Embedded Multicore Building Blocks V1.0.0

Generated by Doxygen 1.8.11

Contents

Chapter 1

Overview

The Embedded Multicore Building Blocks (EMB^2) are an easy to use yet powerful and efficient C/C++ library for the development of parallel applications. EMB^2 has been specifically designed for embedded systems and the typical requirements that accompany them, such as real-time capability and constraints on memory consumption. As a major advantage, low-level operations are hidden in the library which relieves software developers from the burden of thread management and synchronization. This not only improves productivity of parallel software development, but also results in increased reliability and performance of the applications.

EMB^2 is independent of the hardware architecture (x86, ARM, ...) and runs on various platforms, from small devices to large systems containing numerous processor cores. It builds on MTAPI, a standardized programming interface for leveraging task parallelism in embedded systems containing symmetric or asymmetric multicore processors. A core feature of MTAPI is low-overhead scheduling of fine-grained tasks among the available cores during runtime. Unlike existing libraries, EMB^2 supports task priorities and affinities, which allows the creation of soft real-time systems. Additionally, the scheduling strategy can be optimized for non-functional requirements such as minimal latency and fairness.

Besides the task scheduler, EMB^2 provides basic parallel algorithms, concurrent data structures, and skeletons for implementing stream processing applications (see figure below). These building blocks are largely implemented in a non-blocking fashion, thus preventing frequently encountered pitfalls like lock contention, deadlocks, and priority inversion. As another advantage in real-time systems, the algorithms and data structures give certain progress guarantees. For example, wait-free data structures guarantee system-wide progress which means that every operation completes within a finite number of steps independently of any other concurrent operations on the same data structure.

Chapter 2

Module Index

2.1 API

Here is a list of all modules:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

Module Documentation

5.1 Containers

Concurrent data structures, mainly containers.

Modules

- Stacks

  Concurrent stacks.
- Pools

  Concurrent pools.
- Queues

  Concurrent queues.

5.1.1 Detailed Description

Concurrent data structures, mainly containers.

5.2 Stack Concept

Concept for thread-safe stacks.

Classes

- class embb::containers::LockFreeStack< Type, ValuePool >

  Lock-free stack.

5.2.1 Detailed Description

Concept for thread-safe stacks.

Description

A stack is an abstract data type holding a collection of elements of some predetermined type. A stack provides two operations: `TryPush` and `TryPop`. `TryPush` tries to add an element to the collection, and `TryPop` tries to remove an element from the collection. A stack has LIFO (Last-In, First-out) semantics, i.e., the last element added to the collection (`TryPush`) is removed first (`TryPop`). The capacity `cap` of a stack defines the number of elements it can store (depending on the implementation, a stack might store more than `cap` elements, since for thread-safe memory management, more memory than necessary for holding `cap` elements has to be provided).

Requirements

- Let `Stack` be the stack class
- Let `Type` be the element type of the stack
- Let `capacity` be a value of type `size_t`
- Let `element` be a reference to an element of type `Type`

Expression	Return type	Description
`Stack<Type>(capacity)`	Nothing	Constructs a stack with capacity `capacity` that holds elements of type `Type`.
`TryPush(element)`	`bool`	Tries to push `element` onto the stack. Returns `false` if the stack is full, otherwise `true`.
`TryPop(element)`	`bool`	Tries to pop an element from the stack. Returns `false` if the stack is empty, otherwise `true`. In the latter case, the popped element is stored in `element` which must be passed by reference.

5.3 Stacks

Concurrent stacks.

Classes

- class embb::containers::LockFreeStack< Type, ValuePool >

  Lock-free stack.

5.3.1 Detailed Description

Concurrent stacks.

See also

Stack Concept

5.4 Pools

Concurrent pools.

Classes

- class embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >
  Lock-free value pool using binary tree construction.
- class embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >
  Pool for thread-safe management of arbitrary objects.
- class embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >
  Wait-free value pool using array construction.

5.4.1 Detailed Description

Concurrent pools.

5.5 Value Pool Concept

Concept for thread-safe value pools.

Classes

- class embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >

  Lock-free value pool using binary tree construction.
- class embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >

  Wait-free value pool using array construction.

5.5.1 Detailed Description

Concept for thread-safe value pools.

Description

A value pool is a multi-set of elements, where each element has a unique, continuous (starting with 0) index. The elements cannot be modified and are given at construction time by providing first/last iterators.

A value pool provides two primary operations: `Allocate` and `Free`. `Allocate` allocates an element/index "pair" (index via return, element via reference parameter) from the pool, and `Free` returns an element/index pair to the pool. To guarantee linearizability, `element` is not allowed to be modified between `Allocate` and `Free`. It is only allowed to free elements that have previously been allocated. The `Allocate` function does not guarantee an order on which indices are allocated. The count of elements that can be allocated with `Allocate` might be smaller than the count of elements, the pool is initialized with. This might be because of implementation details and respective concurrency effects: for example, if indices are managed within a queue, one has to protect queue elements from concurrency effects (reuse and access). As long as a thread potentially accesses a node (and with that an index), the respective index cannot not be given out to the user, even if being logically not part of the pool anymore. However, the user might want to guarantee a certain amount of indices to the user. Therefore, the static `GetMinimumElementCountForGuaranteedCapacity` method is used. The user passes the count of indices to this method that shall be guaranteed by the pool. The method returns the count on indices, the pool has to be initialized with in order to guarantee this count on indices.

Requirements

- Let `Pool` be the pool class
- Let `Type` be the element type of the pool. Atomic operations must be possible on `Type`.
- Let `b`, d be objects of type `Type`
- Let `i`, j be forward iterators supporting `std::distance`.
- Let `c` be an object of type `Type&`
- Let `e` be a value of type `int`
- Let `f` be a value of type `int`

Valid Expressions

Expression	Return type	Description
`Pool<Type, b>(i, j)`	Nothing	Constructs a value pool holding elements of type `Type`, where `b` is the bottom element. The bottom element cannot be stored in the pool, it is exclusively used to mark empty cells. The pool initially contains `std::distance(i, j)` elements which are copied during construction from the range [i, j]. A concrete class satisfying the value pool concept might provide additional template parameters for specifying allocators.
`Allocate(c)`	`int`	Allocates an element/index "pair" from the pool. Returns -1, if no element is available, i.e., the pool is empty. Otherwise, returns the index of the element in the pool. The value of the pool element is written into parameter reference `c`.
`Free(d, e)`	`void`	Returns an element `d` to the pool, where `e` is its index. The values of `d` and `e` have to match the values of the previous call to `Allocate`. For each allocated element, `Free` must be called exactly once.
`GetMinimumElementCountForGuaranteedCapacity(f)`	`void`	Static method, returns the count of indices, the user has to initialize the pool with in order to guarantee a count of `f` elements (irrespective of concurrency effects).

5.6 Queue Concept

Concept for thread-safe queues.

Classes

- class embb::containers::LockFreeMPMCQueue< Type, ValuePool >

  Lock-free queue for multiple producers and multiple consumers.
- class embb::containers::WaitFreeSPSCQueue< Type, Allocator >

  Wait-free queue for a single producer and a single consumer.

5.6.1 Detailed Description

Concept for thread-safe queues.

Description

A queue is an abstract data type holding a collection of elements of some predetermined type. A queue provides two operations: `TryEnqueue` and `TryDequeue`. `TryEnqueue` tries to add an element to the collection, and `TryDequeue` tries to remove an element from the collection. A queue has per-thread FIFO (First-In, First-out) semantics, i.e., if one thread enqueues two elements and another thread dequeues these elements, then they appear in the same order. The capacity `cap` of a queue defines the number of elements it can store (depending on the implementation, a queue might store more than `cap` elements, since for thread-safe memory management, more memory than necessary for holding `cap` elements has to be provided).

Requirements

- Let `Queue` be the queue class
- Let `Type` be the element type of the queue
- Let `capacity` be a value of type `size_t`
- Let `element` be a reference to an element of type `Type`

Expression	Return type	Description
`Queue<Type>(capacity)`	Nothing	Constructs a queue with minimal capacity `capacity` that holds elements of type `T`.
`TryEnqueue(element)`	`bool`	Tries to enqueue `element` into the queue. Returns `false` if the queue is full, otherwise `true`.
`TryDequeue(element)`	`bool`	Tries to dequeue an element from the queue. Returns `false` if the queue is empty, otherwise `true`. In the latter case, the dequeued element is stored in `element` which must be passed by reference.

5.7 Queues

Concurrent queues.

Classes

- class embb::containers::LockFreeMPMCQueue< Type, ValuePool >
  Lock-free queue for multiple producers and multiple consumers.
- class embb::containers::WaitFreeSPSCQueue< Type, Allocator >
  Wait-free queue for a single producer and a single consumer.

5.7.1 Detailed Description

Concurrent queues.

See also

Queue Concept

5.8 Dataflow

C++ library for parallel, stream-based applications.

Classes

- class embb::dataflow::Network

  Represents a set of processes that are connected by communication channels.

5.8.1 Detailed Description

C++ library for parallel, stream-based applications.

5.9 Algorithms

High-level parallel algorithms and functionalities.

Modules

- Counting
  Parallel count operation.
- Foreach
  Parallel foreach loop.
- Invoke
  Parallel invocation of functions.
- Sorting
  Parallel merge sort and quick sort algorithms.
- Reduction
  Parallel reduction computation.
- Scan
  Parallel scan computation.
- Zip Iterator
  Zip two iterators.

Classes

- struct embb::algorithms::Identity
  Unary identity functor.

5.9.1 Detailed Description

High-level parallel algorithms and functionalities.

5.10 Counting

Parallel count operation.

Functions

- template<typename RAI , typename ValueType >
  std::iterator_traits< RAI >::difference_type embb::algorithms::Count (RAI first, RAI last, const ValueType &value, const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Counts in parallel the number of elements in a range that are equal to the specified value.
- template<typename RAI , typename ComparisonFunction >
  std::iterator_traits< RAI >::difference_type embb::algorithms::CountIf (RAI first, ComparisonFunction comparison, const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Counts in parallel the number of elements in a range for which the comparison function returns `true`*.*

5.10.1 Detailed Description

Parallel count operation.

5.10.2 Function Documentation

5.10.2.1 template<typename RAI , typename ValueType > std::iterator_traits<RAI>::difference_type embb::algorithms::Count (RAI *first*, RAI *last*, const ValueType & *value*, const **embb::mtapi::ExecutionPolicy** & *policy* = **embb::mtapi::ExecutionPolicy** `()` , size_t *block_size* = `0`)

Counts in parallel the number of elements in a range that are equal to the specified value.

The range consists of the elements from `first` to `last`, excluding the last element.

Returns

The number of elements that are equal to `value`

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the execution order of the comparison operations.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

CountIf(), embb::mtapi::ExecutionPolicy

Template Parameters

RAI	Random access iterator
ValueType	Type of `value` that is compared to the elements in the range using the `operator==`.

Parameters

`in`	*first*	Random access iterator pointing to the first element of the range
`in`	*last*	Random access iterator pointing to the last plus one element of the range
`in`	*value*	Value that the elements in the range are compared to using `operator==`
`in`	*policy*	embb::mtapi::ExecutionPolicy for the counting algorithm
`in`	*block_size*	Lower bound for partitioning the range of elements into blocks that are sorted in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.10.2.2 template<typename RAI , typename ComparisonFunction > std::iterator_traits<RAI>::difference_type embb::algorithms::CountIf (RAI *first*, ComparisonFunction *comparison*, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy `()`, size_t *block_size* = `0`)

Counts in parallel the number of elements in a range for which the comparison function returns `true`.

The range consists of the elements from `first` to `last`, excluding the last element.

Returns

The number of elements for which `comparison` returns true

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the execution order of the comparison function.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

Count(), embb::mtapi::ExecutionPolicy

Template Parameters

RAI	Random access iterator
ComparisonFunction	Unary predicate with argument of type `std::iterator_traits<RAI>::value_type` or an embb::mtapi::Job associated with an action function accepting a struct containing one member of type `std::iterator_traits<RAI>::value_type` as its argument buffer and a struct containing one bool member as its result buffer.

Parameters

`in`	*first*	Random access iterator pointing to the first element of the range RAI last, [IN] Random access iterator pointing to the last plus one element of the range
`in`	*comparison*	Unary predicate used to test the elements in the range. Elements for which `comparison` returns true are counted.
`in`	*policy*	embb::mtapi::ExecutionPolicy for the counting algorithm
`in`	*block_size*	Lower bound for partitioning the range of elements into blocks that are sorted in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.11 Foreach

Parallel foreach loop.

Functions

- template<typename RAI , typename Function >
  void embb::algorithms::ForEach (RAI first, RAI last, Function unary, const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Applies a unary function to the elements of a range in parallel.
- template<typename Integer , typename Diff , typename Function >
  void embb::algorithms::ForLoop (Integer first, Integer last, Diff stride=1, Function unary, const embb::mtapi↩ ::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Applies a unary function to the integers of a range in parallel.

5.11.1 Detailed Description

Parallel foreach loop.

5.11.2 Function Documentation

5.11.2.1 template<typename RAI , typename Function > void embb::algorithms::ForEach (RAI *first*, RAI *last*, Function *unary*, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy ( ) , size_t *block_size* = 0)

Applies a unary function to the elements of a range in parallel.

The range consists of the elements from `first` to `last`, excluding the last element.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the order in which the function is applied to the elements.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, ZipIterator

Template Parameters

RAI	Random access iterator
Function	Unary function with argument of type `std::iterator_traits<RAI>::value_type` or an embb::mtapi::Job associated with an action function accepting a struct containing one member of type `std::iterator_traits<RAI>::value_type` as its argument buffer and a struct containing one member of type `std::iterator_traits<RAI>::value_type` as its result buffer.

Parameters

`in`	*first*	Random access iterator pointing to the first element of the range
`in`	*last*	Random access iterator pointing to the last plus one element of the range
`in`	*unary*	Unary function applied to each element in the range
`in`	*policy*	embb::mtapi::ExecutionPolicy for the loop execution
`in`	*block_size*	Lower bound for partitioning the range of elements into blocks that are treated in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.11.2.2 template<typename Integer , typename Diff , typename Function > void embb::algorithms::ForLoop (Integer *first*, Integer *last*, Diff *stride* = `1`, Function *unary*, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy `()`, size_t *block_size* = `0`)

Applies a unary function to the integers of a range in parallel.

The range consists of the integers from `first` to `last`, excluding the last element, strided by `stride`.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe

Note

No guarantee is given on the order in which the function is applied to the integers.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy

Template Parameters

Integer	integer type
Function	Unary function with argument of type `std::iterator_traits<RAI>::value_type` or an embb::mtapi::Job associated with an action function accepting a struct containing one member of type `std::iterator_traits<RAI>::value_type` as its argument buffer and a struct containing one member of type `std::iterator_traits<RAI>::value_type` as its result buffer.

Parameters

`in`	*first*	First integer of the range
`in`	*last*	Last plus one integer of the range
`in`	*stride*	Stride between integers, can be omitted
`in`	*unary*	Unary function applied to each element in the range
`in`	*policy*	embb::mtapi::ExecutionPolicy for the loop execution
`in`	*block_size*	Lower bound for partitioning the range of integers into blocks that are treated in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of integers in the range divided by the number of available cores.

5.12 Invoke

Parallel invocation of functions.

Typedefs

- typedef embb::base::Function< void > embb::algorithms::InvokeFunctionType

    Function type used by Invoke.

Functions

- template<typename Function1 , typename Function2 , ... >
  void embb::algorithms::Invoke (Function1 func1, Function2 func2,...)

    Spawns two to ten function objects or embb::mtapi::Job at once and runs them in parallel.

- template<typename Function1 , typename Function2 , ... >
  void embb::algorithms::Invoke (Function1 func1, Function2 func2,..., const embb::mtapi::ExecutionPolicy &policy)

    Spawns two to ten function objects or embb::mtapi::Job at once and runs them in parallel using the given embb::mtapi::ExecutionPolicy.

5.12.1 Detailed Description

Parallel invocation of functions.

5.12.2 Typedef Documentation

5.12.2.1 typedef **embb::base::Function**<void> **embb::algorithms::InvokeFunctionType**

Function type used by Invoke.

5.12.3 Function Documentation

5.12.3.1 template<typename Function1 , typename Function2 , ... > void embb::algorithms::Invoke (Function1 *func1*, Function2 *func2*, ...)

Spawns two to ten function objects or embb::mtapi::Job at once and runs them in parallel.

Blocks until all of them are done.

Parameters

in	*func1*	First function object to invoke
in	*func2*	Second function object to invoke

5.12.3.2 template<typename Function1 , typename Function2 , ... > void embb::algorithms::Invoke (Function1 *func1*, Function2 *func2*, ..., const embb::mtapi::ExecutionPolicy & *policy*)

Spawns two to ten function objects or embb::mtapi::Job at once and runs them in parallel using the given embb::mtapi::ExecutionPolicy.

Blocks until all of them are done.

Parameters

in	*func1*	Function object to invoke
in	*func2*	Second function object to invoke
in	*policy*	embb::mtapi::ExecutionPolicy to use

5.13 Sorting

Parallel merge sort and quick sort algorithms.

Functions

- template<typename RAI , typename ComparisonFunction >
  void embb::algorithms::MergeSortAllocate (RAI first, RAI last, ComparisonFunction comparison=std::less< typename std::iterator_traits< RAI >::value_type >(), const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Sorts a range of elements using a parallel merge sort algorithm with implicit allocation of dynamic memory.
- template<typename RAI , typename RAITemp , typename ComparisonFunction >
  void embb::algorithms::MergeSort (RAI first, RAI last, RAITemp temporary_first, ComparisonFunction comparison=std::less< typename std::iterator_traits< RAI >::value_type >(), const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Sorts a range of elements using a parallel merge sort algorithm without implicit allocation of dynamic memory.
- template<typename RAI , typename ComparisonFunction >
  void embb::algorithms::QuickSort (RAI first, RAI last, ComparisonFunction comparison=std::less< typename std::iterator_traits< RAI >::value_type >(), const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Sorts a range of elements using a parallel quick sort algorithm.

5.13.1 Detailed Description

Parallel merge sort and quick sort algorithms.

5.13.2 Function Documentation

5.13.2.1 template<typename RAI , typename ComparisonFunction > void embb::algorithms::MergeSortAllocate (RAI *first*, RAI *last*, ComparisonFunction *comparison* = `std::less< typename std::iterator_traits< RAI >::value_type >()`, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy `()`, size_t *block_size* = `0`)

Sorts a range of elements using a parallel merge sort algorithm with implicit allocation of dynamic memory.

The range consists of the elements from `first` to `last`, excluding the last element. Since the algorithm does not sort in-place, it requires additional memory which is implicitly allocated by the function.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Dynamic memory allocation

Array with `last-first` elements of type `std::iterator_traits<RAI>::value_type`.

Concurrency

Thread-safe if the elements in the range `[first,last)` are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the execution order of the comparison operations.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, MergeSort()

Template Parameters

RAI	Random access iterator
ComparisonFunction	Binary predicate with both arguments of type `std::iterator_traits<RAI>::value_type` or an embb::mtapi::Job associated with an action function accepting a struct containing two members of type `std::iterator_traits<RAI>::value_type` as its argument buffer and a struct containing one bool member as its result buffer.

Parameters

in	*first*	Random access iterator pointing to the first element of the range
in	*last*	Random access iterator pointing to the last plus one element of the range
in	*comparison*	Binary predicate used to establish the sorting order. An element `a` appears before an element `b` in the sorted range if `comparison(a, b) == true`. The default value uses the less-than relation.
in	*policy*	embb::mtapi::ExecutionPolicy for the merge sort algorithm
in	*block_size*	Lower bound for partitioning the range of elements into blocks that are sorted in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.13.2.2 template<typename RAI , typename RAITemp , typename ComparisonFunction > void embb::algorithms::MergeSort (RAI *first*, RAI *last*, RAITemp *temporary_first*, ComparisonFunction *comparison* = `std::less< typename std::iterator_traits< RAI >::value_type >()`, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy `()`, size_t *block_size* = `0`)

Sorts a range of elements using a parallel merge sort algorithm without implicit allocation of dynamic memory.

The range consists of the elements from `first` to `last`, excluding the last element. Since the algorithm does not sort in-place, it requires additional memory which must be provided by the user. The range pointed to by `temporary_first` must have the same number of elements as the range to be sorted, and the elements of both ranges must have the same type.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the ranges `[first,last)` and `[temporary_first,temporary_first+(last-first)` are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the execution order of the comparison operations.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, MergeSortAllocate()

Template Parameters

RAI	Random access iterator
RAITemp	Random access iterator for temporary memory. Has to have the same value type as RAI.
ComparisonFunction	Binary predicate with both arguments of type `std::iterator_traits<RAI>::value_type`.

Parameters

in	*first*	Random access iterator pointing to the first element of the range
in	*last*	Random access iterator to last plus one element to be sorted
in	*temporary_first*	Random access iterator pointing to the last plus one element of the range
in	*comparison*	Binary predicate used to establish the sorting order. An element `a` appears before an element `b` in the sorted range if `comparison(a, b) == true`. The default value uses the less-than relation.
in	*policy*	embb::mtapi::ExecutionPolicy for the merge sort algorithm
in	*block_size*	Lower bound for partitioning the range of elements into blocks that are sorted in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.13.2.3 template<typename RAI , typename ComparisonFunction > void embb::algorithms::QuickSort (RAI *first*, RAI *last*, ComparisonFunction *comparison* = `std::less< typename std::iterator_traits< RAI >::value_type >()`, const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy `()`, size_t *block_size* = `0`)

Sorts a range of elements using a parallel quick sort algorithm.

