {
01089             size_t tag;
01090             size_t message;
01091         };
01092
01101         uint32_t GetTopicID (const char* pszTopic, size_t nKeyLenght);
01102
01120         template<typename T> bool LookForWaitings(T& refVar, size_t nTag, size_t hasAtleast,
      atomicx_time waitFor)
01121         {
```

```
01122             Timeout timeout (waitFor);
01123
01124             while ((waitFor == 0 || timeout.IsTimedout () == false) && IsWaiting(refVar, nTag,
      hasAtleast) == false)
01125             {
01126                 SetWaitParammeters (refVar, nTag, aSubTypes::look);
01127
01128                 Yield(waitFor);
01129
01130                 m_lockMessage = {0,0};
01131
01132                 if (m_aSubStatus == aSubTypes::timeout)
01133                 {
01134                     return false;
01135                 }
01136
01137                 // Decrease the timeout time to slice the remaining time otherwise break it
01138                 if (waitFor  == 0 || (waitFor = timeout.GetRemaining ()) == 0)
01139                 {
01140                     break;
01141                 }
01142             }
01143
01144             return (timeout.IsTimedout ()) ? false : true;
01145         }
01146
01157         template<typename T> bool LookForWaitings(T& refVar, size_t nTag, atomicx_time waitFor)
01158         {
01159             if (IsWaiting(refVar, nTag) == false)
01160             {
01161                 SetWaitParammeters (refVar, nTag, aSubTypes::look);
01162
01163                 Yield(waitFor);
01164
01165                 m_lockMessage = {0,0};
01166
01167                 if (m_aSubStatus == aSubTypes::timeout)
01168                 {
01169                     return false;
01170                 }
01171             }
01172
01173             return true;
01174         }
01175
01189         template<typename T> bool IsWaiting(T& refVar, size_t nTag=0, size_t hasAtleast = 1, aSubTypes
      asubType = aSubTypes::wait)
01190         {
01191             hasAtleast = hasAtleast == 0 ? 1 : hasAtleast;
01192
01193             for (auto& thr : *this)
01194             {
01195                 if (thr.m_aSubStatus == asubType && thr.m_aStatus == aTypes::wait && thr.m_pLockId ==
      (void*) &refVar && (thr.m_lockMessage.tag == nTag))
01196                 {
01197                     if ((--hasAtleast) == 0)
01198                     {
01199                         return true;
01200                     }
01201                 }
01202             }
01203
01204             return false;
01205         }
01206
01220         template<typename T> size_t HasWaitings(T& refVar, size_t nTag=0, aSubTypes asubType =
      aSubTypes::wait)
01221         {
01222             size_t nCounter = 0;
01223
01224             for (auto& thr : *this)
01225             {
01226                 if (thr.m_aSubStatus == asubType && thr.m_aStatus == aTypes::wait && thr.m_aStatus ==
      aTypes::wait && thr.m_pLockId == (void*) &refVar && (thr.m_lockMessage.tag == nTag))
01227                 {
01228                     nCounter++;
01229                 }
01230             }
01231
01232             return nCounter;
01233         }
01234
01247         template<typename T> bool Wait(size_t& nMessage, T& refVar, size_t nTag=0, atomicx_time
      waitFor=0, aSubTypes asubType = aSubTypes::wait)
01248         {
01249             SafeNotifyLookWaitings(refVar, nTag);
01250
```

```
01251             SetWaitParammeters (refVar, nTag, asubType);
01252
01253             m_lockMessage.tag = nTag;
01254
01255             Yield(waitFor);
01256
01257             bool bRet = false;
01258
01259             if (m_aSubStatus != aSubTypes::timeout)
01260             {
01261                 nMessage = m_lockMessage.message;
01262                 bRet = true;
01263             }
01264
01265             m_lockMessage = {0,0};
01266
01267             m_aSubStatus = aSubTypes::ok;
01268
01269             return bRet;
01270         }
01271
01284         template<typename T> bool Wait(T& refVar, size_t nTag=0, atomicx_time waitFor=0, aSubTypes
      asubType = aSubTypes::wait)
01285         {
01286             SafeNotifyLookWaitings(refVar, nTag);
01287
01288             SetWaitParammeters (refVar, nTag, asubType);
01289
01290             m_lockMessage.tag = nTag;
01291
01292             Yield(waitFor);
01293
01294             bool bRet = false;
01295
01296             if (m_aSubStatus != aSubTypes::timeout)
01297             {
01298                 bRet = true;
01299             }
01300
01301             m_lockMessage = {0,0};
01302             m_aSubStatus = aSubTypes::ok;
01303
01304             return bRet;
01305         }
01306
01321         template<typename T> size_t SafeNotify(size_t& nMessage, T& refVar, size_t nTag=0,  NotifyType
      notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01322         {
01323             return SafeNotifier(nMessage, refVar, nTag, asubType, notifyAll);
01324         }
01325
01340         template<typename T> size_t Notify(size_t& nMessage, T& refVar, size_t nTag=0, NotifyType
      notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01341         {
01342             size_t bRet = SafeNotify (nMessage, refVar, nTag, notifyAll, asubType);
01343
01344             if (bRet) Yield(0);
01345
01346             return bRet;
01347         }
01348
01349         template<typename T> size_t Notify(size_t&& nMessage, T& refVar, size_t nTag=0, NotifyType
      notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01350         {
01351             size_t bRet = SafeNotify (nMessage, refVar, nTag, notifyAll, asubType);
01352
01353             if (bRet) Yield(0);
01354
01355             return bRet;
01356         }
01357
01373         template<typename T> size_t SyncNotify(size_t& nMessage, T& refVar, size_t nTag=0,
      atomicx_time waitForWaitings=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType =
      aSubTypes::wait)
01374         {
01375             if (LookForWaitings (refVar, nTag, waitForWaitings) == false)
01376             {
01377                 return 0;
01378             }
01379
01380             size_t bRet = SafeNotify (nMessage, refVar, nTag, notifyAll, asubType);
01381
01382             if (bRet) Yield(0);
01383
01384             return bRet;
01385         }
01386
```