The range consists of the elements from `first` to `last`, excluding the last element. The algorithm sorts in-place and requires no additional memory. It has, however, a worst-case time complexity of `O((last-first)`^{2}`)`.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range `[first,last)` are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the execution order of the comparison operations.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, MergeSort()

Template Parameters

RAI	Random access iterator
ComparisonFunction	Binary predicate with both arguments of type `std::iterator_traits<RAI>::value_type` or an embb::mtapi::Job associated with an action function accepting a struct containing two members of type `std::iterator_traits<RAI>::value_type` as its argument buffer and a struct containing one bool member as its result buffer.

Parameters

in	*first*	Random access iterator pointing to the first element of the range
in	*last*	Random access iterator pointing to the last plus one element of the range
in	*comparison*	Binary predicate used to establish the sorting order. An element `a` appears before an element `b` in the sorted range if `comparison(a, b) == true`. The default value uses the less-than relation.
in	*policy*	embb::mtapi::ExecutionPolicy for the quick sort algorithm
in	*block_size*	Lower bound for partitioning the range of elements into blocks that are sorted in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores. Note that quick sort does not guarantee a partitioning into evenly sized blocks, as the partitions depend on the values to be sorted.

5.14 Reduction

Parallel reduction computation.

Functions

- template<typename RAI , typename ReturnType , typename ReductionFunction , typename TransformationFunction >
  ReturnType embb::algorithms::Reduce (RAI first, RAI last, ReturnType neutral, ReductionFunction reduction, TransformationFunction transformation=Identity(), const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Performs a parallel reduction operation on a range of elements.

5.14.1 Detailed Description

Parallel reduction computation.

5.14.2 Function Documentation

5.14.2.1 template<typename RAI , typename ReturnType , typename ReductionFunction , typename TransformationFunction > ReturnType embb::algorithms::Reduce (RAI *first,* RAI *last,* ReturnType *neutral,* ReductionFunction *reduction,* TransformationFunction *transformation* = Identity (), const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy (), size_t *block_size* = 0)

Performs a parallel reduction operation on a range of elements.

The range consists of the elements from `first` to `last`, excluding the last element. The type of the result (`ReturnType`) is deduced from the `neutral` element.

Returns

`reduction(transformation(*first), ..., transformation(*(last-1)))` where the reduction function is applied pairwise.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the order in which the functions `reduction` and `transformation` are applied to the elements.
For all `x` of type `ReturnType` it must hold that `reduction(x, neutral) == x`.
The reduction operation need not be commutative but must be associative, i.e., `reduction(x,`

`reduction(y, z)) == reduction(reduction(x, y), z))` for all `x`, `y`, `z` of type `Return↩ Type`.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, ZipIterator, Identity

Template Parameters

RAI	Random access iterator
ReturnType	Type of result of reduction operation, deduced from `neutral`
ReductionFunction	Binary reduction function object with signature `ReturnType ReductionFunction(ReturnType, ReturnType)` or an embb::mtapi::Job associated with an action function accepting a struct containing two ReturnType members as its argument buffer and a struct containing one ReturnType member as its result buffer.
TransformationFunction	Unary transformation function object with signature `ReturnType TransformationFunction(typename std::iterator_traits<RAI>::value_type)` or an embb::mtapi::Job associated with an action function accepting a struct containing one InputType member as its argument buffer and a struct containing one ReturnType member as its result buffer.

Parameters

`in`	*first*	Random access iterator pointing to the first element of the range
`in`	*last*	Random access iterator pointing to the last plus one element of the range
`in`	*neutral*	Neutral element of the reduction operation.
`in`	*reduction*	Reduction operation to be applied to the elements of the range
`in`	*transformation*	Transforms the elements of the range before the reduction operation is applied
`in`	*policy*	embb::mtapi::ExecutionPolicy for the reduction computation
`in`	*block_size*	Lower bound for partitioning the range of elements into blocks that are treated in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.15 Scan

Parallel scan computation.

Functions

- template<typename RAIIn , typename RAIOut , typename ReturnType , typename ScanFunction , typename TransformationFunction >
void embb::algorithms::Scan (RAIIn first, RAIIn last, RAIOut output_first, ReturnType neutral, ScanFunction scan, TransformationFunction transformation=Identity(), const embb::mtapi::ExecutionPolicy &policy=embb::mtapi::ExecutionPolicy(), size_t block_size=0)

  Performs a parallel scan (or prefix) computation on a range of elements.

5.15.1 Detailed Description

Parallel scan computation.

5.15.2 Function Documentation

5.15.2.1 template<typename RAIIn , typename RAIOut , typename ReturnType , typename ScanFunction , typename TransformationFunction > void embb::algorithms::Scan (RAIIn *first,* RAIIn *last,* RAIOut *output_first,* ReturnType *neutral,* ScanFunction *scan,* TransformationFunction *transformation* = Identity(), const embb::mtapi::ExecutionPolicy & *policy* = embb::mtapi::ExecutionPolicy(), size_t *block_size* = 0)

Performs a parallel scan (or prefix) computation on a range of elements.

The algorithm reads an input range and writes its result to a separate output range. The input range consists of the elements from `first` to `last`, excluding the last element. The output range consists of the elements from `output_first` to `output_first + std::difference(last - first)`.

The algorithm performs two runs on the given range. Hence, a performance speedup can only be expected on processors with more than two cores.

Exceptions

embb::base::ErrorException	if not enough MTAPI tasks can be created to satisfy the requirements of the algorithm.

Concurrency

Thread-safe if the elements in the range are not modified by another thread while the algorithm is executed.

Note

No guarantee is given on the order in which the functions `scan` and `transformation` are applied to the elements.
For all `x` of type `ReturnType` it must hold that `reduction(x, neutral) == x`.
The reduction operation need not be commutative but must be associative, i.e., `reduction(x, reduction(y, z)) == reduction(reduction(x, y), z))` for all `x`, `y`, `z` of type `ReturnType`.
For nested algorithms, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

See also

embb::mtapi::ExecutionPolicy, Identity, ZipIterator

Template Parameters

RAIIn	Random access iterator type of input range
RAIOut	Random access iterator type of output range
ReturnType	Type of output elements of scan operation, deduced from `neutral`
ScanFunction	Binary scan function object with signature `ReturnType ScanFunction(ReturnType, ReturnType)` or an embb::mtapi::Job associated with an action function accepting a struct containing two ReturnType members as its argument buffer and a struct containing one ReturnType member as its result buffer.
TransformationFunction	Unary transformation function object with signature `ReturnType TransformationFunction(typename std::iterator_traits<RAIIn>::value_type)` or an embb::mtapi::Job associated with an action function accepting a struct containing one InputType member as its argument buffer and a struct containing one ReturnType member as its result buffer.

Parameters

`in`	*first*	Random access iterator pointing to the first element of the input range
`in`	*last*	Random access iterator pointing to the last plus one element of the input range
`in`	*output_first*	Random access iterator pointing to the first element of the output range
`in`	*neutral*	Neutral element of the `scan` operation.
`in`	*scan*	Scan operation to be applied to the elements of the input range
`in`	*transformation*	Transforms the elements of the input range before the scan operation is applied
`in`	*policy*	embb::mtapi::ExecutionPolicy for the scan computation
`in`	*block_size*	Lower bound for partitioning the range of elements into blocks that are treated in parallel. Partitioning of a block stops if its size is less than or equal to `block_size`. The default value 0 means that the minimum block size is determined automatically depending on the number of elements in the range divided by the number of available cores.

5.16 Zip Iterator

Zip two iterators.

Classes

- class embb::algorithms::ZipPair< TypeA, TypeB >
  Container for the values of two dereferenced iterators.
- class embb::algorithms::ZipIterator< IteratorA, IteratorB >
  Zip container for two iterators.

Functions

- template<typename IteratorA , typename IteratorB >
  ZipIterator< IteratorA, IteratorB > embb::algorithms::Zip (IteratorA iter_a, IteratorB iter_b)
  Creates a zip iterator from two iterators.

5.16.1 Detailed Description

Zip two iterators.

5.16.2 Function Documentation

5.16.2.1 template<typename IteratorA , typename IteratorB > ZipIterator<IteratorA, IteratorB> embb::algorithms::Zip (IteratorA *iter_a,* IteratorB *iter_b*)

Creates a zip iterator from two iterators.

This is a convenience function which avoids calling the constructor of the templated class.

Returns

Constructed zip iterator

Template Parameters

IteratorA	Type of first iterator
IteratorB	Type of second iterator

Parameters

in	*iter_a*	First iterator
in	*iter_b*	Second iterator

5.17 MTAPI

C++ wrapper around C implementation of MTAPI.

Classes

- class embb::mtapi::Action

  Holds the actual worker function used to execute a Task.
- class embb::mtapi::ActionAttributes

  Contains attributes of an Action.
- class embb::mtapi::Affinity

  Describes the affinity of an Action or Task to a worker thread of a Node.
- class embb::mtapi::ExecutionPolicy

  Describes the execution policy of a parallel algorithm.
- class embb::mtapi::Group

  Represents a facility to wait for multiple related Tasks.
- class embb::mtapi::GroupAttributes

  Contains attributes of a Group.
- class embb::mtapi::Job

  Represents a collection of Actions.
- class embb::mtapi::Node

  A singleton representing the MTAPI runtime.
- class embb::mtapi::NodeAttributes

  Contains attributes of a Node.
- class embb::mtapi::Queue

  Allows for stream processing, either ordered or unordered.
- class embb::mtapi::QueueAttributes

  Contains attributes of a Queue.
- class embb::mtapi::StatusException

  Represents an MTAPI error state and is thrown by almost all mtapi_cpp methods.
- class embb::mtapi::Task

  A Task represents a running Action of a specific Job.
- class embb::mtapi::TaskAttributes

  Contains attributes of a Task.
- class embb::mtapi::TaskContext

  Provides information about the status of the currently running Task.

5.17.1 Detailed Description

C++ wrapper around C implementation of MTAPI.

For a description of the basic concepts, see the C implementation of MTAPI.

5.18 Atomic

Atomic operations.

Classes

- class embb::base::Atomic< BaseType >

  Class representing atomic variables.

5.18.1 Detailed Description

Atomic operations.

5.19 C++ Components

Components written in C++.

Modules

- Containers

  Concurrent data structures, mainly containers.
- Dataflow

  C++ library for parallel, stream-based applications.
- Algorithms

  High-level parallel algorithms and functionalities.
- MTAPI

  C++ wrapper around C implementation of MTAPI.
- Base

  Platform-independent abstraction layer for multithreading and basic operations.

5.19.1 Detailed Description

Components written in C++.

5.20 C++ Concepts

Concepts for C++ components.

Modules

- Stack Concept

    Concept for thread-safe stacks.
- Value Pool Concept

    Concept for thread-safe value pools.
- Queue Concept

    Concept for thread-safe queues.
- Mutex Concept

    Concept for thread synchronization.

5.20.1 Detailed Description

Concepts for C++ components.

5.21 Base

Platform-independent abstraction layer for multithreading and basic operations.

Modules

- Atomic

  Atomic operations.
- Condition Variable

  Condition variables for thread synchronization.
- Core Set

  Core sets for thread-to-core affinities.
- Duration and Time

  Relative time durations and absolute time points.
- Exception

  Exception types.
- Function

  Function wrapper and binding of parameters.
- Logging

  Simple logging facilities.
- Memory Allocation

  Functions, classes, and allocators for dynamic memory allocation.
- Mutex and Lock

  Mutexes and locks for thread synchronization.
- Thread

  Threads supporting thread-to-core affinities.
- Thread-Specific Storage

  Thread specific storage.

5.21.1 Detailed Description

Platform-independent abstraction layer for multithreading and basic operations.

Base C++ is mainly a C++ wrapper around the Base C abstractions. It adds additional convenience types and functions that leverage the capabilities of C++ such as templates, operator overloading, or RAII paradigms.

5.22 Condition Variable

Condition variables for thread synchronization.

Classes

- class embb::base::ConditionVariable

  Represents a condition variable for thread synchronization.

5.22.1 Detailed Description

Condition variables for thread synchronization.

5.23 Core Set

Core sets for thread-to-core affinities.

Classes

- class embb::base::CoreSet

  Represents a set of processor cores, used to set thread-to-core affinities.

5.23.1 Detailed Description

Core sets for thread-to-core affinities.

5.24 Duration and Time

Relative time durations and absolute time points.

Classes

- class embb::base::Duration< Tick >

  Represents a relative time duration for a given tick type.
- class embb::base::Time

  Represents an absolute time point.

Typedefs

- typedef Duration< internal::Seconds > embb::base::DurationSeconds

  Duration with seconds tick.
- typedef Duration< internal::Milliseconds > embb::base::DurationMilliseconds

  Duration with milliseconds tick.
- typedef Duration< internal::Microseconds > embb::base::DurationMicroseconds

  Duration with microseconds tick.
- typedef Duration< internal::Nanoseconds > embb::base::DurationNanoseconds

  Duration with nanoseconds tick.

Functions

- template<typename Tick >
  bool embb::base::operator== (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (equality).
- template<typename Tick >
  bool embb::base::operator!= (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (inequality).
- template<typename Tick >
  bool embb::base::operator< (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (less than)
- template<typename Tick >
  bool embb::base::operator> (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (greater than)
- template<typename Tick >
  bool embb::base::operator<= (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (less than or equal to)
- template<typename Tick >
  bool embb::base::operator>= (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Compares two durations (greater than or equal to)
- template<typename Tick >
  Duration< Tick > embb::base::operator+ (const Duration< Tick > &lhs, const Duration< Tick > &rhs)

  Adds two durations.

5.24.1 Detailed Description

Relative time durations and absolute time points.

5.24.2 Typedef Documentation

5.24.2.1 typedef Duration<internal::Seconds> embb::base::DurationSeconds

Duration with seconds tick.

5.24.2.2 typedef Duration<internal::Milliseconds> embb::base::DurationMilliseconds

Duration with milliseconds tick.

5.24.2.3 typedef Duration<internal::Microseconds> embb::base::DurationMicroseconds

Duration with microseconds tick.

5.24.2.4 typedef Duration<internal::Nanoseconds> embb::base::DurationNanoseconds

Duration with nanoseconds tick.

5.24.3 Function Documentation

5.24.3.1 template<typename Tick > bool embb::base::operator== (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (equality).

Returns

`true` if `lhs` is equal to `rhs`, otherwise `false`

Parameters

in	*lhs*	Left-hand side of equality operator
in	*rhs*	Right-hand side of equality operator

5.24.3.2 template<typename Tick > bool embb::base::operator!= (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (inequality).

Returns

`true` if `lhs` is not equal to `rhs`, otherwise `false`

Parameters

in	*lhs*	Left-hand side of inequality operator
in	*rhs*	Right-hand side of inequality operator

5.24.3.3 template< typename Tick > bool embb::base::operator< (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (less than)

Returns

`true` if `lhs` is shorter than `rhs`.

Parameters

in	*lhs*	Left-hand side of less than operator
in	*rhs*	Right-hand side of less than operator

5.24.3.4 template< typename Tick > bool embb::base::operator> (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (greater than)

Returns

`true` if `lhs` is longer than `rhs`.

Parameters

in	*lhs*	Left-hand side of greater than operator
in	*rhs*	Right-hand side of greater than operator

5.24.3.5 template< typename Tick > bool embb::base::operator<= (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (less than or equal to)

Returns

`true` if `lhs` is shorter than or equal to `rhs`.

Parameters

in	*lhs*	Left-hand side of less than or equal to operator
in	*rhs*	Right-hand side of less than or equal to operator

5.24.3.6 template< typename Tick > bool embb::base::operator>= (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Compares two durations (greater than or equal to)

Returns

`true` if `lhs` is longer than or equal to `rhs`.

Parameters

`in`	*lhs*	Left-hand side of greater than or equal to operator
`in`	*rhs*	Right-hand side of greater than or equal to operator

5.24.3.7 template< typename Tick > Duration<Tick> embb::base::operator+ (const Duration< Tick > & *lhs*, const Duration< Tick > & *rhs*)

Adds two durations.

Returns

Sum of `lhs` and `rhs`.

Parameters

`in`	*lhs*	Left-hand side of addition operator
`in`	*rhs*	Right-hand side of addition operator

5.25 Exception

Exception types.

Classes

- class embb::base::Exception

  Abstract base class for exceptions.
- class embb::base::NoMemoryException

  Indicates lack of memory necessary to allocate a resource.
- class embb::base::ResourceBusyException

  Indicates business (unavailability) of a required resource.
- class embb::base::UnderflowException

  Indicates a numeric underflow.
- class embb::base::OverflowException

  Indicates a numeric overflow.
- class embb::base::ErrorException

  Indicates a general error.

5.25.1 Detailed Description

Exception types.

If exceptions are disabled, i.e., if the library was built without support for exceptions, no exceptions will be thrown. Instead, an error message is printed to `stderr` and the program exits with the code representing the exception.

5.26 Function

Function wrapper and binding of parameters.

Classes

- class embb::base::Placeholder

    Provides placeholders for Function arguments used in Bind()
- class embb::base::Function< ReturnType,... >

    Wraps function pointers, member function pointers, and functors with up to five arguments.

Functions

- template<class ClassType , typename ReturnType , ... >
  Function< ReturnType,[Arg1,..., Arg5]> embb::base::MakeFunction (ClassType &obj, ReturnType(ClassType::*func)([Arg1,..., Arg5]))

    Wraps an object and a member function pointer into a Function.
- template<typename ReturnType , ... >
  Function< ReturnType,[Arg1,..., Arg5]> embb::base::MakeFunction (ReturnType(*func)([Arg1,..., Arg5]))

    Wraps a function pointer into a Function.
- template<typename ReturnType , UnboundArgument , Arg1 , ... >
  Function< ReturnType[, UnboundArgument]> embb::base::Bind (Function< ReturnType, Arg1[,..., Arg5]> func, Arg1 value1,...)

    Binds given values as arguments of `func` into a new Function.

5.26.1 Detailed Description

Function wrapper and binding of parameters.

5.26.2 Function Documentation

5.26.2.1 template<class ClassType , typename ReturnType , ... > Function<ReturnType, [Arg1, ..., Arg5]> embb::base::MakeFunction (ClassType & *obj*, ReturnType(ClassType::*)([Arg1,..., Arg5]) *func*)

Wraps an object and a member function pointer into a Function.

Returns

Function with same return value and argument syntax as `func`

See also

Function

Template Parameters

ClassType	Class that contains the member function pointed to by `func`.
ReturnType	Return type of member function pointed to by `func`
[Arg1,...,Arg5]	(Optional) Types of up to five arguments of the member function

Parameters

`in`	*obj*	Reference to the object with corresponding member function
`in`	*func*	Member function pointer with up to five optional arguments

5.26.2.2 template<typename ReturnType , ... > Function<ReturnType, [Arg1, ..., Arg5]> embb::base::MakeFunction (ReturnType(∗)([Arg1,..., Arg5]) *func*)

Wraps a function pointer into a Function.

Returns

Function with same return value and argument syntax as `func`

See also

Function

Template Parameters

ReturnType	Return type of member function pointed to by `func`.
[Arg1,...,Arg5]	(Optional) Types of up to five arguments of the member function

Parameters

`in`	*func*	Function pointer with up to five optional arguments

5.26.2.3 template<typename ReturnType , UnboundArgument , Arg1 , ... > Function<ReturnType[, UnboundArgument]> embb::base::Bind (Function< ReturnType, Arg1[,..., Arg5]> *func,* Arg1 *value1,* ...)

Binds given values as arguments of `func` into a new Function.

The new Function has no arguments or one, if Placeholder::_1 is given as one of the values. The position of Placeholder::_1 determines which argument of `func` is not bound.

Dynamic memory allocation

Allocates dynamic memory to hold the parameters.

Returns

Function that uses given values as parameters

See also

Placeholder, Function

Template Parameters

ReturnType	Return type of `func` and parameterless function returned
[UnboundArgument]	Type of not bound argument of `func`, only present when a placeholder is used as value in the bind.
Arg1[,...,Arg5]	Types of up to five arguments of the values to bind

Parameters

`in`	*func*	The Function to bind the values (`value1`, ...) to
`in`	*value1*	At least one and up to five values to bind as arguments of `func`. Placeholder::_1 can be used instead of one of the values to keep the corresponding argument of `func` unbound.

5.27 Logging

Simple logging facilities.

Classes

- class embb::base::Log

    Simple logging facilities.

5.27.1 Detailed Description

Simple logging facilities.

5.28 Memory Allocation

Functions, classes, and allocators for dynamic memory allocation.

Classes

- class embb::base::Allocation

    Common (static) functionality for unaligned and aligned memory allocation.
- class embb::base::Allocatable

    Overloaded new/delete operators.
- class embb::base::CacheAlignedAllocatable

    Overloaded new/delete operators.
- class embb::base::Allocator< Type >

    Allocator according to the C++ standard.
- class embb::base::AllocatorCacheAligned< Type >

    Allocator according to the C++ standard.

5.28.1 Detailed Description

Functions, classes, and allocators for dynamic memory allocation.

5.29 Mutex Concept

Concept for thread synchronization.

Classes

- class embb::base::Spinlock

  Spinlock.
- class embb::base::Mutex

  Non-recursive, exclusive mutex.
- class embb::base::RecursiveMutex

  Recursive, exclusive mutex.

5.29.1 Detailed Description

Concept for thread synchronization.

Description

The mutex concept is used for thread synchronization and provides a lock. At any point in time, only one thread can exclusively hold the lock and the lock is held until the thread explicitly releases it.

Requirements

- Let `Mutex` be the mutex type
- Let `m` be an object of type `Mutex`.

Valid Expressions

Expression	Return type	Description
Mutex()	`void`	Constructs a mutex.
m.TryLock()	`bool`	Tries to lock the mutex and immediately returns. Returns `false`, if the mutex could not be acquired (locked), otherwise `true`.
m.Lock()	`void`	Locks the mutex. When the mutex is already locked, the current thread is blocked until the mutex is unlocked.
m.Unlock()	`void`	Unlocks the mutex.

5.30 Mutex and Lock

Mutexes and locks for thread synchronization.

Classes

- class embb::base::Spinlock

    Spinlock.
- class embb::base::Mutex

    Non-recursive, exclusive mutex.
- class embb::base::RecursiveMutex

    Recursive, exclusive mutex.
- class embb::base::LockGuard< Mutex >

    Scoped lock (according to the RAII principle) using a mutex.
- class embb::base::UniqueLock< Mutex >

    Flexible ownership wrapper for a mutex.

UniqueLock Tag Variables

- const DeferLockTag embb::base::defer_lock = DeferLockTag()

    Tag variable for deferred UniqueLock construction.
- const TryLockTag embb::base::try_lock = TryLockTag()

    Tag variable for try-lock UniqueLock construction.
- const AdoptLockTag embb::base::adopt_lock = AdoptLockTag()

    Tag variable for adopt UniqueLock construction.

5.30.1 Detailed Description

Mutexes and locks for thread synchronization.

5.30.2 Variable Documentation

5.30.2.1 const DeferLockTag embb::base::defer_lock = DeferLockTag()

Tag variable for deferred UniqueLock construction.

5.30.2.2 const TryLockTag embb::base::try_lock = TryLockTag()

Tag variable for try-lock UniqueLock construction.

5.30.2.3 const AdoptLockTag embb::base::adopt_lock = AdoptLockTag()

Tag variable for adopt UniqueLock construction.

5.31 Thread

Threads supporting thread-to-core affinities.

Classes

- class embb::base::Thread

  Represents a thread of execution.

Functions

- bool embb::base::operator== (Thread::ID lhs, Thread::ID rhs)

  Compares two thread IDs for equality.
- bool embb::base::operator!= (Thread::ID lhs, Thread::ID rhs)

  Compares two thread IDs for inequality.
- template<class CharT , class Traits >
  std::basic_ostream< CharT, Traits > & embb::base::operator<< (std::basic_ostream< CharT, Traits > &os, Thread::ID id)

  Writes thread ID to stream.

5.31.1 Detailed Description

Threads supporting thread-to-core affinities.

5.31.2 Function Documentation

5.31.2.1 bool embb::base::operator== (Thread::ID *lhs*, Thread::ID *rhs*)

Compares two thread IDs for equality.

Comparison operators need to access the internal ID representation.

Returns

`true` if thread IDs are equivalent, otherwise `false`

Parameters

in	*lhs*	Left-hand side of equality sign
in	*rhs*	Right-hand side of equality sign

5.31.2.2 bool embb::base::operator!= (Thread::ID *lhs*, Thread::ID *rhs*)

Compares two thread IDs for inequality.

Returns

`true` if thread IDs are not equivalent, otherwise `false`

Parameters

`in`	*lhs*	Left-hand side of inequality sign
`in`	*rhs*	Left-hand side of inequality sign

5.31.2.3 template<class CharT , class Traits > std::basic_ostream<CharT, Traits>& embb::base::operator<< (std::basic_ostream< CharT, Traits > & *os*, Thread::ID *id*)

Writes thread ID to stream.

The streaming operator needs to access the internal ID representation.

Returns

Reference to the stream

Parameters

`in,out`	*os*	Stream to which thread ID is written
`in`	*id*	Thread ID to be written

5.32 Thread-Specific Storage

Thread specific storage.

Classes

- class embb::base::ThreadSpecificStorage< Type >
  Represents thread-specific storage (TSS).

5.32.1 Detailed Description

Thread specific storage.

5.33 MTAPI

Multicore Task Management API (MTAPI®).

Modules

- General

  Initialization, introspection, and finalization functions.
- Actions

  Hardware or software implementations of jobs.
- Action Functions

  Executable software functions that implement actions.
- Core Affinities

  Affinities for executing action functions on subsets of cores.
- Queues

  Queues for controlling the scheduling policy of tasks.
- Jobs

  Jobs implementing one or more actions.
- Tasks

  Tasks representing pieces of work "in flight" (similar to a thread handles).
- Task Groups

  Facilities for synchronizing on groups of tasks.
- MTAPI Extensions

  Provides extensions to the standard MTAPI API.

5.33.1 Detailed Description

Multicore Task Management API (MTAPI®).

MTAPI is an API standardized by the `Multicore Association` for leveraging task parallelism on a wide range of embedded devices containing symmetric or asymmetric multicore processors. A description of the basic terms and concepts is given below. More information can be found on the website of the `Multicore Task Management Working Group`.

Definitions

Action	An action is the hardware or software implementation of a job. An action implemented in software consists of the implementation of an action function with a predefined signature. Software actions are registered with the MTAPI runtime and associated with a job. While executing, an action is also associated with a task and task context. Hardware implementations of actions must be known a priori in the MTAPI runtime implementation. There is no standardized way of registering hardware actions because they are highly hardware-dependent. Hardware and software actions are referenced by handles or indirectly through job IDs and job handles.
Action Function	The executable function of an action, invoked by the MTAPI runtime when a task is started.
Affinity	Defines which cores can execute a given action function.
Blocking	A blocking function does not return until the function completes successfully or returns with an error.

Core	A core is an undividable processing element. Two cores can share resources such as memory or ALUs for hyperthreaded cores. The core notion is necessary for core affinity, but is implementation-specific.
Domain	An implementation of MTAPI includes one or more domains, each with one or more nodes. The concept of domains is consistent in all Multicore Association APIs. A domain is comparable to a subnet in a network or a namespace for unique names and IDs. Domains are supported by a runtime.
Handle	An abstract reference to an object on the same node or to an object managed by another node. A handle is valid only on the node on which it was requested and generated. A handle is opaque, that is, its underlying representation is implementation-defined. Handles can be copied, assigned, and passed as arguments, but the application should make no other assumptions about the type, representation, or contents of a handle.
Job	A job provides a way to reference one or more actions. Jobs are abstractions of the processing implemented in hardware or software by actions. Multiple actions can implement the same job based on different hardware resources (for instance a job can be implemented by one action on a DSP and by another action on a general purpose core, or a job can be implemented by both hardware and software actions). Each job is represented by a domain-wide job ID, or by a job handle local to a node.
MCA	The Multicore Association.
MTAPI	Multicore Task Management API, defined by The Multicore Association.
Node	A node represents an independent unit of execution that maps to a process, thread, thread pool, instance of an operating system, hardware accelerator, processor core, a cluster of processor cores, or other abstract processing entity with an independent program counter. Each node can belong to only one domain. The concept of nodes is consistent in all Multicore Associations APIs. Code executed on an MTAPI node shares memory (data) with any other code executed on the same node.
Queue	A software or hardware entity in which tasks are enqueued in a given order. The queue can ensure in-order execution of tasks. Furthermore, queues might implement other scheduling policies that can be configured by setting queue attributes.
Reference	A reference exists when an object or abstract entity has knowledge or access to another object, without regard to the specific means of the implementation.
Resource	A processing core or chip, hardware accelerator, memory region, or I/O.
Remote Memory	Memory that cannot be accessed using standard load and store operations. For example, host memory is remote to a GPU core.
Runtime System	An MTAPI runtime system (or "runtime") is the underlying implementation of MTAPI. The core of the runtime system supports task scheduling and communication with other nodes. Each MTAPI has an MTAPI runtime system.
SMP	SMP is short for symmetric multiprocessing, in which two or more identical processing cores are connected to a shared main memory and are controlled by a single OS instance.
Task	A task is the invocation of an action. A task is associated with a job object, which is associated with one or more actions. A task may optionally be associated with a task group. A task has attributes and an internal state. A task begins its lifetime with a call to mtapi_task_start() or mtapi_task_enqueue(). A task is referenced by a handle of type mtapi_task_hndl_t. After a task has started, it is possible to wait for task completion from other parts of the program. Every task can run exactly once, i.e., the task cannot be started a second time. (Note that in other contexts, the term "task" has a different meaning. Some real-time operating systems use "task" for operating system threads, for example.)
Task Context	Information about the task, accessible by the corresponding action function; useful for action code reflection.

The MTAPI Feature Set

MTAPI supports two programming modes derived from use cases of the working group members:

- **Tasks**
  MTAPI allows a programmer to start tasks and to synchronize on task completion. Tasks are executed by the runtime system, concurrently to other tasks that have been started and have not been completed at that point in time. A task can be implemented by software or by hardware. Tasks can be started from remote nodes, i.e., the implementation can be done on one node, but the starting and synchronization of corresponding tasks can be done on other nodes. The developer decides where to deploy a task implementation. On the executing node, the runtime system selects the cores that execute a particular task. This mapping can be influenced by application-specific attributes. Tasks can start sub-tasks. MTAPI provides a basic mechanism to pass data to the node that executes a task, and back to the calling node.
- **Queues**
  Explicit queues can be used to control the task scheduling policies for related tasks. Order-preserving queues ensure that tasks are executed sequentially in queue order with no subsequent task starting until the previous one is complete. MTAPI also supports non-order-preserving queues, allowing control of the scheduling policies of tasks started via the same queue (queues may offer implementation specific scheduling policies controlled by implementation specific queue attributes). Even hardware queues can be associated with queue objects.

MTAPI also supports the following types of tasks:

- **Single tasks**
  Single tasks are the standard case: After a task is started, the application may wait for completion of the task at a later point in time. In some cases the application waits for completion of a group of tasks. In other cases waiting is not required at all. When a software-implemented task is started, the corresponding code (action function) is executed once by the MTAPI runtime environment. When a hardware-implemented task is started, the task execution is triggered once by the MTAPI runtime system.
- **Multi-instance tasks**
  Multi-instance tasks execute the same action multiple times in parallel (similar to parallel regions in OpenMP or parallel MPI processes).
- **Multiple-implementation tasks / load balancing**
  In heterogeneous systems, there could be implementations of the same job for different types of processor cores, e.g., one general purpose implementation and a second one for a hardware accelerator. MTAPI allows attaching multiple actions to a job. The runtime system shall decide dynamically during runtime, depending on the system load, which action to utilize. Only one of the alternative actions will be executed.

5.34 General

Initialization, introspection, and finalization functions.

Classes

- struct mtapi_info_struct
  Info structure.
- struct mtapi_node_attributes_struct
  Node attributes.

Functions

- void mtapi_initialize (const mtapi_domain_t domain_id, const mtapi_node_t node_id, const mtapi_node_attributes_t *attributes, mtapi_info_t *mtapi_info, mtapi_status_t *status)
  Initializes the MTAPI environment on a given MTAPI node in a given MTAPI domain.
- void mtapi_node_get_attribute (const mtapi_node_t node, const mtapi_uint_t attribute_num, void *attribute, const mtapi_size_t attribute_size, mtapi_status_t *status)
  Given a node and attribute number, returns a copy of the corresponding attribute value in `*attribute`.
- void mtapi_finalize (mtapi_status_t *status)
  Finalizes the MTAPI environment on a given MTAPI node and domain.
- mtapi_domain_t mtapi_domain_id_get (mtapi_status_t *status)
  Returns the domain id associated with the local node.
- mtapi_node_t mtapi_node_id_get (mtapi_status_t *status)
  Returns the node id associated with the local node and domain.

5.34.1 Detailed Description

Initialization, introspection, and finalization functions.

All applications wishing to use MTAPI functionality must use the initialization and finalization routines. After initialization, the introspection functions can provide important information to MTAPI-based applications.

5.34.2 Function Documentation

5.34.2.1 void mtapi_initialize (const mtapi_domain_t *domain_id*, const mtapi_node_t *node_id*, const mtapi_node_attributes_t * *attributes*, mtapi_info_t * *mtapi_info*, mtapi_status_t * *status*)

Initializes the MTAPI environment on a given MTAPI node in a given MTAPI domain.

It must be called on each node using MTAPI. A node maps to a process, thread, thread pool, instance of an operating system, hardware accelerator, processor core, a cluster of processor cores, or another abstract processing entity with an independent program counter. In other words, an MTAPI node is an independent thread of control.

Application software running on an MTAPI node must call mtapi_initialize() once per node. It is an error to call mtapi_initialize() multiple times from a given node, unless mtapi_finalize() is called in between.

The values for `domain_id` and `node_id` must be known a priori by the application and MTAPI.

`mtapi_info` is used to obtain information from the MTAPI implementation, including MTAPI and the underlying implementation version numbers, implementation vendor identification, the number of cores of a node, and vendor-specific implementation information. See the header files for additional information.

A given MTAPI implementation will specify what is a node, i.e., how the concrete system is partitioned into nodes and what are the underlying units of execution executing tasks, e.g., threads, a thread pool, processes, or hardware units.

`attributes` is a pointer to a node attributes object that was previously prepared with mtapi_nodeattr_init() and mtapi_nodeattr_set(). If `attributes` is `MTAPI_NULL`, then the following default attributes will be used:

- all available cores will be used
- the main thread will be reused as a worker
- maximum number of tasks is 1024
- maximum number of groups is 128
- maximum number of queues is 16
- maximum queue capacity is 1024
- maximum number of priorities is 4.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_NODE_INITFAILED`	MTAPI environment could not be initialized.
`MTAPI_ERR_NODE_INITIALIZED`	MTAPI environment was already initialized.
`MTAPI_ERR_NODE_INVALID`	The `node_id` parameter is not valid.
`MTAPI_ERR_DOMAIN_INVALID`	The `domain_id` parameter is not valid.
`MTAPI_ERR_PARAMETER`	Invalid mtapi_node_attributes or mtapi_info.

See also

mtapi_nodeattr_init(), mtapi_nodeattr_set()

Concurrency

Not thread-safe

Dynamic memory allocation

Allocates some memory depending on the node attributes. The amount allocated is returned in the mtapi_info structure.

Parameters

`in`	*domain_id*	Domain id
`in`	*node_id*	Node id
`in`	*attributes*	Pointer to attributes
`out`	*mtapi_info*	Pointer to info struct
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.34.2.2 void mtapi_node_get_attribute (const mtapi_node_t *node*, const mtapi_uint_t *attribute_num*, void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Given a node and attribute number, returns a copy of the corresponding attribute value in `*attribute`.

See mtapi_nodeattr_set() for a list of predefined attribute numbers and the sizes of the attribute values. The application is responsible for allocating sufficient space for the returned attribute value and for setting `attribute_size` to the exact size in bytes of the attribute value.

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value will be written to `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_nodeattr_set()

Concurrency

Thread-safe and wait-free

Parameters

`in`	*node*	Node handle
`in`	*attribute_num*	Attribute id
`out`	*attribute*	Pointer to attribute value
`in`	*attribute_size*	Size of attribute value
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.34.2.3 void mtapi_finalize (mtapi_status_t ∗ *status*)

Finalizes the MTAPI environment on a given MTAPI node and domain.

It has to be called by each node using MTAPI. It is an error to call mtapi_finalize() without first calling mtapi_initialize(). An MTAPI node can call mtapi_finalize() once for each call to mtapi_initialize(), but it is an error to call mtapi_finalize() multiple times from a given node unless mtapi_initialize() has been called prior to each mtapi_finalize() call.

All tasks that have not completed and that have been started on the node where mtapi_finalize() is called will be canceled (see mtapi_task_cancel()). mtapi_finalize() blocks until all tasks that have been started on the same node return (long-running tasks already executing must actively poll the task state and return if canceled). Tasks that execute actions on the node where mtapi_finalize() is called, also block finalization of the MTAPI runtime system on that node. They are canceled as well and return with an `MTAPI_ERR_NODE_NOTINIT` status. Other functions that have a dependency to the node and that are called after mtapi_finalize() also return `MTAPI_ERR_NODE_NOTINIT` (e.g., mtapi_task_get() starting a task associated with an action implemented on the already-finalized node).

mtapi_finalize() may not be called from an action function.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_NODE_FINALFAILED`	The MTAPI environment couldn't be finalized.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_initialize(), mtapi_task_cancel(), mtapi_task_get()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.34.2.4 mtapi_domain_t mtapi_domain_id_get (mtapi_status_t * *status*)

Returns the domain id associated with the local node.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

Returns

Domain id of local node

Concurrency

Thread-safe and wait-free

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.34.2.5 mtapi_node_t mtapi_node_id_get (mtapi_status_t * *status*)

Returns the node id associated with the local node and domain.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Returns

Node id of local node

Concurrency

Thread-safe and wait-free

Parameters

out	*status*	Pointer to error code, may be MTAPI_NULL

5.35 Actions

Hardware or software implementations of jobs.

Classes

- struct mtapi_action_attributes_struct

  Action attributes.
- struct mtapi_action_hndl_struct

  Action handle.

Functions

- mtapi_action_hndl_t mtapi_action_create (const mtapi_job_id_t job_id, const mtapi_action_function_t function, const void ∗node_local_data, const mtapi_size_t node_local_data_size, const mtapi_action_attributes←_t ∗attributes, mtapi_status_t ∗status)

  This function creates a software action (hardware actions are considered to be pre-existent and do not need to be created).
- void mtapi_action_set_attribute (const mtapi_action_hndl_t action, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  This function changes the value of the attribute that corresponds to the given `attribute_num` for this action.
- void mtapi_action_get_attribute (const mtapi_action_hndl_t action, const mtapi_uint_t attribute_num, void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Returns the attribute value that corresponds to the given `attribute_num` for this action.
- void mtapi_action_delete (const mtapi_action_hndl_t action, const mtapi_timeout_t timeout, mtapi_status_t ∗status)

  This function deletes a software action (Hardware actions exist perpetually and cannot be deleted).
- void mtapi_action_disable (const mtapi_action_hndl_t action, const mtapi_timeout_t timeout, mtapi_status←_t ∗status)

  This function disables an action.
- void mtapi_action_enable (const mtapi_action_hndl_t action, mtapi_status_t ∗status)

  This function enables a previously disabled action.

5.35.1 Detailed Description

Hardware or software implementations of jobs.

An action is referenced by an opaque handle of type `mtapi_action_hndl_t`, or indirectly through a handle to a job of type `mtapi_job_hndl_t`. A job refers to all actions implementing the same job, regardless of the node(s) where they are implemented.

An action's lifetime begins when the application successfully calls mtapi_action_create() and obtains a handle to the action. Its lifetime ends upon successful completion of mtapi_action_delete() or mtapi_finalize().

While an opaque handle to an action may be used in the scope of one node only, a job can be used to refer to all its associated actions implementing the same job, regardless of the node where they are implemented. Tasks may be invoked in this way from nodes that do not share memory or even the same ISA with the node where the action resides.

5.35.2 Function Documentation

5.35.2.1 mtapi_action_hndl_t mtapi_action_create (const mtapi_job_id_t *job_id*, const mtapi_action_function_t *function*, const void * *node_local_data*, const mtapi_size_t *node_local_data_size*, const mtapi_action_attributes_t * *attributes*, mtapi_status_t * *status*)

This function creates a software action (hardware actions are considered to be pre-existent and do not need to be created).

It is called on the node where the action function is implemented. An action is an abstract encapsulation of everything needed to implement a job. An action contains attributes, a reference to a job, a reference to an action function, and a reference to node-local data. After an action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`. An action's life-cycle begins with mtapi_action_create(), and ends when mtapi_action_delete() or mtapi_finalize() is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

For non-default behavior, `*attributes` must be prepared with mtapi_actionattr_init() and mtapi_actionattr_set() prior to calling mtapi_action_create(). If attributes is `MTAPI_NULL`, then default attributes will be used.

If `node_local_data_size` is not zero, `node_local_data` specifies the start of node local data shared by action functions executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

On success, an action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the action already exists, `status` will be set to `MTAPI_ERR_ACTION_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The `job_id` is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
`MTAPI_ERR_ACTION_EXISTS`	This action is already created.
`MTAPI_ERR_ACTION_LIMIT`	Exceeded maximum number of actions allowed.
`MTAPI_ERR_ACTION_NOAFFINITY`	The action was created with an `MTAPI_ACTION_AFFINITY` attribute that has set the affinity to all cores of the node to `MTAPI_FALSE`.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.

See also

mtapi_actionattr_init(), mtapi_actionattr_set(), mtapi_action_delete(), mtapi_finalize()

Returns

Handle to newly created action, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*job_id*	Job id
in	*function*	Action function pointer
in	*node_local_data*	Data shared across tasks
in	*node_local_data_size*	Size of shared data
in	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.35.2.2 void mtapi_action_set_attribute (const **mtapi_action_hndl_t** *action*, const mtapi_uint_t *attribute_num*, const void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

This function changes the value of the attribute that corresponds to the given `attribute_num` for this action.

`attribute` must point to the attribute value, and `attribute_size` must be set to the exact size of the attribute value. See mtapi_actionattr_set() for a list of predefined attribute numbers and the sizes of their values.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_actionattr_set()

Concurrency

Not thread-safe

Parameters

in	*action*	Action handle
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.35.2.3 void mtapi_action_get_attribute (const **mtapi_action_hndl_t** *action*, const mtapi_uint_t *attribute_num*, void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Returns the attribute value that corresponds to the given `attribute_num` for this action.

`attribute` must point to the location where the attribute value is to be returned, and `attribute_size` must be set to the exact size of the attribute value. See mtapi_actionattr_set() for a list of predefined attribute numbers and the sizes of their values.

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_actionattr_set()

Concurrency

Thread-safe and wait-free

Parameters

`in`	*action*	Action handle
`in`	*attribute_num*	Attribute id
`out`	*attribute*	Pointer to attribute value
`in`	*attribute_size*	Size of attribute value
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.35.2.4 void mtapi_action_delete (const mtapi_action_hndl_t *action,* const mtapi_timeout_t *timeout,* mtapi_status_t * *status*)

This function deletes a software action (Hardware actions exist perpetually and cannot be deleted).

mtapi_action_delete() may be called by any node that has a valid action handle. Tasks associated with an action that has been deleted may still be executed depending on their internal state:

- If mtapi_action_delete() is called on an action that is currently executing, the associated task's state will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accomplish this, action functions must poll the task state with mtapi_context_taskstate_get(). A call to mtapi_task_wait() on the task executing this code will return the status set by mtapi_context_status_set(), or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are started or enqueued but waiting for execution by the MTAPI runtime when mtapi_action_delete() is called will not be executed anymore if the deleted action is the only action associated with that task. A call to mtapi_task_wait() will return the status `MTAPI_ERR_ACTION_DELETED`.
- Tasks that are started or enqueued after deletion of the action will return `MTAPI_ERR_ACTION_INVALID` if the deleted action is the only action associated with that task.