```
01387         template<typename T> size_t SyncNotify(size_t&& nMessage, T& refVar, size_t nTag=0,
      atomicx_time waitForWaitings=0, NotifyType notifyAll=NotifyType::one, aSubTypes asubType =
      aSubTypes::wait)
01388         {
01389             if (LookForWaitings (refVar, nTag, waitForWaitings) == false)
01390             {
01391                 return 0;
01392             }
01393 
01394             size_t bRet = SafeNotify (nMessage, refVar, nTag, notifyAll, asubType);
01395 
01396             if (bRet) Yield(0);
01397 
01398             return bRet;
01399         }
01400 
01414         template<typename T> size_t SafeNotify(T& refVar, size_t nTag=0, NotifyType
      notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01415         {
01416              size_t message=0;
01417              return SafeNotifier (message, refVar, nTag, asubType, notifyAll);
01418         }
01419 
01434         template<typename T> size_t SyncNotify(T& refVar, size_t nTag, atomicx_time waitForWaitings=0,
      NotifyType notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01435         {
01436             if (LookForWaitings (refVar, nTag, waitForWaitings) == false)
01437             {
01438                 return 0;
01439             }
01440 
01441             size_t bRet = SafeNotify(refVar, nTag, notifyAll, asubType);
01442 
01443             if (bRet) Yield(0);
01444 
01445             return bRet;
01446         }
01447 
01461         template<typename T> size_t Notify(T& refVar, size_t nTag=0, NotifyType
      notifyAll=NotifyType::one, aSubTypes asubType = aSubTypes::wait)
01462         {
01463             size_t bRet = SafeNotify(refVar, nTag, notifyAll, asubType);
01464 
01465             if (bRet) Yield(0);
01466 
01467             return bRet;
01468         }
01469 
01485         bool WaitBrokerMessage (const char* pszKey, size_t nKeyLenght, Message& message);
01486 
01495         bool WaitBrokerMessage (const char* pszKey, size_t nKeyLenght);
01496 
01506         bool Publish (const char* pszKey, size_t nKeyLenght, const Message message);
01507 
01519         bool SafePublish (const char* pszKey, size_t nKeyLenght, const Message message);
01520 
01529         bool Publish (const char* pszKey, size_t nKeyLenght);
01530 
01541         bool SafePublish (const char* pszKey, size_t nKeyLenght);
01542 
01551         bool HasSubscriptions (const char* pszTopic, size_t nKeyLenght);
01552 
01560         bool HasSubscriptions (uint32_t nKeyID);
01561 
01574         virtual bool BrokerHandler(const char* pszKey, size_t nKeyLenght, Message& message)
01575         {
01576             (void) pszKey; (void) nKeyLenght; (void) message;
01577             return false;
01578         }
01579 
01588         virtual bool IsSubscribed (const char* pszKey, size_t nKeyLenght)
01589         {
01590             (void) pszKey; (void) nKeyLenght;
01591 
01592             return false;
01593         }
01594 
01600         void SetStackIncreasePace(size_t nIncreasePace);
01601 
01602     private:
01603 
01607         void AddThisThread();
01608 
01612         void RemoveThisThread();
01613 
01623         uint16_t crc16(const uint8_t* pData, size_t nSize, uint16_t nCRC);
01624 
```

```
01631         static bool SelectNextThread(void);
01632
01637         atomicx* m_paNext = nullptr;
01638         atomicx* m_paPrev = nullptr;
01639
01640         jmp_buf m_context;
01641
01642         size_t m_stackSize=0;
01643         size_t m_stacUsedkSize=0;
01644         size_t m_stackIncreasePace=1;
01645
01646         Message m_lockMessage = {0,0};
01647
01648         atomicx_time m_nTargetTime=0;
01649         atomicx_time m_nice=0;
01650         atomicx_time m_LastUserExecTime=0;
01651         atomicx_time m_lastResumeUserTime=0;
01652
01653         uint32_t m_TopicId=0;
01654
01655         aTypes  m_aStatus = aTypes::start;
01656         aSubTypes m_aSubStatus = aSubTypes::ok;
01657
01658         volatile uint8_t* m_stack;
01659         volatile uint8_t* m_pStaskStart=nullptr;
01660         volatile uint8_t* m_pStaskEnd=nullptr;
01661
01662         uint8_t* m_pLockId=nullptr;
01663
01664         struct
01665         {
01666             bool autoStack : 1;
01667             bool dynamicNice : 1;
01668         } m_flags = {};
01669     };
01670 }
01671
01672 #endif /* atomicx_hpp */
```

Index