Calling mtapi_action_get_attribute() on a deleted action will return MTAPI_ERR_ACTION_INVALID if all actions implementing the job had been deleted.

The function mtapi_action_delete() blocks until the corresponding action code is left by all tasks that are executing the code or until the timeout is reached. If timeout is a constant 0 or the symbolic constant MTAPI_NOWAIT, this function only returns MTAPI_SUCCESS if no tasks are executing the action when it is called. If it is set to MTAPI_INFINITE, the function may block infinitely.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ACTION_INVALID	Argument is not a valid action handle.
MTAPI_TIMEOUT	Timeout was reached.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_context_taskstate_get(), mtapi_context_status_set(), mtapi_task_wait()

Concurrency

Thread-safe

Parameters

in	*action*	Action handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.35.2.5 void mtapi_action_disable (const mtapi_action_hndl_t *action*, const mtapi_timeout_t *timeout*, mtapi_status_t * *status*)

This function disables an action.

Tasks associated with an action that has been disabled may still be executed depending on their internal state:

- If mtapi_action_disable() is called on an action that is currently executing, the associated task's state will be set to MTAPI_TASK_CANCELLED and execution will continue. To accomplish this, action functions must poll the task with mtapi_context_taskstate_get(). A call to mtapi_task_wait() on the task executing this code will return the status set by mtapi_context_status_set(), or MTAPI_SUCCESS if not explicitly set.
- Tasks that are started or enqueued but waiting for execution by the MTAPI runtime when mtapi_action_disable() is called will not be executed anymore if the disabled action is the only action associated with that task. A call to mtapi_task_wait() will return the status MTAPI_ERR_ACTION_DISABLED.
- Tasks that are started or enqueued after the action has been disabled will return MTAPI_ERR_ACTION_DISABLED if either the disabled action is the only action associated with a task or all actions associated with a task are disabled. mtapi_action_disable() blocks until all running tasks exit the code, or until the timeout is reached. If timeout is the constant 0 or the symbolic constant MTAPI_NOWAIT, this function only returns MTAPI_SUCCESS if no tasks are executing the action when it is called. If it is set to MTAPI_INFINITE the function may block infinitely.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ACTION_INVALID	Argument is not a valid action handle.
MTAPI_TIMEOUT	Timeout was reached.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_context_taskstate_get(), mtapi_context_status_set(), mtapi_task_wait()

Concurrency

Thread-safe and wait-free

Parameters

in	*action*	Action handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.35.2.6 void mtapi_action_enable (const mtapi_action_hndl_t *action*, mtapi_status_t * *status*)

This function enables a previously disabled action.

If this function is called on an action that no longer exists, an MTAPI_ERR_ACTION_INVALID error will be returned.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ACTION_INVALID	Argument is not a valid action handle.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Concurrency

Thread-safe and wait-free

Parameters

in	*action*	Action handle
out	*status*	Pointer to error code, may be MTAPI_NULL

5.36 Action Functions

Executable software functions that implement actions.

Typedefs

- typedef void(∗ mtapi_action_function_t) (const void ∗args, mtapi_size_t args_size, void ∗result_buffer, mtapi_size_t result_buffer_size, const void ∗node_local_data, mtapi_size_t node_local_data_size, mtapi_task_context_t ∗context)

  An action function is the executable software function that implements an action.

Functions

- void mtapi_context_status_set (mtapi_task_context_t ∗task_context, const mtapi_status_t error_code, mtapi_status_t ∗status)

  This function can be called from an action function to set the status that can be obtained by a subsequent call to mtapi_task_wait() or mtapi_group_wait_any().
- void mtapi_context_runtime_notify (const mtapi_task_context_t ∗task_context, const mtapi_notification_t notification, const void ∗data, const mtapi_size_t data_size, mtapi_status_t ∗status)

  This function can be called from an action function to notify the runtime system.
- mtapi_task_state_t mtapi_context_taskstate_get (const mtapi_task_context_t ∗task_context, mtapi_status_t ∗status)

  An action function may call this function to obtain the state of the task that is associated with the action function.
- mtapi_uint_t mtapi_context_instnum_get (const mtapi_task_context_t ∗task_context, mtapi_status_t ∗status)

  This function can be called from an action function to query the instance number of the associated task.
- mtapi_uint_t mtapi_context_numinst_get (const mtapi_task_context_t ∗task_context, mtapi_status_t ∗status)

  This function can be called from an action function to query the total number of parallel task instances.
- mtapi_uint_t mtapi_context_corenum_get (const mtapi_task_context_t ∗task_context, mtapi_status_t ∗status)

  This function can be called from an action function to query the current core number for debugging purposes.

5.36.1 Detailed Description

Executable software functions that implement actions.

The runtime passes arguments to the action function when a task is started. Passing arguments from one node to another node should be implemented as a copy operation. Just as the arguments are passed before start of execution, the result buffer is copied back to the calling node after the action function terminates. In shared memory environments, the copying of data in both cases is not necessary. The node-local data is data used by several action functions being executed on the same node (or at least in the same address space). The shared data is specified when the action is created.

An action function can interact with the runtime environment through a task context object of type `mtapi_task_context_t`. A task context object is allocated and managed by the runtime. The runtime passes a pointer to the context object when the action function is invoked.The action may then query information about the execution context(e.g., its core number, the number of tasks and task number in a multi - instance task, polling the task state) by calling the `mtapi_context_*` functions. Furthermore it is possible to pass information from the action function to the runtime system which is executing the action function(setting the status manually, for example). All of these `mtapi_context_*` functions are called in the context of task execution.

5.36.2 Typedef Documentation

5.36.2.1 typedef void(∗ mtapi_action_function_t) (const void ∗args,mtapi_size_t args_size,void ∗result_buffer,mtapi_size_t result_buffer_size,const void ∗node_local_data,mtapi_size_t node_local_data_size,mtapi_task_context_t ∗context)

An action function is the executable software function that implements an action.

The runtime passes arguments to the action function when a task is started. Passing arguments from one node to another node should be implemented as a copy operation. Just as the arguments are passed before start of execution, the result buffer is copied back to the calling node after the action function terminates. In shared memory environments, the copying of data in both cases is not necessary. The node-local data is data used by several action functions being executed on the same node (or at least in the same address space). The shared data is specified when the action is created.

An action function can interact with the runtime environment through a task context object of type mtapi_task_↩ context_t . A task context object is allocated and managed by the runtime. The runtime passes a pointer to the context object when the action function is invoked. The action may then query information about the execution context (e.g., its core number, the number of tasks and task number in a multi-instance task, polling the task state) by calling the mtapi_context_∗ functions. Furthermore it is possible to pass information from the action function to the runtime system which is executing the action function (setting the status manually, for example). All of these mtapi_context_∗ functions are called in the context of task execution.

5.36.3 Function Documentation

5.36.3.1 void mtapi_context_status_set (mtapi_task_context_t ∗ *task_context,* const mtapi_status_t *error_code,* mtapi_status_t ∗ *status*)

This function can be called from an action function to set the status that can be obtained by a subsequent call to mtapi_task_wait() or mtapi_group_wait_any().

`task_context` must be the same value as the context parameter that the runtime passes to the action function when it is invoked.

The status can be passed from the action function to the runtime system by setting error_code to one of the following values:

- `MTAPI_SUCCESS` for successful completion
- `MTAPI_ERR_ACTION_CANCELLED` if the action execution is canceled
- `MTAPI_ERR_ACTION_FAILED` if the task could not be completed as intended The error code will be especially important in future versions of MTAPI where tasks shall be chained (flow graphs). The chain execution can then be aborted if the error code is not `MTAPI_SUCCESS`.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_task_wait(), mtapi_group_wait_any()

Concurrency

Not thread-safe

Parameters

in,out	*task_context*	Pointer to task context
in	*error_code*	Task return value
out	*status*	Pointer to error code, may be MTAPI_NULL

5.36.3.2 void mtapi_context_runtime_notify (const mtapi_task_context_t * *task_context*, const mtapi_notification_t *notification*, const void * *data*, const mtapi_size_t *data_size*, mtapi_status_t * *status*)

This function can be called from an action function to notify the runtime system.

This is used to communicate certain states to the runtime implementation to allow it to optimize task execution.

`task_context` must be the same value as the context parameter that the runtime passes to the action function when it is invoked.

The underlying type `mtapi_notification_t` and the valid values for notification are implementation-defined. The notification system is meant to be flexible, and can be used in many ways, for example:

- To trigger prefetching of data for further processing
- To order execution via queues there might be point in the action code where the next task in the queue may be started, even if the current code, started from the same queue, is still executing

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_CONTEXT_OUTOFCONTEXT	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Concurrency

Not thread-safe

Parameters

in	*task_context*	Pointer to task context
in	*notification*	Notification id
in	*data*	Pointer to associated data
in	*data_size*	Size of data
out	*status*	Pointer to error code, may be MTAPI_NULL

5.36.3.3 mtapi_task_state_t mtapi_context_taskstate_get (const mtapi_task_context_t * *task_context,* mtapi_status_t * *status*)

An action function may call this function to obtain the state of the task that is associated with the action function.

`task_context` must be the same value as the context parameter that the runtime passes to the action function when it is invoked.

The underlying representation of type `mtapi_task_state_t` is implementation-defined. Values of type `mtapi_task_state_t` may be copied, assigned, and compared with other values of type `mtapi_task_state_t`, but the caller should make no other assumptions about its type or contents. A minimal implementation must return a status of `MTAPI_TASK_CANCELLED` if the task is canceled, and `MTAPI_TASK_RUNNING` otherwise. Other values of the task state are implementation-defined. This task state can be used to abort a long running computation inside an action function.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

Returns

Task state of current context

Concurrency

Not thread-safe

Parameters

in	*task_context*	Pointer to task context
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.36.3.4 mtapi_uint_t mtapi_context_instnum_get (const mtapi_task_context_t * *task_context,* mtapi_status_t * *status*)

This function can be called from an action function to query the instance number of the associated task.

A task can have multiple instances (multi-instance tasks), in which case the same job is executed multiple times in parallel. Each instance has a number, and this function gives the instance number. Task instances are numbered sequentially, starting at zero.

`task_context` must be the same value as the context parameter that the runtime passes to the action function when it is invoked.

On success, `*status` is set to `MTAPI_SUCCESS` and the task instance number is returned. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_CONTEXT_OUTOFCONTEXT	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Returns

Instance number of current task

Concurrency

Not thread-safe

Parameters

in	*task_context*	Pointer to task context
out	*status*	Pointer to error code, may be MTAPI_NULL

5.36.3.5 mtapi_uint_t mtapi_context_numinst_get (const mtapi_task_context_t * *task_context*, mtapi_status_t * *status*)

This function can be called from an action function to query the total number of parallel task instances.

This value is greater than one for multi-instance tasks.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_CONTEXT_OUTOFCONTEXT	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Returns

Total number of parallel task instances

Concurrency

Not thread-safe

Parameters

in	*task_context*	Pointer to task context
out	*status*	Pointer to error code, may be MTAPI_NULL

5.36.3.6 mtapi_uint_t mtapi_context_corenum_get (const mtapi_task_context_t * *task_context*, mtapi_status_t * *status*)

This function can be called from an action function to query the current core number for debugging purposes.

The core numbering is implementation-defined.

`task_context` must be the same value as the context parameter that the runtime passes to the action function when it was invoked.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

Returns

Worker thread index the current task is running on

Concurrency

Not thread-safe

Parameters

`in`	*task_context*	Pointer to task context
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.37 Core Affinities

Affinities for executing action functions on subsets of cores.

Typedefs

- typedef mtapi_uint64_t mtapi_affinity_t

    Core affinity type.

Functions

- void mtapi_affinity_init (mtapi_affinity_t *mask, const mtapi_boolean_t affinity, mtapi_status_t *status)

    This function initializes an affinity mask object.
- void mtapi_affinity_set (mtapi_affinity_t *mask, const mtapi_uint_t core_num, const mtapi_boolean_t affinity, mtapi_status_t *status)

    This function is used to change the default values of an affinity mask object.
- mtapi_boolean_t mtapi_affinity_get (mtapi_affinity_t *mask, const mtapi_uint_t core_num, mtapi_status_t *status)

    Returns the affinity that corresponds to the given `core_num` for this affinity mask.

5.37.1 Detailed Description

Affinities for executing action functions on subsets of cores.

To set core affinities, the application must allocate an affinity mask object of type `mtapi_affinity_t` and initialize it with a call to mtapi_affinity_init(). Affinities are specified by calling mtapi_affinity_set(). The application must also allocate and initialize an action attributes object of type `mtapi_action_attributes_t`. The affinity mask object is then passed to mtapi_actionattr_set() to set the prescribed affinities in the action attributes object. The action attributes object is then passed to mtapi_action_create() to create a new action with those attributes.

It is in the nature of core affinities to be highly hardware dependent. The least common denominator for different architectures is enabling and disabling core numbers in the affinity mask. Action-to-core affinities can be set via the action attribute `MTAPI_ACTION_AFFINITY` during the creation of an action.

5.37.2 Typedef Documentation

5.37.2.1 typedef mtapi_uint64_t **mtapi_affinity_t**

Core affinity type.

5.37.3 Function Documentation

5.37.3.1 void mtapi_affinity_init (**mtapi_affinity_t** * *mask*, const mtapi_boolean_t *affinity*, mtapi_status_t * *status*)

This function initializes an affinity mask object.

The affinity to all cores will be initialized to the value of affinity. This function should be called prior to calling mtapi_affinity_set() to specify non-default affinity settings. The affinity mask object may then be used to set the `MTAPI_ACTION_AFFINITY` attribute when creating an action with mtapi_action_create().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_AFFINITY_MASK	Invalid mask parameter.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_affinity_set(), mtapi_action_create()

Concurrency

Not thread-safe

Parameters

out	*mask*	Pointer to affinity mask
in	*affinity*	Initial affinity
out	*status*	Pointer to error code, may be MTAPI_NULL

5.37.3.2 void mtapi_affinity_set (mtapi_affinity_t * *mask*, const mtapi_uint_t *core_num*, const mtapi_boolean_t *affinity*, mtapi_status_t * *status*)

This function is used to change the default values of an affinity mask object.

The affinity mask object can then be passed to mtapi_actionattr_set() to set the MTAPI_ACTION_AFFINITY action attribute. An action function will be executed on a core only if the core's affinity is set to MTAPI_TRUE. Calls to mtapi_affinity_set() have no effect on action attributes after the action has been created.

`mask` must be a pointer to an affinity mask object previously initialized with mtapi_affinity_init().

The `core_num` is a hardware- and implementation-specific numeric identifier for a single core of the current node.

On success, `*status` is set to MTAPI_SUCCESS. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_AFFINITY_MASK	Invalid mask parameter.
MTAPI_ERR_CORE_NUM	Unknown core number.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_actionattr_set(), mtapi_affinity_init()

Concurrency

Not thread-safe

Parameters

`in,out`	*mask*	Pointer to affinity mask
`in`	*core_num*	Core number
`in`	*affinity*	Affinity to given core
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.37.3.3 mtapi_boolean_t mtapi_affinity_get (mtapi_affinity_t * *mask*, const mtapi_uint_t *core_num*, mtapi_status_t * *status*)

Returns the affinity that corresponds to the given `core_num` for this affinity mask.

`mask` is a pointer to an affinity mask object previously initialized with mtapi_affinity_init().

Note that affinities may be queried but may not be changed for an action after it has been created. If affinities need to be modified at runtime, new actions must be created.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_AFFINITY_MASK`	Invalid mask parameter.
`MTAPI_ERR_CORE_NUM`	Unknown core number.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_affinity_init()

Returns

`MTAPI_TRUE` if affinity to `core_num` is set, `MTAPI_FALSE` otherwise

Concurrency

Thread-safe and wait-free

Parameters

`out`	*mask*	Pointer to affinity mask
`in`	*core_num*	Core number
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.38 Queues

Queues for controlling the scheduling policy of tasks.

Classes

- struct mtapi_queue_attributes_struct

  Queue attributes.
- struct mtapi_queue_hndl_struct

  Queue handle.

Functions

- mtapi_queue_hndl_t mtapi_queue_create (const mtapi_queue_id_t queue_id, const mtapi_job_hndl_t job, const mtapi_queue_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function creates a software queue object and associates it with the specified job.
- void mtapi_queue_set_attribute (const mtapi_queue_hndl_t queue, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Changes the attribute value that corresponds to the given `attribute_num` for the specified queue.
- void mtapi_queue_get_attribute (const mtapi_queue_hndl_t queue, const mtapi_uint_t attribute_num, void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Returns the attribute value that corresponds to the given `attribute_num` for the specified queue.
- mtapi_queue_hndl_t mtapi_queue_get (const mtapi_queue_id_t queue_id, const mtapi_domain_t domain_id, mtapi_status_t ∗status)

  This function converts a domain-wide `queue_id` into a node-local queue handle.
- void mtapi_queue_delete (const mtapi_queue_hndl_t queue, const mtapi_timeout_t timeout, mtapi_status_t ∗status)

  This function deletes the specified software queue.
- void mtapi_queue_disable (const mtapi_queue_hndl_t queue, const mtapi_timeout_t timeout, mtapi_status↩ _t ∗status)

  This function disables the specified queue in such a way that it can be resumed later.
- void mtapi_queue_enable (const mtapi_queue_hndl_t queue, mtapi_status_t ∗status)

  This function may be called from any node with a valid queue handle to re-enable a queue previously disabled with mtapi_queue_disable().

5.38.1 Detailed Description

Queues for controlling the scheduling policy of tasks.

The default scheduling policy for queues is ordered task execution. Tasks that have to be executed sequentially are enqueued into the same queue. In this case every queue is associated with exactly one action. Tasks started via different queues can be executed in parallel. This is needed for packet processing applications, for example: each stream is processed by one queue. This ensures sequential processing of packets belonging to the same stream. Different streams are processed in parallel.

Queues were made explicit in MTAPI. This allows mapping of queues onto hardware queues, if available. One MTAPI queue is associated with one action, or for purposes of load balancing, with actions implementing the same job on different nodes.

5.38.2 Function Documentation

5.38.2.1 mtapi_queue_hndl_t mtapi_queue_create (const mtapi_queue_id_t *queue_id,* const mtapi_job_hndl_t *job,* const mtapi_queue_attributes_t * *attributes,* mtapi_status_t * *status*)

This function creates a software queue object and associates it with the specified job.

A job is associated with one or more actions that provide the executable implementation of the job. Hardware queues are considered to be pre-existent and do not need to be created.

`queue_id` is an identifier of implementation-defined type that must be supplied by the application. If `queue_id` is set to `MTAPI_QUEUE_ID_NONE`, the queue will be accessible only on the node on which it was created by using the returned queue handle. Otherwise the application may supply a `queue_id` by which the queue can be referenced domain-wide using mtapi_queue_get() to convert the id into a handle. The minimum and maximum values for `queue_id` may be derived from `MTAPI_MIN_USER_QUEUE_ID` and `MTAPI_MAX_USER_QUEU↩ E_ID`.

job is a handle to a job obtained by a previous call to mtapi_job_get(). If `attributes` is `MTAPI_NULL`, the queue will be created with default attribute values. Otherwise `attributes` must point to a queue attributes object previously prepared using mtapi_queueattr_init() and mtapi_queueattr_set().

There is an implementation-defined maximum number of queues permitted.

If more than one action is associated with the job, the runtime system chooses dynamically which action is used for execution (for load balancing purposes).

On success, a queue handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the queue already exists, `*status` will be set to `MTAPI↩ _QUEUE_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_QUEUE_INVALID`	The `queue_id` is not a valid queue id.
`MTAPI_ERR_QUEUE_EXISTS`	This queue is already created.
`MTAPI_ERR_QUEUE_LIMIT`	Exceeded maximum number of queues allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_JOB_INVALID`	The associated job is not valid.

See also

mtapi_queue_get(), mtapi_job_get(), mtapi_queueattr_init(), mtapi_queueattr_set()

Returns

Handle to newly created queue, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*queue←_id*	Queue id
in	*job*	Job handle
in	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.38.2.2 void mtapi_queue_set_attribute (const **mtapi_queue_hndl_t** *queue*, const mtapi_uint_t *attribute_num*, const void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Changes the attribute value that corresponds to the given `attribute_num` for the specified queue.

See mtapi_queueattr_set() for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_queueattr_set()

Concurrency

Not thread-safe

Parameters

in	*queue*	Queue handle
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.38.2.3 void mtapi_queue_get_attribute (const **mtapi_queue_hndl_t** *queue*, const mtapi_uint_t *attribute_num*, void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Returns the attribute value that corresponds to the given `attribute_num` for the specified queue.

attribute must point to a location in memory sufficiently large to hold the returned attribute value. See mtapi_queueattr_set() for a list of predefined attribute numbers and the sizes of the attribute values. The application must set attribute_size to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

On success, *status is set to MTAPI_SUCCESS and the attribute value is returned in *attribute. On error, *status is set to the appropriate error defined below and the *attribute value is undefined. If this function is called on a queue that no longer exists, an MTAPI_ERR_QUEUE_INVALID error will be returned.

Error code	Description
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_QUEUE_INVALID	Argument is not a valid queue handle.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_queueattr_set()

Concurrency

Thread-safe and wait-free

Parameters

in	*queue*	Queue handle
in	*attribute_num*	Attribute id
out	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value
out	*status*	Pointer to error code, may be MTAPI_NULL

5.38.2.4 mtapi_queue_hndl_t mtapi_queue_get (const mtapi_queue_id_t *queue_id*, const mtapi_domain_t *domain_id*, mtapi_status_t * *status*)

This function converts a domain-wide queue_id into a node-local queue handle.

queue_id must match the queue_id that was associated with a software queue that was created with mtapi_queue_create(), or it must be a valid predefined queue identifier known a priori to the runtime and application (e.g., to reference a hardware queue. The minimum and maximum values for queue_id may be derived from MTAPI_MIN_USER_QUEUE_ID and MTAPI_MAX_USER_QUEUE_ID.

On success, the queue handle is returned and *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below. If this function is called on a queue that no longer exists, an MTAPI_ERR_QUEUE_INVALID error will be returned.

Error code	Description
MTAPI_ERR_QUEUE_INVALID	The queue_id parameter does not refer to a valid queue or it is set to MTAPI_QUEUE_ID_ANY.
MTAPI_ERR_NODE_NOTINIT	The node/domain is not initialized.
MTAPI_ERR_DOMAIN_NOTSHARED	This resource cannot be shared by this domain.

See also

mtapi_queue_create()

Returns

Handle to preexisting queue with given `queue_id`, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*queue_id*	Queue id
in	*domain_id*	Domain id
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.38.2.5 void mtapi_queue_delete (const mtapi_queue_hndl_t *queue*, const mtapi_timeout_t *timeout*, mtapi_status_t ∗ *status*)

This function deletes the specified software queue.

Hardware queues are perpetual and cannot be deleted.

`queue` must be a valid handle to an existing queue.

`timeout` determines how long the function should wait for tasks already started via that queue to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function deletes the queue and returns immediately. If `timeout` is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

This function can be called from any node that has a valid queue handle. Tasks previously enqueued in a queue that has been deleted may still be executed depending on their internal state:

- If mtapi_queue_delete() is called on a queue that is currently executing an action, the task state of the corresponding task will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accomplish this, the action function must poll the task state with mtapi_context_taskstate_get(). A call to mtapi_task_wait() on the task executing this code will return the status set by mtapi_context_status_set(), or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are enqueued and waiting for execution by the MTAPI runtime environment when mtapi_queue_delete() is called will not be executed any more. A call to mtapi_task_wait() will return the status `MTAPI_ERR_QUEUE_DELETED`.
- Tasks that are enqueued after deletion of the queue will return a status of `MTAPI_ERR_QUEUE_INVALID`.

If this function is called on a queue that no longer exists, an `MTAPI_ERR_QUEUE_INVALID` status will be returned. A call to mtapi_queue_get() on a deleted queue will return `MTAPI_ERR_QUEUE_INVALID` as well, as long as no new queue has been created for the same queue ID.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_QUEUE_INVALID	Argument is not a valid queue handle.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_TIMEOUT	Timeout was reached.

See also

mtapi_context_taskstate_get(), mtapi_context_status_set(), mtapi_task_wait(), mtapi_queue_get()

Concurrency

Thread-safe

Parameters

in	*queue*	Queue handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.38.2.6 void mtapi_queue_disable (const mtapi_queue_hndl_t *queue*, const mtapi_timeout_t *timeout*, mtapi_status_t * *status*)

This function disables the specified queue in such a way that it can be resumed later.

This is needed to perform certain maintenance tasks. It can be called by any node that has a valid queue handle.

timeout determines how long the function should wait for tasks already started via that queue to finish. The underlying type of mtapi_timeout_t is implementation-defined. If timeout is a constant 0 or the symbolic constant MTAPI_NOWAIT, this function deletes the queue and returns immediately. If timeout is set to MTAPI_INFINITE the function may block infinitely. Other values for timeout and the units of measure are implementation defined.

Tasks previously enqueued in a queue that has been disabled may still be executed depending on their internal state:

- If mtapi_queue_disable() is called on a queue that is currently executing an action, the task state of the corresponding task will be set to MTAPI_TASK_CANCELLED and execution will continue. To accomplish this, the action function must poll the task state by calling mtapi_context_taskstate_get(). A call to mtapi_task_wait() on the task executing this code will return the status set by mtapi_context_status_set(), or MTAPI_SUCCESS if not explicitly set.
- Tasks that are enqueued and waiting for execution by the MTAPI runtime environment when mtapi_queue_disable() is called will not be executed anymore. They will be held in anticipation the queue is enabled again if the MTAPI_QUEUE_RETAIN attribute is set to MTAPI_TRUE. A call to mtapi_task_wait() will return the status MTAPI_ERR_QUEUE_DISABLED.
- Tasks that are enqueued after the queue had been disabled will return MTAPI_ERR_QUEUE_DISABLED if the MTAPI_QUEUE_RETAIN attribute is set to MTAPI_FALSE.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_QUEUE_INVALID	Argument is not a valid queue handle.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_TIMEOUT	Timeout was reached.

See also

mtapi_context_taskstate_get(), mtapi_context_status_set(), mtapi_task_wait()

Concurrency

Thread-safe

Parameters

in	*queue*	Queue handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.38.2.7 void mtapi_queue_enable (const **mtapi_queue_hndl_t** *queue*, mtapi_status_t * *status*)

This function may be called from any node with a valid queue handle to re-enable a queue previously disabled with mtapi_queue_disable().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_QUEUE_INVALID	Argument is not a valid queue handle.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Concurrency

Thread-safe

Parameters

in	*queue*	Queue handle
out	*status*	Pointer to error code, may be MTAPI_NULL

5.39 Jobs

Jobs implementing one or more actions.

Classes

- struct mtapi_job_hndl_struct
  Job handle.
- struct mtapi_ext_job_attributes_struct
  Job attributes.

Functions

- mtapi_job_hndl_t mtapi_job_get (const mtapi_job_id_t job_id, const mtapi_domain_t domain_id, mtapi_status_t ∗status)
  Given a `job_id`, this function returns the MTAPI handle for referencing the actions implementing the job.
- void mtapi_ext_job_set_attribute (MTAPI_IN mtapi_job_hndl_t job, MTAPI_IN mtapi_uint_t attribute_num, MTAPI_IN void ∗attribute, MTAPI_IN mtapi_size_t attribute_size, MTAPI_OUT mtapi_status_t ∗status)
  This function changes the value of the attribute that corresponds to the given `attribute_num` for this job.

5.39.1 Detailed Description

Jobs implementing one or more actions.

An action is a hardware or software implementation of a job. In some cases, an action is referenced by an action handle, while in other cases, an action is referenced indirectly through a job handle. Each job is represented by a domain-wide job ID, or by a job handle which is local to one node.

Several actions can implement the same job based on different hardware resources (for instance a job can be implemented by one action on a DSP and by another action on a general purpose core, or a job can be implemented by both hardware and software actions).

5.39.2 Function Documentation

5.39.2.1 mtapi_job_hndl_t mtapi_job_get (const mtapi_job_id_t *job_id,* const mtapi_domain_t *domain_id,* mtapi_status_t ∗ *status*)

Given a `job_id`, this function returns the MTAPI handle for referencing the actions implementing the job.

This function converts a domain-wide job ID into a node-local job handle.

On success, the action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The job ID does not refer to a valid action.
`MTAPI_ERR_DOMAIN_NOTSHARED`	The resource can't be shared by this domain.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

Returns

Handle to job with given `job_id`, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*job_id*	Job id
in	*domain←_id*	Domain id
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.39.2.2 void mtapi_ext_job_set_attribute (MTAPI_IN mtapi_job_hndl_t *job*, MTAPI_IN mtapi_uint_t *attribute_num*, MTAPI_IN void * *attribute*, MTAPI_IN mtapi_size_t *attribute_size*, MTAPI_OUT mtapi_status_t * *status*)

This function changes the value of the attribute that corresponds to the given `attribute_num` for this job.

`attribute` must point to the attribute value, and `attribute_size` must be set to the exact size of the attribute value.

MTAPI-defined action attributes:

Attribute num	Description	Data Type	Default
MTAPI_JOB_PROBLEM_←SIZE_FUNCTION	Function to calculate the relative problem size of tasks started on this job.	mtapi_ext_problem_size_←function_t	MTAPI_NULL
MTAPI_JOB_DEFAULT_P←ROBLEM_SIZE	Indicates the default relative problem size of tasks started on this job	mtapi_uint_t	1

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to one of the errors defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_JOB_INVALID`	Argument is not a valid job handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

Concurrency

Not thread-safe

Parameters

in	*job*	Action handle
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value

Parameters

in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.40 Tasks

Tasks representing pieces of work "in flight" (similar to a thread handles).

Classes

- struct mtapi_task_attributes_struct

  Task attributes.
- struct mtapi_task_hndl_struct

  Task handle.

Functions

- mtapi_task_hndl_t mtapi_task_start (const mtapi_task_id_t task_id, const mtapi_job_hndl_t job, const void ∗arguments, const mtapi_size_t arguments_size, void ∗result_buffer, const mtapi_size_t result_size, const mtapi_task_attributes_t ∗attributes, const mtapi_group_hndl_t group, mtapi_status_t ∗status)

  This function schedules a task for execution.
- mtapi_task_hndl_t mtapi_task_enqueue (const mtapi_task_id_t task_id, const mtapi_queue_hndl_t queue, const void ∗arguments, const mtapi_size_t arguments_size, void ∗result_buffer, const mtapi_size_t result_↩ size, const mtapi_task_attributes_t ∗attributes, const mtapi_group_hndl_t group, mtapi_status_t ∗status)

  This function schedules a task for execution using a queue.
- void mtapi_task_get_attribute (const mtapi_task_hndl_t task, const mtapi_uint_t attribute_num, void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Returns a copy of the attribute value that corresponds to the given `attribute_num` for the specified task.
- void mtapi_task_cancel (const mtapi_task_hndl_t task, mtapi_status_t ∗status)

  This function cancels a task and sets the task status to `MTAPI_TASK_CANCELLED`.
- void mtapi_task_wait (const mtapi_task_hndl_t task, const mtapi_timeout_t timeout, mtapi_status_t ∗status)

  This function waits for the completion of the specified task.

5.40.1 Detailed Description

Tasks representing pieces of work "in flight" (similar to a thread handles).

A task is associated with a job object, which is associated with one or more actions implementing the same job for load balancing purposes. A task may optionally be associated with a task group. A task has attributes, and an internal state. A task begins its lifetime with a call to mtapi_task_start() or mtapi_task_enqueue(). A task is referenced by a handle of type `mtapi_task_hndl_t`. The underlying type of `mtapi_task_hndl_t` is implementation defined. Task handles may be copied, assigned, and passed as arguments, but otherwise the application should make no assumptions about the internal representation of a task handle.

Once a task is started, it is possible to wait for task completion from other parts of the program.

5.40.2 Function Documentation

5.40.2.1 mtapi_task_hndl_t mtapi_task_start (const mtapi_task_id_t *task_id*, const mtapi_job_hndl_t *job*, const void ∗ *arguments*, const mtapi_size_t *arguments_size*, void ∗ *result_buffer*, const mtapi_size_t *result_size*, const mtapi_task_attributes_t ∗ *attributes*, const mtapi_group_hndl_t *group*, mtapi_status_t ∗ *status*)

This function schedules a task for execution.

A task is associated with a job. A job is associated with one or more actions. An action provides an action function, which is the executable implementation of a job. If more than one action is associated with the job, the runtime system chooses dynamically which action is used for execution for load balancing purposes.

`task_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_TASK_ID_NONE`. The minimum and maximum values for `task_id` may be derived from `MTAPI_MIN_USER_TASK_ID` and `MTAPI_MAX_USER_TASK_ID`.

`job` must be a handle to a job obtained by a previous call to mtapi_job_get().

If `arguments_size` is not zero, then arguments must point to data of `arguments_size` bytes. The arguments will be transferred by the runtime from the node where the action was created to the executing node if necessary. Marshalling of arguments is not part of the MTAPI specification and is implementation-defined.

If `attributes` is `MTAPI_NULL`, the task will be started with default attribute values. Otherwise `attributes` must point to a task attributes object previously prepared using mtapi_taskattr_init() and mtapi_taskattr_set(). The attributes of a task cannot be changed after the task is created.

`group` must be set to `MTAPI_GROUP_NONE` if the task is not part of a task group. Otherwise `group` must be a group handle obtained by a previous call to mtapi_group_create().

On success, a task handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_TASK_LIMIT`	Exceeded maximum number of tasks allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid group handle.
`MTAPI_ERR_JOB_INVALID`	The associated job is not valid.

See also

mtapi_job_get(), mtapi_taskattr_init(), mtapi_taskattr_set(), mtapi_group_create()

Returns

Handle to newly started task, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*task_id*	Task id
in	*job*	Job handle
in	*arguments*	Pointer to arguments
in	*arguments_size*	Size of arguments
out	*result_buffer*	Pointer to result buffer
in	*result_size*	Size of one result
in	*attributes*	Pointer to attributes
in	*group*	Group handle, may be `MTAPI_GROUP_NONE`
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.40.2.2 mtapi_task_hndl_t mtapi_task_enqueue (const mtapi_task_id_t *task_id,* const mtapi_queue_hndl_t *queue,* const void ∗ *arguments,* const mtapi_size_t *arguments_size,* void ∗ *result_buffer,* const mtapi_size_t *result_size,* const mtapi_task_attributes_t ∗ *attributes,* const mtapi_group_hndl_t *group,* mtapi_status_t ∗ *status*)

This function schedules a task for execution using a queue.

A queue is a task associated with a job. A job is associated with one or more actions. An action provides an action function, which is the executable implementation of a job.

`task_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_TASK_ID_NONE`. The underlying type of `mtapi_task_id_t` is implementation-defined. The minimum and maximum values for `task_id` may be derived from `MTAPI_MIN_USER_TASK_ID` and `MTAPI_MAX_USER_TASK_ID`.

`queue` must be a handle to a queue obtained by a previous call to mtapi_queue_create().

If `arguments_size` is not zero, then arguments must point to data of `arguments_size` bytes. The arguments will be transferred by the runtime from the node where the action was created to the executing node. Marshalling of arguments is not part of the MTAPI specification and is implementation-defined.

If `attributes` is `MTAPI_NULL`, the task will be started with default attribute values. Otherwise `attributes` must point to a task attributes object previously prepared using mtapi_taskattr_init() and mtapi_taskattr_set(). Once a task has been enqueued, its attributes may not be changed.

`group` must be set to `MTAPI_GROUP_NONE` if the task is not part of a task group. Otherwise `group` must be a group handle obtained by a previous call to mtapi_group_create().

On success, a task handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_TASK_LIMIT`	Exceeded maximum number of tasks allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.

See also

mtapi_queue_create(), mtapi_taskattr_init(), mtapi_taskattr_set(), mtapi_group_create()

Returns

Handle to newly enqueued task, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*task_id*	Task id
in	*queue*	Queue handle
in	*arguments*	Pointer to arguments
in	*arguments_size*	Size of arguments
out	*result_buffer*	Pointer to result buffer
in	*result_size*	Size of one result
in	*attributes*	Pointer to task attributes
in	*group*	Group handle, may be `MTAPI_GROUP_NONE`
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.40.2.3 void mtapi_task_get_attribute (const **mtapi_task_hndl_t** *task,* const mtapi_uint_t *attribute_num,* void ∗ *attribute,* const mtapi_size_t *attribute_size,* mtapi_status_t ∗ *status*)

Returns a copy of the attribute value that corresponds to the given `attribute_num` for the specified task.

The attribute value will be returned in `*attribute`. Note that task attributes may be queried but may not be changed after a task has been created.

`task` must be a valid handle to a task that was obtained by a previous call to mtapi_task_start() or mtapi_task_enqueue().

See mtapi_taskattr_set() for a list of predefined attribute numbers and the sizes of the attribute values. The application is responsible for allocating sufficient space for the returned attribute value and for setting `attribute_size` to the exact size in bytes of the attribute value.

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. If this function is called on a task that no longer exists, an `MTAPI_ERR_TASK_INVALID` error code will be returned.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_TASK_INVALID`	Argument is not a valid task handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_task_start(), mtapi_task_enqueue(), mtapi_taskattr_set()

Concurrency

Thread-safe and wait-free

Parameters

in	*task*	Task handle
in	*attribute_num*	Attribute id
out	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.40.2.4 void mtapi_task_cancel (const **mtapi_task_hndl_t** *task*, mtapi_status_t ∗ *status*)

This function cancels a task and sets the task status to `MTAPI_TASK_CANCELLED`.

`task` must be a valid handle to a task that was obtained by a previous call to mtapi_task_start() or mtapi_task_↩ enqueue().

If the execution of a task has not been started, the runtime system might remove the task from the runtime-internal task queues. If task execution is already running, an action function implemented in software can poll the task status and react accordingly.

Since the task is referenced by a task handle which can only be used node-locally, a task can be canceled only on the node where the task was created.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_TASK_INVALID`	Argument is not a valid task handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_task_start(), mtapi_task_enqueue()

Concurrency

Thread-safe and wait-free

Parameters

in	*task*	Task handle
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.40.2.5 void mtapi_task_wait (const **mtapi_task_hndl_t** *task*, const mtapi_timeout_t *timeout*, mtapi_status_t ∗ *status*)

This function waits for the completion of the specified task.

`task` must be a valid handle to a task that was obtained by a previous call to mtapi_task_start() or mtapi_task↩ _enqueue(). The task handle becomes invalid on a successful wait, i.e., after the task had run to completion and mtapi_task_wait() returns `MTAPI_SUCCESS`.

timeout determines how long the function should wait for tasks already started via that queue to finish. The underlying type of mtapi_timeout_t is implementation-defined. If timeout is a constant 0 or the symbolic constant MTAPI_NOWAIT, this function does not block and returns immediately. If timeout is set to MTAPI_INFINITE the function may block infinitely. Other values for timeout and the units of measure are implementation-defined.

Results of completed tasks can be obtained via result_buffer associated with the task. The size of the buffer has to be equal to the result size written in the action code. If the result is not needed by the calling code, result←_buffer may be set to MTAPI_NULL. For multi-instance tasks, the result buffer is filled by an array of all the task instances' results. I.e., the result buffer has to be allocated big enough (number of instances times size of result).

Calling mtapi_task_wait() more than once for the same task results in undefined behavior.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below. If this function is called on a task that no longer exists, an MTAPI_ERR_TASK_INVALID error code will be returned. status will be MTAPI_ERR_ARG_SIZE or MTAPI_ERR_RESULT_SIZE if the sizes of arguments or result buffer do not match.

Error code	Description
MTAPI_ERR_TASK_INVALID	Argument is not a valid task handle.
MTAPI_TIMEOUT	Timeout was reached.
MTAPI_ERR_PARAMETER	Invalid timeout parameter.
MTAPI_ERR_TASK_CANCELLED	The task has been canceled because of mtapi_task_cancel() was called before the task was executed or the error code was set to M←TAPI_ERR_TASK_CANCELLED by mtapi_context_status_set() in the action function.
MTAPI_ERR_WAIT_PENDING	mtapi_task_wait() had already been called for the same task and the first wait call is still pending.
MTAPI_ERR_ACTION_CANCELLED	Action execution was canceled by the action function (mtapi_←context_status_set()).
MTAPI_ERR_ACTION_FAILED	Error set by action function (mtapi_context_status_set()).
MTAPI_ERR_ACTION_DELETED	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action function to MTAPI_ERR_ACTION_DELETED by mtapi←_context_status_set().
MTAPI_ERR_ARG_SIZE	The size of the arguments expected by action differs from arguments size of the caller.
MTAPI_ERR_RESULT_SIZE	The size of the result buffer expected by action differs from result buffer size of the caller.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_task_start(), mtapi_task_enqueue(), mtapi_task_wait(), mtapi_task_cancel(), mtapi_context_status_←set()

Concurrency

Thread-safe

Parameters

in	*task*	Task handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41 Task Groups

Facilities for synchronizing on groups of tasks.

Classes

- struct mtapi_group_attributes_struct

  Group attributes.
- struct mtapi_group_hndl_struct

  Group handle.

Functions

- mtapi_group_hndl_t mtapi_group_create (const mtapi_group_id_t group_id, const mtapi_group_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function creates a task group and returns a handle to the group.
- void mtapi_group_set_attribute (const mtapi_group_hndl_t group, const mtapi_uint_t attribute_num, void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Changes the value of the attribute that corresponds to the given `attribute_num` for the specified task group.
- void mtapi_group_get_attribute (const mtapi_group_hndl_t group, const mtapi_uint_t attribute_num, void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  Returns the attribute value that corresponds to the given `attribute_num` for this task group.
- void mtapi_group_wait_all (const mtapi_group_hndl_t group, const mtapi_timeout_t timeout, mtapi_status_t ∗status)

  This function waits for the completion of a task group.
- void mtapi_group_wait_any (const mtapi_group_hndl_t group, void ∗∗result, const mtapi_timeout_t timeout, mtapi_status_t ∗status)

  This function waits for the completion of any task in a task group.
- void mtapi_group_delete (const mtapi_group_hndl_t group, mtapi_status_t ∗status)

  This function deletes a task group.

5.41.1 Detailed Description

Facilities for synchronizing on groups of tasks.

This concept is similar to barrier synchronization of threads. MTAPI specifies a minimal task group feature set in order to allow small and efficient implementations.

5.41.2 Function Documentation

5.41.2.1 mtapi_group_hndl_t mtapi_group_create (const mtapi_group_id_t *group_id*, const mtapi_group_attributes_t ∗ *attributes*, mtapi_status_t ∗ *status*)

This function creates a task group and returns a handle to the group.

After a group is created, a task may be associated with a group when the task is started with mtapi_task_start() or mtapi_task_enqueue().

`group_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_GROUP_ID_NONE`. The underlying type of `mtapi_group_id_t` is implementation-defined. The minimum and maximum values for `group_id` may be derived from `MTAPI_MIN_USER_GROUP_ID` and `MT↵ API_MAX_USER_GROUP_ID`.

If `attributes` is `MTAPI_NULL`, the group will be created with default attribute values. Otherwise `attributes` must point to a group attributes object previously prepared using mtapi_groupattr_init() and mtapi_groupattr_set().

On success, a group handle is returned and `∗status` is set to `MTAPI_SUCCESS`. On error, `∗status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_GROUP_LIMIT	Exceeded maximum number of groups allowed.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_ERR_PARAMETER	Invalid attributes parameter.

See also

mtapi_task_start(), mtapi_task_enqueue(), mtapi_groupattr_init(), mtapi_groupattr_set()

Returns

Handle to newly created group, invalid handle on error

Concurrency

Thread-safe

Dynamic memory allocation

This function allocates a new queue for tracking completion of the tasks belonging to the group.

Parameters

in	*group_id*	Group id
in	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41.2.2 void mtapi_group_set_attribute (const **mtapi_group_hndl_t** *group*, const mtapi_uint_t *attribute_num*, void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Changes the value of the attribute that corresponds to the given `attribute_num` for the specified task group.

`attribute` must point to the attribute value, and `attribute_size` must be set to the exact size of the attribute value. See mtapi_groupattr_set() for a list of predefined attribute numbers and the sizes of their values.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_GROUP_INVALID	Argument is not a valid group handle.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_groupattr_set()

Concurrency

Not thread-safe

Parameters

in	*group*	Group handle
in	*attribute_num*	Attribute id
out	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41.2.3 void mtapi_group_get_attribute (const mtapi_group_hndl_t *group*, const mtapi_uint_t *attribute_num*, void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

Returns the attribute value that corresponds to the given `attribute_num` for this task group.

`attribute` must point to the location where the attribute value is to be returned, and `attribute_size` must be set to the exact size of the attribute value. See mtapi_groupattr_set() for a list of predefined attribute numbers and the sizes of their values.

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined. If this function is called on a group that no longer exists, an `MTAPI_ERR_GROUP_INVALID` error code will be returned.

Error code	Description
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_GROUP_INVALID	Argument is not a valid group handle.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_groupattr_set()

Concurrency

Thread-safe and wait-free

Parameters

in	*group*	Group handle
in	*attribute_num*	Attribute id
out	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41.2.4 void mtapi_group_wait_all (const mtapi_group_hndl_t *group*, const mtapi_timeout_t *timeout*, mtapi_status_t * *status*)

This function waits for the completion of a task group.

Tasks may be associated with groups when the tasks are started. Each task is associated with one or more actions. This function returns when all the associated action functions have completed or canceled. The group handle becomes invalid if this function returns `MTAPI_SUCCESS`.

`timeout` determines how long the function should wait for tasks already started in the group to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function does not block and returns immediately. If `timeout` is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

To obtain results from a task, the application should call mtapi_group_wait_any() instead.

During execution, an action function may optionally call mtapi_context_status_set() to set a task status that will be returned in this function in `*status`. If multiple action functions set different task status values, it is implementation-defined which of those is returned in mtapi_group_wait_all(). The following task status values may be set by an action function: `MTAPI_ERR_TASK_CANCELLED`, `MTAPI_ERR_ACTION_CANCELLED`, `MTAPI_ERR_ACTION_FAILED`, and `MTAPI_ERR_ACTION_DELETED`.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_TIMEOUT`	Timeout was reached.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid task handle.
`MTAPI_ERR_WAIT_PENDING`	mtapi_group_wait_all() had already been called for the same group and the first wait call is still pending.
`MTAPI_ERR_PARAMETER`	Invalid timeout parameter.
`MTAPI_ERR_ARG_SIZE`	The size of the arguments expected by action differs from arguments size of the caller.
`MTAPI_ERR_RESULT_SIZE`	The size of the result buffer expected by action differs from result buffer size of the caller.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_GROUP_COMPLETED`	Group completed, i.e., there are no more task to wait for in the group.
`MTAPI_ERR_TASK_CANCELLED`	At least one task has been canceled because of mtapi_task_cancel() was called before the task was executed or the error code was set to `MTAPI_ERR_TASK_CANCELLED` by mtapi_context_status_set() in the action function.
`MTAPI_ERR_ACTION_CANCELLED`	The action execution of at least one task was canceled by the action function (mtapi_context_status_set()).
`MTAPI_ERR_ACTION_FAILED`	Error set by at least one action function (mtapi_context_status_set()).
`MTAPI_ERR_ACTION_DELETED`	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action function to `MTAPI_ERR_ACTION_DELETED` by mtapi_context_status_set().

See also

mtapi_group_wait_any(), mtapi_context_status_set(), mtapi_task_cancel()

Concurrency

Thread-safe

Parameters

in	*group*	Group handle
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41.2.5 void mtapi_group_wait_any (const mtapi_group_hndl_t *group*, void ** *result*, const mtapi_timeout_t *timeout*, mtapi_status_t * *status*)

This function waits for the completion of any task in a task group.

Tasks may be associated with groups when the tasks are started. Each task is associated with one or more actions. This function returns when any of the associated action functions have completed or have been canceled.

The group handle does not become invalid if this function returns MTAPI_SUCCESS. The group handle becomes invalid if this function returns MTAPI_GROUP_COMPLETED.

group must be a valid group handle obtained by a previous call to mtapi_group_create().

Action functions may pass results that will be available in *result after mtapi_group_wait_any() returns. If the results are not needed, result may be set to MTAPI_NULL. Otherwise, result must point to an area in memory of sufficient size to hold the array of results from the completed task(s). The size of the result buffer is given in the argument result_buffer_size that the runtime passes to an action function upon invocation.

timeout determines how long the function should wait for a task in the group to finish. The underlying type of mtapi_timeout_t is implementation-defined. If timeout is a constant 0 or the symbolic constant MTAPI_NOWAIT, this function does not block and returns immediately. If timeout is set to MTAPI_INFINITE the function may block infinitely. Other values for timeout and the units of measure are implementation defined.

During execution, an action function may optionally call mtapi_context_status_set() to set a task status that will be returned in this function in *status. The following task status values may be set by an action function: MTAPI_ERR_TASK_CANCELLED, MTAPI_ERR_ACTION_CANCELLED, MTAPI_ERR_ACTION_FAILED, and MTAPI_ERR_ACTION_DELETED.

On success, *status is either set to MTAPI_SUCCESS if one of the tasks in the group completed or to MTAPI_GROUP_COMPLETED if all tasks of the group have completed and successfully waited for. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_TIMEOUT	Timeout was reached.
MTAPI_ERR_GROUP_INVALID	Argument is not a valid task handle.
MTAPI_ERR_PARAMETER	Invalid timeout parameter.
MTAPI_ERR_ARG_SIZE	The size of the arguments expected by action differs from arguments size of the caller.
MTAPI_ERR_RESULT_SIZE	The size of the result buffer expected by action differs from result buffer size of the caller.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_GROUP_COMPLETED	Group completed, i.e., there are no more tasks to wait for in the group.

Error code	Description
MTAPI_ERR_TASK_CANCELLED	The task has been canceled because mtapi_task_cancel() was called before the task was executed, or the error code was set to MTAPI_ERR_TASK_CANCELLED by mtapi_context_status_set() in the action code.
MTAPI_ERR_ACTION_CANCELLED	Action execution was canceled by the action function (mtapi_context_status_set()).
MTAPI_ERR_ACTION_FAILED	Error set by action function (mtapi_context_status_set()).
MTAPI_ERR_ACTION_DELETED	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action code to MTAPI_ERR_ACTION_DELETED by mtapi_context_status_set().

See also

mtapi_group_create(), mtapi_context_status_set(), mtapi_task_cancel()

Concurrency

Thread-safe

Parameters

in	*group*	Group handle
out	*result*	Pointer to result buffer supplied at task start
in	*timeout*	Timeout duration in milliseconds
out	*status*	Pointer to error code, may be MTAPI_NULL

5.41.2.6 void mtapi_group_delete (const mtapi_group_hndl_t *group*, mtapi_status_t * *status*)

This function deletes a task group.

Deleting a group does not have any influence on tasks belonging to the group. Adding tasks to a group that is already deleted will result in an MTAPI_ERR_GROUP_INVALID error.

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_GROUP_INVALID	Argument is not a valid group handle.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

Concurrency

Thread-safe

Parameters

in	*group*	Group handle
out	*status*	Pointer to error code, may be MTAPI_NULL

5.42 MTAPI Extensions

Provides extensions to the standard MTAPI API.

Modules

- MTAPI OpenCL Plugin

  Provides functionality to execute tasks on OpenCL devices.
- MTAPI Network Plugin

  Provides functionality to distribute tasks across nodes in a TCP/IP network.
- MTAPI CUDA Plugin

  Provides functionality to execute tasks on CUDA devices.

Typedefs

- typedef void(∗ mtapi_ext_plugin_task_start_function_t) (MTAPI_IN mtapi_task_hndl_t task, MTAPI_OUT mtapi_status_t ∗status)

  Represents a callback function that is called when a plugin action is about to start a plugin task.
- typedef void(∗ mtapi_ext_plugin_task_cancel_function_t) (MTAPI_IN mtapi_task_hndl_t task, MTAPI_OUT mtapi_status_t ∗status)

  Represents a callback function that is called when a plugin task is about to be canceled.
- typedef void(∗ mtapi_ext_plugin_action_finalize_function_t) (MTAPI_IN mtapi_action_hndl_t action, MTAPI_OUT mtapi_status_t ∗status)

  Represents a callback function that is called when a plugin action is about to be finalized.

Functions

- mtapi_action_hndl_t mtapi_ext_plugin_action_create (MTAPI_IN mtapi_job_id_t job_id, MTAPI_IN mtapi_ext_plugin_task_start_function_t task_start_function, MTAPI_IN mtapi_ext_plugin_task_cancel_function_t task_cancel_function, MTAPI_IN mtapi_ext_plugin_action_finalize_function_t action_finalize_function, MTAPI_IN void ∗plugin_data, MTAPI_IN void ∗node_local_data, MTAPI_IN mtapi_size_t node_local_data_size, MTAPI_IN mtapi_action_attributes_t ∗attributes, MTAPI_OUT mtapi_status_t ∗status)

  This function creates a plugin action.
- void mtapi_ext_yield ()

  This function yields execution to the MTAPI scheduler for at most one task.

5.42.1 Detailed Description

Provides extensions to the standard MTAPI API.

There are two extension functions defined here. One to support user defined behavior of an action to allow for actions that are not implemented locally in software, but e.g., on a remote node in a network or on an accelerator device like a GPU or an FPGA. The other one is used to specify job attributes.

5.42.2 Typedef Documentation

5.42.2.1 typedef void(* mtapi_ext_plugin_task_start_function_t) (MTAPI_IN **mtapi_task_hndl_t** task, MTAPI_OUT mtapi_status_t *status)

Represents a callback function that is called when a plugin action is about to start a plugin task.

This function should return MTAPI_SUCCESS if the task could be started and the appropriate MTAPI_ERR_* if not.

5.42.2.2 typedef void(* mtapi_ext_plugin_task_cancel_function_t) (MTAPI_IN **mtapi_task_hndl_t** task, MTAPI_OUT mtapi_status_t *status)

Represents a callback function that is called when a plugin task is about to be canceled.

This function should return MTAPI_SUCCESS if the task could be canceled and the appropriate MTAPI_ERR_* if not.

5.42.2.3 typedef void(* mtapi_ext_plugin_action_finalize_function_t) (MTAPI_IN **mtapi_action_hndl_t** action, MTAPI_OUT mtapi_status_t *status)

Represents a callback function that is called when a plugin action is about to be finalized.

This function should return MTAPI_SUCCESS if the action could be deleted and the appropriate MTAPI_ERR_* if not.

5.42.3 Function Documentation

5.42.3.1 **mtapi_action_hndl_t** mtapi_ext_plugin_action_create (MTAPI_IN mtapi_job_id_t *job_id,* MTAPI_IN **mtapi_ext_plugin_task_start_function_t** *task_start_function,* MTAPI_IN **mtapi_ext_plugin_task_cancel_function_t** *task_cancel_function,* MTAPI_IN **mtapi_ext_plugin_action_finalize_function_t** *action_finalize_function,* MTAPI_IN void * *plugin_data,* MTAPI_IN void * *node_local_data,* MTAPI_IN mtapi_size_t *node_local_data_size,* MTAPI_IN **mtapi_action_attributes_t** * *attributes,* MTAPI_OUT mtapi_status_t * *status*)

This function creates a plugin action.

It is called on the node where the plugin action is implemented. A plugin action is an abstract encapsulation of a user defined action that is needed to implement a job that does not represent a software action. A plugin action contains a reference to a job, callback functions to start and cancel tasks and a reference to an callback function to finalize the action. After a plugin action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`. A plugin action's life-cycle begins with mtapi_ext_plugin_action_create(), and ends when mtapi_action_delete() or mtapi_finalize() is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

If `node_local_data_size` is not zero, `node_local_data` specifies the start of node local data shared by action functions executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

On success, an action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the action already exists, `status` will be set to `MTAPI_ERR_ACTION_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The `job_id` is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
`MTAPI_ERR_ACTION_EXISTS`	This action is already created.
`MTAPI_ERR_ACTION_LIMIT`	Exceeded maximum number of actions allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_action_delete(), mtapi_finalize()

Returns

Handle to newly created plugin action, invalid handle on error

Concurrency

Thread-safe

Parameters

`in`	*job_id*	Job id
`in`	*task_start_function*	Task start function
`in`	*task_cancel_function*	Task cancel function
`in`	*action_finalize_function*	Finalize action function
`in`	*plugin_data*	Pointer to plugin data
`in`	*node_local_data*	Pointer to node local data
`in`	*node_local_data_size*	Size of node local data
`out`	*attributes*	Pointer to attributes
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.42.3.2 void mtapi_ext_yield ( )

This function yields execution to the MTAPI scheduler for at most one task.

Concurrency

Not thread-safe

5.43 Atomic

Atomic operations.

Functions

- void embb_atomic_init_TYPE (emb_atomic_TYPE ∗variable, TYPE initial_value)

  Initializes an atomic variable.
- void embb_atomic_destroy_TYPE (emb_atomic_TYPE ∗variable)

  Destroys an atomic variable and frees its resources.
- void embb_atomic_and_assign_TYPE (embb_atomic_TYPE ∗variable, TYPE value)

  Computes the logical "and" of the value stored in `variable` *and* `value`*.*
- int embb_atomic_compare_and_swap_TYPE (embb_atomic_TYPE ∗variable, TYPE ∗expected, TYPE desired)

  Compares `variable` *with* `expected` *and, if equivalent, swaps its value with* `desired`*.*
- TYPE embb_atomic_fetch_and_add_TYPE (embb_atomic_TYPE ∗variable, TYPE value)

  Adds `value` *to* `variable` *and returns its old value.*
- TYPE embb_atomic_load_TYPE (const embb_atomic_TYPE ∗variable)

  Loads the value of `variable` *and returns it.*
- void embb_atomic_memory_barrier ()

  Enforces a memory barrier (full fence).
- void embb_atomic_or_assign_TYPE (embb_atomic_TYPE ∗variable, TYPE value)

  Computes the logical "or" of the value stored in `variable` *and* `value`*.*
- void embb_atomic_store_TYPE (embb_atomic_TYPE ∗variable, int value)

  Stores `value` *in* `variable`*.*
- TYPE embb_atomic_swap_TYPE (embb_atomic_TYPE ∗variable, TYPE value)

  Swaps the current value of `variable` *with* `value`*.*
- void embb_atomic_xor_assign_TYPE (embb_atomic_TYPE ∗variable, TYPE value)

  Computes the logical "xor" of the value stored in `variable` *and* `value`*.*

5.43.1 Detailed Description

Atomic operations.

Atomic operations are not directly applied to fundamental types. Instead, there is for each character and integer type an associated atomic type that has the same bit width (if the target CPU supports atomic operations on that type):

Fundamental type	Atomic type
char	embb_atomic_char
short	embb_atomic_short
unsigned short	embb_atomic_unsigned_short
int	embb_atomic_int
unsigned int	embb_atomic_unsigned_int
long	embb_atomic_long
unsigned long	embb_atomic_unsigned_long
long long	embb_atomic_long_long
unsigned long long	embb_atomic_unsigned_long_long
intptr_t	embb_atomic_intptr_t
uintptr_t	embb_atomic_uintptr_t
size_t	embb_atomic_size_t
ptrdiff_t	embb_atomic_ptrdiff_t
uintmax_t	embb_atomic_uintmax_t

Each of the atomic operations described in the following can be applied to the types listed above. However, to avoid unnecessary redundancy, we document them only once in a generic way. The keyword TYPE serves as a placeholder which has to be replaced by the concrete type (e.g., int). If the fundamental type contains spaces (e.g., unsigned int), "_" is used for concatenation (e.g. unsigned_int).

Usage example:

Store the value `5` in an atomic `"unsigned int"` variable.

Step 1 (declare atomic variable):

```
embb_atomic_unsigned_int my_var;
```

Step 2 (store the value):

```
embb_atomic_store_unsigned_int( &my_var, 5 );
```

The current implementation guarantees sequential consistency (full fences) for all atomic operations. Relaxed memory models may be added in the future.

5.43.2 Function Documentation

5.43.2.1 void embb_atomic_init_TYPE (emb_atomic_TYPE * *variable*, TYPE *initial_value*)

Initializes an atomic variable.

Precondition

The atomic variable has not been initialized.

Postcondition

The atomic variable has the value `initial_value` and can be used in atomic operations.

Concurrency

Not thread-safe

Parameters

`in,out`	*variable*	Pointer to atomic variable to be initialized.
`in`	*initial_value*	Initial value to be assigned to atomic variable.

5.43.2.2 void embb_atomic_destroy_TYPE (emb_atomic_TYPE * *variable*)

Destroys an atomic variable and frees its resources.

Precondition

The atomic variable has been initialized.

Postcondition

The atomic variable is uninitialized.

Concurrency

Not thread-safe

Parameters

`in,out`	*variable*	Pointer to atomic variable to be destroyed.

5.43.2.3 void embb_atomic_and_assign_TYPE (embb_atomic_TYPE * *variable,* TYPE *value*)

Computes the logical "and" of the value stored in `variable` and `value`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Postcondition

The result is stored in `variable`.

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable which serves as left-hand side for the "and" operation and is used to store the result.
`in`	*value*	Right-hand side of "and" operation, passed by value

5.43.2.4 int embb_atomic_compare_and_swap_TYPE (embb_atomic_TYPE * *variable,* TYPE * *expected,* TYPE *desired*)

Compares `variable` with `expected` and, if equivalent, swaps its value with `desired`.

Stores `desired` in `variable` if the value of `variable` is equivalent to the value of `expected`. Otherwise, stores the value of `variable` in `expected`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Returns

!= 0 if the values of `variable` and `expected` were equivalent
0 otherwise

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable
`in,out`	*expected*	Pointer to expected value
`in`	*desired*	Value to be stored in `variable`

5.43.2.5 TYPE embb_atomic_fetch_and_add_TYPE (embb_atomic_TYPE * *variable*, TYPE *value*)

Adds `value` to `variable` and returns its old value.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Returns

The value before the operation

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable
`in`	*value*	The value to be added to `variable` (can be negative)

5.43.2.6 TYPE embb_atomic_load_TYPE (const embb_atomic_TYPE * *variable*)

Loads the value of `variable` and returns it.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Returns

The value of the atomic variable.

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in`	*variable*	Pointer to atomic variable

5.43.2.7 void embb_atomic_memory_barrier ( )

Enforces a memory barrier (full fence).

Concurrency

Thread-safe and wait-free

5.43.2.8 void embb_atomic_or_assign_TYPE (embb_atomic_TYPE * *variable*, TYPE *value*)

Computes the logical "or" of the value stored in `variable` and `value`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Postcondition

The result is stored in `variable`.

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable which serves as left-hand side for the "or" operation and is used to store the result.
`in`	*value*	Right-hand side of "or" operation, passed by value

5.43.2.9 void embb_atomic_store_TYPE (embb_atomic_TYPE * *variable*, int *value*)

Stores `value` in `variable`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable
`in`	*value*	Value to be stored

5.43.2.10 TYPE embb_atomic_swap_TYPE (embb_atomic_TYPE * *variable*, TYPE *value*)

Swaps the current value of `variable` with `value`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Returns

The old value of `variable`

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable whose value is swapped
`in`	*value*	Value which will be stored in the atomic variable

5.43.2.11 void embb_atomic_xor_assign_TYPE (embb_atomic_TYPE * *variable,* TYPE *value*)

Computes the logical "xor" of the value stored in `variable` and `value`.

Precondition

The atomic variable has been initialized with `embb_atomic_init_TYPE`

Postcondition

The result is stored in `variable`.

See also

Detailed description for general information and the meaning of **TYPE**.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*variable*	Pointer to atomic variable which serves as left-hand side for the "xor" operation and is used to store the result.
`in`	*value*	Right-hand side of "xor" operation, passed by value

5.44 C Components

Components written in C.

Modules

- MTAPI

    Multicore Task Management API (MTAPI®).
- Base

    Platform-independent abstraction layer for multithreading and basic operations.

5.44.1 Detailed Description

Components written in C.

5.45 Base

Platform-independent abstraction layer for multithreading and basic operations.

Modules

- Atomic

  Atomic operations.
- Condition Variable

  Condition variables for thread synchronization.
- Core Set

  Core sets for thread-to-core affinities.
- Counter

  Thread-safe counter.
- Duration and Time

  Relative time durations and absolute time points.
- Error

  Error codes for function return values.
- Logging

  Simple logging facilities.
- Memory Allocation

  Functions for dynamic memory allocation.
- Mutex

  Mutexes for thread synchronization.
- Thread

  Threads supporting thread-to-core affinities.
- Thread-Specific Storage

  Thread-specific storage.

5.45.1 Detailed Description

Platform-independent abstraction layer for multithreading and basic operations.

This component provides basic functionalities, mainly for creating and synchronizing threads. Most of the functions are essentially wrappers for functions specific to the underlying operating system.

5.46 Condition Variable

Condition variables for thread synchronization.

Typedefs

- typedef opaque_type embb_condition_t

  Opaque type representing a condition variable.

Functions

- int embb_condition_init (embb_condition_t *condition_var)

  Initializes a condition variable.
- int embb_condition_notify_one (embb_condition_t *condition_var)

  Wakes up one thread waiting for `condition_var`.
- int embb_condition_notify_all (embb_condition_t *condition_var)

  Wakes up all threads waiting for `condition_var`.
- int embb_condition_wait (embb_condition_t *condition_var, embb_mutex_t *mutex)

  Unlocks `mutex` and waits until the thread is woken up.
- int embb_condition_wait_until (embb_condition_t *condition_var, embb_mutex_t *mutex, const embb_time_t *time)

  Unlocks `mutex` and waits until the thread is woken up or `time` has passed.
- int embb_condition_wait_for (embb_condition_t *condition_var, embb_mutex_t *mutex, const embb_duration_t *duration)

  Unlocks `mutex` and waits until the thread is woken up or `duration` has passed.
- int embb_condition_destroy (embb_condition_t *condition_var)

  Destroys `condition_var` and frees used memory.

5.46.1 Detailed Description

Condition variables for thread synchronization.

Provides an abstraction from platform-specific condition variable implementations. Condition variables can be waited for with timeouts using relative durations and absolute time points.

5.46.2 Typedef Documentation

5.46.2.1 typedef opaque_type embb_condition_t

Opaque type representing a condition variable.

5.46.3 Function Documentation

5.46.3.1 int embb_condition_init (embb_condition_t * condition_var)

Initializes a condition variable.

Dynamic memory allocation

Potentially allocates dynamic memory

Precondition

`condition_var` is not initialized

Postcondition

If successful, `condition_var` is initialized

Returns

EMBB_SUCCESS if successful
EMBB_ERROR otherwise

Concurrency

Not thread-safe

See also

embb_condition_destroy()

Parameters

out	*condition_var*	Pointer to condition variable

5.46.3.2 int embb_condition_notify_one (embb_condition_t * condition_var)

Wakes up one thread waiting for `condition_var`.

Precondition

`condition_var` is initialized

Returns

EMBB_SUCCESS if signaling was successful
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_condition_notify_all(), embb_condition_wait(), embb_condition_wait_until(), embb_condition_wait_for()

Parameters

`in,out`	*condition_var*	Pointer to condition variable

5.46.3.3 int embb_condition_notify_all (**embb_condition_t** ∗ *condition_var*)

Wakes up all threads waiting for `condition_var`.

Precondition

`condition_var` is initialized

Returns

EMBB_SUCCESS if broadcast was successful
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_condition_notify_one(), embb_condition_wait(), embb_condition_wait_until(), embb_condition_wait_for()

Parameters

`in,out`	*condition_var*	Pointer to condition variable

5.46.3.4 int embb_condition_wait (**embb_condition_t** ∗ *condition_var,* **embb_mutex_t** ∗ *mutex*)

Unlocks `mutex` and waits until the thread is woken up.

Precondition

`condition_var` is initialized and `mutex` is locked by calling thread

Postcondition

If successful, `mutex` is locked by the calling thread

Returns

EMBB_SUCCESS if successful
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_condition_notify_one(), embb_condition_notify_all(), embb_condition_wait_until(), embb_condition_↩
wait_for()

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*condition_var*	Pointer to condition variable
`in,out`	*mutex*	Pointer to mutex

5.46.3.5 int embb_condition_wait_until (**embb_condition_t** ∗ *condition_var,* **embb_mutex_t** ∗ *mutex,* const **embb_time_t** ∗ *time*)

Unlocks `mutex` and waits until the thread is woken up or `time` has passed.

Precondition

`condition_var` is initialized and `mutex` is locked by calling thread

Postcondition

If successful, `mutex` is locked by the calling thread

Returns

EMBB_SUCCESS if successful
EMBB_TIMEDOUT if mutex could not be locked until the specified point of time
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_condition_notify_one(), embb_condition_notify_all(), embb_condition_wait(), embb_condition_wait_↩
for()

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*condition_var*	Pointer to condition variable
`in,out`	*mutex*	Pointer to mutex
`in`	*time*	Point of time until the thread waits

5.46.3.6 int embb_condition_wait_for (**embb_condition_t** ∗ *condition_var,* **embb_mutex_t** ∗ *mutex,* const **embb_duration_t** ∗ *duration*)

Unlocks `mutex` and waits until the thread is woken up or `duration` has passed.

Precondition

`condition_var` is initialized and `mutex` is locked by calling thread

Postcondition

If successful, `mutex` is locked by the calling thread

Returns

EMBB_SUCCESS if successful
EMBB_TIMEDOUT if mutex could not be locked within the specified time span
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_condition_notify_one(), embb_condition_notify_all(), embb_condition_wait(), embb_condition_wait_until()

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*condition_var*	Pointer to condition variable
`in,out`	*mutex*	Pointer to mutex
`in`	*duration*	Duration in microseconds the thread waits

5.46.3.7 int embb_condition_destroy (**embb_condition_t** ∗ *condition_var*)

Destroys `condition_var` and frees used memory.

Precondition

`condition_var` is initialized and no thread is waiting for it using embb_condition_wait(), embb_condition↩_wait_for(), or embb_condition_wait_until().

Postcondition

`condition_var` is uninitialized

Returns

EMBB_SUCCESS if destruction of condition variable was successful
EMBB_ERROR otherwise

Concurrency

Not thread-safe

See also

embb_condition_init()

Parameters

in	*condition_var*	Pointer to condition variable

5.47 Core Set

Core sets for thread-to-core affinities.

Typedefs

- typedef opaque_type embb_core_set_t

  Opaque type representing a set of processor cores.

Functions

- unsigned int embb_core_count_available ()

  Returns the number of available processor cores.
- void embb_core_set_init (embb_core_set_t ∗core_set, int initializer)

  Initializes the specified core set.
- void embb_core_set_add (embb_core_set_t ∗core_set, unsigned int core_number)

  Adds a core to the specified set.
- void embb_core_set_remove (embb_core_set_t ∗core_set, unsigned int core_number)

  Removes a core from the specified set.
- int embb_core_set_contains (const embb_core_set_t ∗core_set, unsigned int core_number)

  Determines whether a core is contained in the specified set.
- void embb_core_set_intersection (embb_core_set_t ∗set1, const embb_core_set_t ∗set2)

  Computes the intersection of core `set1` and `set2`.
- void embb_core_set_union (embb_core_set_t ∗set1, const embb_core_set_t ∗set2)

  Computes the union of core `set1` and `set2`.
- unsigned int embb_core_set_count (const embb_core_set_t ∗core_set)

  Returns the number of cores contained in the specified set.

5.47.1 Detailed Description

Core sets for thread-to-core affinities.

5.47.2 Typedef Documentation

5.47.2.1 typedef opaque_type **embb_core_set_t**

Opaque type representing a set of processor cores.

An instance of this type represents a subset of processor cores. Core sets can be used to set thread-to-core affinities. A core in a core set might just represent a logical core (hyper-thread), depending on the underlying hardware. Each core is identified by a unique integer starting with 0. For example, the cores of a quad-core system are represented by the set {0,1,2,3}.

See also

embb_core_count_available()

5.47.3 Function Documentation

5.47.3.1 unsigned int embb_core_count_available ( )

Returns the number of available processor cores.

If the processor supports hyper-threading, each hyper-thread is treated as a separate processor core.

Returns

Number of cores including hyper-threads

Concurrency

Not thread-safe

5.47.3.2 void embb_core_set_init (embb_core_set_t * *core_set*, int *initializer*)

Initializes the specified core set.

The second parameter specifies whether the set is initially empty or contains all cores.

Precondition

`core_set` is not NULL.

Concurrency

Not thread-safe

Parameters

`out`	*core_set*	Core set to initialize
`in`	*initializer*	The set is initially empty if initializer == 0, otherwise it contains all available processor cores.

5.47.3.3 void embb_core_set_add (embb_core_set_t * *core_set*, unsigned int *core_number*)

Adds a core to the specified set.

If the core is already contained in the set, the operation has no effect.

Precondition

`core_set` is not NULL and `core_number` is smaller than embb_core_count_available().

Concurrency

Not thread-safe

See also

embb_core_set_remove()

Parameters

`in,out`	*core_set*	Core set to be manipulated
`in`	*core_number*	Number of core to be added.

5.47.3.4 void embb_core_set_remove (embb_core_set_t * *core_set*, unsigned int *core_number*)

Removes a core from the specified set.

If the core is not in the set, the operation has no effect.

Precondition

`core_set` is not NULL and `core_number` is smaller than embb_core_count_available().

Concurrency

Not thread-safe

See also

embb_core_set_add()

Parameters

`in,out`	*core_set*	Core set to be manipulated
`in`	*core_number*	Number of core to be removed

5.47.3.5 int embb_core_set_contains (const embb_core_set_t * *core_set*, unsigned int *core_number*)

Determines whether a core is contained in the specified set.

Precondition

`core_set` is not NULL and `core_number` is smaller than embb_core_count_available().

Returns

0 if the core is not contained in the set, otherwise a number greater than zero.

Concurrency

Not thread-safe

Parameters

`in`	*core_set*	Core set
`in`	*core_number*	Number of core

5.47.3.6 void embb_core_set_intersection (embb_core_set_t * set1, const embb_core_set_t * set2)

Computes the intersection of core `set1` and `set2`.

The result is stored in `set1`.

Precondition

`set1` and `set2` are not NULL.

Concurrency

Not thread-safe

See also

embb_core_set_union()

Parameters

`in,out`	*set1*	First set, gets overwritten by the result
`in`	*set2*	Second set

5.47.3.7 void embb_core_set_union (embb_core_set_t * set1, const embb_core_set_t * set2)

Computes the union of core `set1` and `set2`.

The result is stored in `set1`.

Precondition

`set1` and `set2` are not NULL.

Concurrency

Not thread-safe

See also

embb_core_set_intersection()

Parameters

`in,out`	*set1*	First set
`in`	*set2*	Second set

5.47.3.8 unsigned int embb_core_set_count (const embb_core_set_t * core_set)

Returns the number of cores contained in the specified set.

Precondition

`core_set` is not NULL.

Concurrency

Not thread-safe

Returns

Number of cores in `core_set`

Parameters

in	*core_set*	Core set whose elements are counted

5.48 Counter

Thread-safe counter.

Typedefs

- typedef opaque_type embb_counter_t
  Opaque type representing a thread-safe counter.

Functions

- int embb_counter_init (embb_counter_t *counter)
  Initializes `counter` and sets it to zero.
- unsigned int embb_counter_get (embb_counter_t *counter)
  Returns the current value of `counter`.
- unsigned int embb_counter_increment (embb_counter_t *counter)
  Increments `counter` and returns the old value.
- unsigned int embb_counter_decrement (embb_counter_t *counter)
  Decrements `counter` and returns the old value.
- void embb_counter_reset (embb_counter_t *counter)
  Resets an initialized counter to 0.
- void embb_counter_destroy (embb_counter_t *counter)
  Destroys an initialized counter.

5.48.1 Detailed Description

Thread-safe counter.

5.48.2 Typedef Documentation

5.48.2.1 typedef opaque_type **embb_counter_t**

Opaque type representing a thread-safe counter.

5.48.3 Function Documentation

5.48.3.1 int embb_counter_init (**embb_counter_t** * *counter*)

Initializes `counter` and sets it to zero.

Returns

EMBB_SUCCESS if counter could be initialized
EMBB_ERROR otherwise

Concurrency

Thread-safe and wait-free

Parameters

`out`	*counter*	Pointer to counter

5.48.3.2 unsigned int embb_counter_get (embb_counter_t ∗ *counter*)

Returns the current value of `counter`.

Precondition

`counter` is not NULL.

Returns

Current value

Concurrency

Thread-safe and wait-free

Parameters

`in`	*counter*	Pointer to counter

5.48.3.3 unsigned int embb_counter_increment (embb_counter_t ∗ *counter*)

Increments `counter` and returns the old value.

Precondition

`counter` is not NULL.

Returns

Old, non-incremented value

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*counter*	Pointer to counter

5.48.3.4 unsigned int embb_counter_decrement (embb_counter_t * *counter*)

Decrements `counter` and returns the old value.

Precondition

`counter` is not NULL.

Returns

Old, non-decremented value

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*counter*	Pointer to counter

5.48.3.5 void embb_counter_reset (embb_counter_t * *counter*)

Resets an initialized counter to 0.

Precondition

`counter` is initialized and not NULL.

Concurrency

Thread-safe and wait-free

Parameters

`in,out`	*counter*	Pointer to counter

5.48.3.6 void embb_counter_destroy (embb_counter_t * *counter*)

Destroys an initialized counter.

Precondition

`counter` is initialized and not NULL.

Postcondition

`counter` is invalid and cannot be used anymore

Concurrency

Thread-safe and wait-free

Parameters

out	*counter*	Pointer to counter

5.49 Duration and Time

Relative time durations and absolute time points.

Duration

- typedef opaque_type embb_duration_t

  Opaque type representing a relative time duration.
- const embb_duration_t ∗ embb_duration_max ()

  Returns duration with maximum ticks representable by implementation.
- const embb_duration_t ∗ embb_duration_min ()

  Returns duration with minimum ticks representable by implementation.
- const embb_duration_t ∗ embb_duration_zero ()

  Returns duration of length zero.
- int embb_duration_set_nanoseconds (embb_duration_t ∗duration, unsigned long long nanoseconds)

  Set duration from nanosecond ticks.
- int embb_duration_set_microseconds (embb_duration_t ∗duration, unsigned long long microseconds)

  Sets duration from microsecond ticks.
- int embb_duration_set_milliseconds (embb_duration_t ∗duration, unsigned long long milliseconds)

  Sets duration from millisecond ticks.
- int embb_duration_set_seconds (embb_duration_t ∗duration, unsigned long long seconds)

  Sets duration from second ticks.
- int embb_duration_add (embb_duration_t ∗lhs, const embb_duration_t ∗rhs)

  Adds two durations.
- int embb_duration_as_nanoseconds (const embb_duration_t ∗duration, unsigned long long ∗nanoseconds)

  Converts duration to nanosecond ticks.
- int embb_duration_as_microseconds (const embb_duration_t ∗duration, unsigned long long ∗microseconds)

  Converts duration to microsecond ticks.
- int embb_duration_as_milliseconds (const embb_duration_t ∗duration, unsigned long long ∗milliseconds)

  Converts duration to millisecond ticks.
- int embb_duration_as_seconds (const embb_duration_t ∗duration, unsigned long long ∗seconds)

  Converts duration to second ticks.
- int embb_duration_compare (const embb_duration_t ∗lhs, const embb_duration_t ∗rhs)

  Compares two durations.
- #define EMBB_DURATION_INIT

  Macro for initializing a duration with zero length at definition.

Time

- typedef opaque_type embb_time_t

  Opaque type representing an absolute time point.
- void embb_time_now (embb_time_t ∗time)

  Sets time point to now.
- int embb_time_in (embb_time_t ∗time, const embb_duration_t ∗duration)

  Sets time point to now plus the given duration.
- int embb_time_compare (const embb_time_t ∗lhs, const embb_time_t ∗rhs)

  Compares two time points.

5.49.1 Detailed Description

Relative time durations and absolute time points.

5.49.2 Macro Definition Documentation

5.49.2.1 #define EMBB_DURATION_INIT

Macro for initializing a duration with zero length at definition.

5.49.3 Typedef Documentation

5.49.3.1 typedef opaque_type **embb_duration_t**

Opaque type representing a relative time duration.

5.49.3.2 typedef opaque_type **embb_time_t**

Opaque type representing an absolute time point.

5.49.4 Function Documentation

5.49.4.1 const **embb_duration_t**∗ embb_duration_max ( )

Returns duration with maximum ticks representable by implementation.

Returns

Pointer to duration with maximum value

Concurrency

Not thread-safe

See also

embb_duration_min()

5.49.4.2 const embb_duration_t* embb_duration_min ( )

Returns duration with minimum ticks representable by implementation.

Returns

Pointer to duration with minimum value

Concurrency

Not thread-safe

See also

embb_duration_max()

5.49.4.3 const embb_duration_t* embb_duration_zero ( )

Returns duration of length zero.

Returns

Pointer to duration of length zero

Concurrency

Not thread-safe

5.49.4.4 int embb_duration_set_nanoseconds (embb_duration_t * *duration*, unsigned long long *nanoseconds*)

Set duration from nanosecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if given nanosecond interval is too small to be represented by implementation
EMBB_OVERFLOW if given nanosecond interval is too large to be represented by implementation

Concurrency

Not thread-safe

Parameters

out	*duration*	Pointer to duration
in	*nanoseconds*	Nanosecond ticks

5.49.4.5 int embb_duration_set_microseconds (embb_duration_t * *duration*, unsigned long long *microseconds*)

Sets duration from microsecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if given microsecond interval is too small to be represented by implementation
EMBB_OVERFLOW if given microsecond interval is too large to be represented by implementation

Concurrency

Not thread-safe

Parameters

`out`	*duration*	Pointer to duration
`in`	*microseconds*	Microsecond ticks

5.49.4.6 int embb_duration_set_milliseconds (embb_duration_t * *duration*, unsigned long long *milliseconds*)

Sets duration from millisecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if given millisecond interval is too small to be represented by implementation
EMBB_OVERFLOW if given millisecond interval is too large to be represented by implementation

Concurrency

Not thread-safe

Parameters

`out`	*duration*	Pointer to duration
`in`	*milliseconds*	Millisecond ticks

5.49.4.7 int embb_duration_set_seconds (embb_duration_t * *duration*, unsigned long long *seconds*)

Sets duration from second ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if given second interval is too small to be represented by implementation
EMBB_OVERFLOW if given second interval is too large to be represented by implementation

Concurrency

Not thread-safe

Parameters

out	*duration*	Pointer to duration
in	*seconds*	Second ticks

5.49.4.8 int embb_duration_add (embb_duration_t * lhs, const embb_duration_t * rhs)

Adds two durations.

Computest the sum of `rhs` and `lhs` and stores the result in `lhs`.

Returns

EMBB_SUCCESS
EMBB_OVERFLOW if sum is greater than embb_duration_max()

Concurrency

Not thread-safe

Parameters

in,out	*lhs*	Left-hand side operand, overwritten by result of addition
in	*rhs*	Right-hand side operand of addition

5.49.4.9 int embb_duration_as_nanoseconds (const embb_duration_t * duration, unsigned long long * nanoseconds)

Converts duration to nanosecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if duration contains fractions less than a nanosecond
EMBB_OVERFLOW if duration is not representable by tick type

Concurrency

Not thread-safe

Parameters

in	*duration*	Pointer to duration
out	*nanoseconds*	Pointer to nanosecond ticks of duration

5.49.4.10 int embb_duration_as_microseconds (const embb_duration_t * duration, unsigned long long * microseconds)

Converts duration to microsecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if duration contains fractions less than a microsecond
EMBB_OVERFLOW if duration is not representable by tick type

Concurrency

Not thread-safe

Parameters

in	*duration*	Pointer to duration
out	*microseconds*	Pointer to microsecond ticks of duration

5.49.4.11 int embb_duration_as_milliseconds (const embb_duration_t ∗ *duration,* unsigned long long ∗ *milliseconds*)

Converts duration to millisecond ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if duration contains fractions less than a millisecond
EMBB_OVERFLOW if duration is not representable by tick type

Concurrency

Not thread-safe

Parameters

in	*duration*	Pointer to duration
out	*milliseconds*	Pointer to millisecond ticks of duration

5.49.4.12 int embb_duration_as_seconds (const embb_duration_t ∗ *duration,* unsigned long long ∗ *seconds*)

Converts duration to second ticks.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if duration contains fractions less than a second
EMBB_OVERFLOW if duration is not representable by tick type

Concurrency

Not thread-safe

Parameters

in	*duration*	Pointer to duration
out	*seconds*	Pointer to second ticks of duration

5.49.4.13 int embb_duration_compare (const embb_duration_t * *lhs*, const embb_duration_t * *rhs*)

Compares two durations.

Precondition

`lhs` and `rhs` are not NULL and properly initialized.

Returns

-1 if `lhs` < `rhs`
0 if `lhs` == `rhs`
1 if `lhs` > `rhs`

Concurrency

Not thread-safe

Parameters

in	*lhs*	Pointer to left-hand side operand
in	*rhs*	Pointer to right-hand side operand

5.49.4.14 void embb_time_now (embb_time_t * *time*)

Sets time point to now.

Concurrency

Not thread-safe

See also

embb_time_in()

Parameters

out	*time*	Pointer to time point

5.49.4.15 int embb_time_in (**embb_time_t** * *time*, const **embb_duration_t** * *duration*)

Sets time point to now plus the given duration.

Returns

EMBB_SUCCESS
EMBB_UNDERFLOW if duration is smaller than implementation allows
EMBB_OVERFLOW if time + duration is larger than implementation allows

Concurrency

Not thread-safe

See also

embb_time_now()

Parameters

`out`	*time*	Pointer to time point
`in`	*duration*	Pointer to duration

5.49.4.16 int embb_time_compare (const **embb_time_t** * *lhs*, const **embb_time_t** * *rhs*)

Compares two time points.

Precondition

`lhs` and `rhs` are not NULL and properly initialized.

Returns

-1 if `lhs` < `rhs`
0 if `lhs` == `rhs`
1 if `lhs` > `rhs`

Concurrency

Not thread-safe

Parameters

`in`	*lhs*	Pointer to left-hand side operand
`in`	*rhs*	Pointer to right-hand side operand

5.50 Error

Error codes for function return values.

Enumerations

- enum embb_errors_t {
  EMBB_SUCCESS, EMBB_NOMEM, EMBB_TIMEDOUT, EMBB_BUSY,
  EMBB_OVERFLOW, EMBB_UNDERFLOW, EMBB_ERROR }

  Return value codes for functions.

5.50.1 Detailed Description

Error codes for function return values.

5.50.2 Enumeration Type Documentation

5.50.2.1 enum embb_errors_t

Return value codes for functions.

Enumerator

EMBB_SUCCESS Successful.

EMBB_NOMEM Error, not enough memory.

EMBB_TIMEDOUT Error, timed out.

EMBB_BUSY Resource busy.

EMBB_OVERFLOW Error, numeric overflow.

EMBB_UNDERFLOW Error, numeric underflow.

EMBB_ERROR Error, not further specified.

5.51 Logging

Simple logging facilities.

Typedefs

- typedef void(∗ embb_log_function_t) (void ∗context, char const ∗message)

  Logging function type.

Enumerations

- enum embb_log_level_t {
  EMBB_LOG_LEVEL_NONE, EMBB_LOG_LEVEL_ERROR, EMBB_LOG_LEVEL_WARNING, EMBB_LO↩
  G_LEVEL_INFO,
  EMBB_LOG_LEVEL_TRACE }

  Log levels available for filtering the log.

Functions

- void embb_log_write_file (void ∗context, char const ∗message)

  Default logging function.
- void embb_log_set_log_level (embb_log_level_t log_level)

  Sets the global log level.
- void embb_log_set_log_function (void ∗context, embb_log_function_t func)

  Sets the global logging function.
- void embb_log_write (char const ∗channel, embb_log_level_t log_level, char const ∗message,...)

  Logs a message to the given channel with the specified log level.
- void embb_log_trace (char const ∗channel, char const ∗message,...)

  Logs a message to the given channel with EMBB_LOG_LEVEL_TRACE using embb_log_write().
- void embb_log_info (char const ∗channel, char const ∗message,...)

  Logs a message to the given channel with EMBB_LOG_LEVEL_INFO using embb_log_write().
- void embb_log_warning (char const ∗channel, char const ∗message,...)

  Logs a message to the given channel with EMBB_LOG_LEVEL_WARNING using embb_log_write().
- void embb_log_error (char const ∗channel, char const ∗message,...)

  Logs a message to the given channel with EMBB_LOG_LEVEL_ERROR using embb_log_write().

5.51.1 Detailed Description

Simple logging facilities.

5.51.2 Typedef Documentation

5.51.2.1 typedef void(∗ embb_log_function_t) (void ∗context, char const ∗message)

Logging function type.

This function is used by embb_log_write() to transfer a log message to its desired destination. The user may specify a pointer to a context that contains additional data (filter rules, file handles etc.) needed to put the message where it should go. This pointer might be NULL if no additional data is needed.

Concurrency

Thread-safe

5.51.3 Enumeration Type Documentation

5.51.3.1 enum embb_log_level_t

Log levels available for filtering the log.

Enumerator

- ***EMBB_LOG_LEVEL_NONE*** show no log messages
- ***EMBB_LOG_LEVEL_ERROR*** show errors only
- ***EMBB_LOG_LEVEL_WARNING*** show warnings and errors
- ***EMBB_LOG_LEVEL_INFO*** show info, warnings, and errors
- ***EMBB_LOG_LEVEL_TRACE*** show everything

5.51.4 Function Documentation

5.51.4.1 void embb_log_write_file (void ∗ *context,* char const ∗ *message*)

Default logging function.

Writes to the given file (context needs to be a FILE∗).

Precondition

`context` is not NULL.

Concurrency

Thread-safe

Parameters

`in`	*context*	User data, in this case a FILE∗ file handle.
`in`	*message*	The message to write

5.51.4.2 void embb_log_set_log_level (embb_log_level_t *log_level*)

Sets the global log level.

This determines what messages will be shown, messages with a more detailed log level will be filtered out. The default log level is EMBB_LOG_LEVEL_NONE.

Concurrency

Not thread-safe

Parameters

in	*log_level*	Log level to use for filtering

5.51.4.3 void embb_log_set_log_function (void * *context*, embb_log_function_t *func*)

Sets the global logging function.

The logging function implements the mechanism for transferring log messages to their destination. `context` is a pointer to data the user needs in the function to determine where the messages should go (may be NULL if no additional data is needed). The default logging function is embb_log_write_file() with context set to `stdout`.

See also

embb_log_function_t

Concurrency

Not thread-safe

Parameters

in	*context*	User context to supply as the first parameter of the logging function
in	*func*	The logging function

5.51.4.4 void embb_log_write (char const * *channel*, embb_log_level_t *log_level*, char const * *message*, ...)

Logs a message to the given channel with the specified log level.

If the log level is greater than the configured log level for the channel, the message will be ignored.

See also

embb_log_set_log_level, embb_log_set_log_function

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id for filtering the log later on. Might be NULL, channel identifier will be "global" in that case
in	*log_level*	Log level to use
in	*message*	Message to convey, may use `printf` style formatting

5.51.4.5 void embb_log_trace (char const * *channel*, char const * *message*, ...)

Logs a message to the given channel with EMBB_LOG_LEVEL_TRACE using embb_log_write().

In non-debug builds, this function does nothing.

See also

embb_log_write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

5.51.4.6 void embb_log_info (char const * *channel*, char const * *message*, ...)

Logs a message to the given channel with EMBB_LOG_LEVEL_INFO using embb_log_write().

In non-debug builds, this function does nothing.

See also

embb_log_write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

5.51.4.7 void embb_log_warning (char const * *channel*, char const * *message*, ...)

Logs a message to the given channel with EMBB_LOG_LEVEL_WARNING using embb_log_write().

See also

embb_log_write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

5.51.4.8 void embb_log_error (char const * *channel*, char const * *message*, ...)

Logs a message to the given channel with EMBB_LOG_LEVEL_ERROR using embb_log_write().

See also

embb_log_write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

5.52 Memory Allocation

Functions for dynamic memory allocation.

Functions

- void * embb_alloc (size_t size)
  Allocates `size` bytes of memory.
- void embb_free (void *ptr)
  Frees memory that has been allocated by embb_alloc() for some pointer `ptr`.
- void * embb_alloc_aligned (size_t alignment, size_t size)
  Allocates `size` bytes of memory with alignment `alignment`.
- void * embb_alloc_cache_aligned (size_t size)
  Allocates `size` bytes of cache-aligned memory.
- void embb_free_aligned (void *ptr)
  Frees memory that has been allocated by an aligned method for `ptr`.
- size_t embb_get_bytes_allocated ()
  Returns the total number of bytes currently allocated.

5.52.1 Detailed Description

Functions for dynamic memory allocation.

There are functions for aligned and unaligned memory allocation. In debug mode, memory usage is tracked to detect memory leaks.

5.52.2 Function Documentation

5.52.2.1 void* embb_alloc (size_t *size*)

Allocates `size` bytes of memory.

Keeps track of allocated memory in debug mode.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Dynamic memory allocation

`size+3*sizeof(size_t)` bytes in debug mode, otherwise `size` bytes

Concurrency

Thread-safe

See also

embb_get_bytes_allocated()

Parameters

in	*size*	Size of memory block to be allocated in bytes

5.52.2.2 void embb_free (void * *ptr*)

Frees memory that has been allocated by embb_alloc() for some pointer `ptr`.

Keeps track of freed memory in debug mode.

Precondition

`ptr` is not NULL.

Concurrency

Thread-safe

See also

embb_get_bytes_allocated()

Parameters

in,out	*ptr*	Pointer to memory block to be freed

5.52.2.3 void* embb_alloc_aligned (size_t *alignment,* size_t *size*)

Allocates `size` bytes of memory with alignment `alignment`.

This function can be used to align objects to certain boundaries such as cache lines, memory pages, etc.

Keeps track of allocated memory in debug mode.

It is not required that `size` is a multiple of `alignment` as, e.g., for the `aligned_alloc` function of the C11 standard.

Precondition

The alignment has to be power of 2 and a multiple of `size(void*)`.

Postcondition

The returned pointer is a multiple of `alignment`.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Dynamic memory allocation

Debug mode: Let `n` be the number of aligned cells necessary to fit the payload. Then, `(n+1)*alignment+3*size_of(size_t)-1` bytes are allocated.
Release mode: `size` bytes are requested using the functions provided by the operating systems.

Concurrency

Thread-safe

Note

Memory allocated using this function must be freed using embb_free_aligned().

See also

embb_alloc_cache_aligned(), embb_free_aligned(), embb_get_bytes_allocated()

Parameters

in	*alignment*	Alignment in bytes
in	*size*	Size of memory block to be allocated in bytes

5.52.2.4 void* embb_alloc_cache_aligned (size_t *size*)

Allocates `size` bytes of cache-aligned memory.

Specialized version of embb_alloc_aligned(). The alignment is chosen automatically (usually 64 bytes).

Keeps track of allocated memory in debug mode.

Postcondition

The returned pointer is a multiple of the cache line size.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Dynamic memory allocation

See embb_alloc_aligned()

Concurrency

Thread-safe

Note

Memory allocated using this function must be freed using embb_free_aligned().

See also

embb_alloc_aligned(), embb_free_aligned(), embb_get_bytes_allocated()

Parameters

`in`	*size*	Size of memory block to be allocated in bytes

5.52.2.5 void embb_free_aligned (void * *ptr*)

Frees memory that has been allocated by an aligned method for `ptr`.

The available aligned methods are embb_alloc_aligned() or embb_alloc_cache_aligned().

Keeps track of freed memory in debug mode.

Precondition

`ptr` is not NULL and was allocated by an aligned method.

Concurrency

Thread-safe

See also

embb_alloc_aligned(), embb_alloc_cache_aligned(), embb_get_bytes_allocated()

Parameters

`in,out`	*ptr*	Pointer to memory block to be freed

5.52.2.6 size_t embb_get_bytes_allocated ( )

Returns the total number of bytes currently allocated.

Only the bytes allocated by embb_alloc(), embb_alloc_aligned(), and embb_alloc_cache_aligned() in debug mode are counted.

Returns

Number of currently allocated bytes in debug mode, otherwise 0.

Concurrency

Thread-safe and wait-free

See also

embb_alloc(), embb_alloc_aligned(), embb_alloc_cache_aligned()

5.53 Mutex

Mutexes for thread synchronization.

Typedefs

- typedef opaque_type embb_mutex_t

  Opaque type representing a mutex.
- typedef opaque_type embb_spinlock_t

  Opaque type representing a spinlock.

Enumerations

- enum { EMBB_MUTEX_PLAIN, EMBB_MUTEX_RECURSIVE }

  Types of mutexes to be used in embb_mutex_init()

Functions

- int embb_mutex_init (embb_mutex_t ∗mutex, int type)

  Initializes a mutex.
- int embb_mutex_lock (embb_mutex_t ∗mutex)

  Waits until the mutex can be locked and locks it.
- int embb_mutex_try_lock (embb_mutex_t ∗mutex)

  Tries to lock the mutex and returns immediately.
- int embb_mutex_unlock (embb_mutex_t ∗mutex)

  Unlocks a locked mutex.
- void embb_mutex_destroy (embb_mutex_t ∗mutex)

  Destroys a mutex and frees its resources.
- int embb_spin_init (embb_spinlock_t ∗spinlock)

  Initializes a spinlock.
- int embb_spin_lock (embb_spinlock_t ∗spinlock)

  Spins until the spinlock can be locked and locks it.
- int embb_spin_try_lock (embb_spinlock_t ∗spinlock, unsigned int max_number_spins)

  Tries to lock the spinlock and returns if not successful.
- int embb_spin_unlock (embb_spinlock_t ∗spinlock)

  Unlocks a locked spinlock.
- void embb_spin_destroy (embb_spinlock_t ∗spinlock)

  Destroys a spinlock and frees its resources.

5.53.1 Detailed Description

Mutexes for thread synchronization.

Provides an abstraction from platform-specific mutex implementations. Plain and recursive mutexes are available, where the plain version can only be locked once by the same thread.

5.53.2 Typedef Documentation

5.53.2.1 typedef opaque_type embb_mutex_t

Opaque type representing a mutex.

5.53.2.2 typedef opaque_type embb_spinlock_t

Opaque type representing a spinlock.

5.53.3 Enumeration Type Documentation

5.53.3.1 anonymous enum

Types of mutexes to be used in embb_mutex_init()

Enumerator

EMBB_MUTEX_PLAIN Mutex can be locked only once by the same thread.

EMBB_MUTEX_RECURSIVE Mutex can be locked recursively by the same thread.

5.53.4 Function Documentation

5.53.4.1 int embb_mutex_init (embb_mutex_t * *mutex,* int *type*)

Initializes a mutex.

Postcondition

`mutex` is initialized

Returns

EMBB_SUCCESS if mutex could be initialized
EMBB_ERROR otherwise

Dynamic memory allocation

(Potentially) allocates dynamic memory

Concurrency

Not thread-safe

See also

embb_mutex_destroy()

Parameters

`out`	*mutex*	Pointer to mutex
`in`	*type*	EMBB_MUTEX_PLAIN or EMBB_MUTEX_RECURSIVE. There is no guarantee that a mutex is non-recursive if the plain type is given.

5.53.4.2 int embb_mutex_lock (embb_mutex_t * *mutex*)

Waits until the mutex can be locked and locks it.

Precondition

`mutex` is initialized
If the mutex type is plain, `mutex` must not be locked by the current thread.

Postcondition

If successful, `mutex` is locked.

Returns

EMBB_SUCCESS if mutex could be locked
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_mutex_try_lock(), embb_mutex_unlock()

Parameters

`in,out`	*mutex*	Pointer to mutex

5.53.4.3 int embb_mutex_try_lock (embb_mutex_t * *mutex*)

Tries to lock the mutex and returns immediately.

Precondition

`mutex` is initialized

Postcondition

If successful, `mutex` is locked

Returns

EMBB_SUCCESS if mutex could be locked
EMBB_BUSY if mutex could not be locked
EMBB_ERROR if an error occurred

Concurrency

Thread-safe

See also

embb_mutex_lock(), embb_mutex_unlock()

Parameters

`in,out`	*mutex*	Pointer to mutex

5.53.4.4 int embb_mutex_unlock (embb_mutex_t * *mutex*)

Unlocks a locked mutex.

Precondition

`mutex` has been locked by the current thread.

Postcondition

If successful and when the given mutex type is plain, `mutex` is unlocked. If its type is recursive, `mutex` is only unlocked if the number of successful unlocks has reached the number of successful locks done by the current thread.

Returns

EMBB_SUCCESS if the operation was successful
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_mutex_lock(), embb_mutex_try_lock()

Parameters

`in,out`	*mutex*	Pointer to mutex

5.53.4.5 void embb_mutex_destroy (embb_mutex_t * mutex)

Destroys a mutex and frees its resources.

Precondition

`mutex` is initialized and is not NULL.

Postcondition

`mutex` is uninitialized

Concurrency

Not thread-safe

See also

embb_mutex_init()

Parameters

`in,out`	*mutex*	Pointer to mutex

5.53.4.6 int embb_spin_init (embb_spinlock_t * spinlock)

Initializes a spinlock.

Precondition

`spinlock` is uninitialized

Postcondition

`spinlock` is initialized

Returns

EMBB_SUCCESS if spinlock could be initialized
EMBB_ERROR otherwise

Dynamic memory allocation

(Potentially) allocates dynamic memory

Concurrency

Not thread-safe

See also

embb_spinlock_destroy()

Parameters

out	*spinlock*	Pointer to spinlock

5.53.4.7 int embb_spin_lock (embb_spinlock_t * *spinlock*)

Spins until the spinlock can be locked and locks it.

Note

This method yields the current thread in regular, implementation-defined intervals.

Precondition

`spinlock` is initialized

Postcondition

If successful, `spinlock` is locked.

Returns

EMBB_SUCCESS if spinlock could be locked.
EMBB_ERROR if an error occurred.

Concurrency

Thread-safe

See also

embb_spinlock_try_lock(), embb_mutex_unlock()

Parameters

in,out	*spinlock*	Pointer to spinlock

5.53.4.8 int embb_spin_try_lock (embb_spinlock_t * *spinlock*, unsigned int *max_number_spins*)

Tries to lock the spinlock and returns if not successful.

Precondition

`spinlock` is initialized

Postcondition

If successful, `spinlock` is locked

Returns

EMBB_SUCCESS if spinlock could be locked
EMBB_BUSY if spinlock could not be locked
EMBB_ERROR if an error occurred

Concurrency

Thread-safe

See also

embb_spin_lock(), embb_spin_unlock()

Parameters

`in,out`	*spinlock*	Pointer to spinlock
`in`	*max_number_spins*	Maximal count of spins to perform, trying to acquire lock. Note that passing 0 here results in not trying to obtain the lock at all.

5.53.4.9 int embb_spin_unlock (embb_spinlock_t * *spinlock*)

Unlocks a locked spinlock.

Precondition

`spinlock` has been locked by the current thread.

Postcondition

If successful, `spinlock` is unlocked.

Returns

EMBB_SUCCESS if the operation was successful
EMBB_ERROR otherwise

Concurrency

Thread-safe

See also

embb_spin_lock(), embb_spin_try_lock()

Parameters

`in,out`	*spinlock*	Pointer to spinlock

5.53.4.10 void embb_spin_destroy (embb_spinlock_t * *spinlock*)

Destroys a spinlock and frees its resources.

Precondition

`spinlock` is initialized and is not NULL.

Postcondition

`spinlock` is uninitialized

Concurrency

Not thread-safe

See also

embb_spin_init()

Parameters

in,out	*spinlock*	Pointer to spinlock

5.54 Thread

Threads supporting thread-to-core affinities.

Typedefs

- typedef opaque_type embb_thread_t

  Opaque type representing a thread of execution.
- typedef int(∗ embb_thread_start_t) (void ∗)

  Thread start function pointer type.

Enumerations

- enum embb_thread_priority_t {
EMBB_THREAD_PRIORITY_IDLE, **EMBB_THREAD_PRIORITY_LOWEST**, **EMBB_THREAD_PRIORI↩
TY_BELOW_NORMAL**, **EMBB_THREAD_PRIORITY_NORMAL**,
EMBB_THREAD_PRIORITY_ABOVE_NORMAL, **EMBB_THREAD_PRIORITY_HIGHEST**, **EMBB_THR↩
EAD_PRIORITY_TIME_CRITICAL** }

  Thread priority type.

Functions

- unsigned int embb_thread_get_max_count ()

  Returns the maximum number of threads handled by EMB[2].
- void embb_thread_set_max_count (unsigned int max)

  Sets maximum number of threads handled by EMBB.
- embb_thread_t embb_thread_current ()

  Returns the calling thread (that is, this thread).
- void embb_thread_yield ()

  Reschedule the current thread for later execution.
- int embb_thread_create (embb_thread_t ∗thread, const embb_core_set_t ∗core_set, embb_thread_start_t function, void ∗arg)

  Creates and runs a thread.
- int embb_thread_create_with_priority (embb_thread_t ∗thread, const embb_core_set_t ∗core_set, embb_↩
thread_priority_t priority, embb_thread_start_t function, void ∗arg)

  Creates and runs a thread.
- int embb_thread_join (embb_thread_t ∗thread, int ∗result_code)

  Waits until the given thread has finished execution.
- int embb_thread_equal (const embb_thread_t ∗lhs, const embb_thread_t ∗rhs)

  Compares two threads represented by their handles for equality.

5.54.1 Detailed Description

Threads supporting thread-to-core affinities.

Provides an abstraction from platform-specific threading implementations to create, manage, and join threads of execution. Support for thread-to-core affinities is given on thread creation by using the core set functionality.

5.54.2 Typedef Documentation

5.54.2.1 typedef opaque_type embb_thread_t

Opaque type representing a thread of execution.

5.54.2.2 typedef int(* embb_thread_start_t) (void *)

Thread start function pointer type.

The return value can be used to return a user-defined exit code when the thread is joined.

5.54.3 Enumeration Type Documentation

5.54.3.1 enum embb_thread_priority_t

Thread priority type.

5.54.4 Function Documentation

5.54.4.1 unsigned int embb_thread_get_max_count ( )

Returns the maximum number of threads handled by EMB^2.

The maximum thread number concerns all threads in a program using EMB^2 functionalities or data structures, regardless of whether a thread is started by EMB^2 or other threading libraries. Each thread that makes use of EMB^2 at least once consumes one entry in the internal tables. The entry is permanently consumed during a program run, even if the thread does not exist any longer. If more threads than the maximum thread count access EMB^2, undefined behavior or abortion of program execution can occur.

Returns

Maximum number of threads

Concurrency

Thread-safe and lock-free

See also

embb_thread_set_max_count()

5.54.4.2 void embb_thread_set_max_count (unsigned int *max*)

Sets maximum number of threads handled by EMBB.

It needs to be set before any EMB^2 functionalities are used or data structures are defined, unless the default value is sufficient.

Concurrency

Not thread-safe

See also

embb_thread_get_max_count()

Parameters

in	*max*	Maximum number of threads

5.54.4.3 embb_thread_t embb_thread_current ( )

Returns the calling thread (that is, this thread).

The returned handle is only valid for the thread calling the function.

Returns

Calling thread

Concurrency

Thread-safe

5.54.4.4 void embb_thread_yield ( )

Reschedule the current thread for later execution.

This is only a request, the realization depends on the implementation and the scheduler employed by the operating system.

Concurrency

Thread-safe

5.54.4.5 int embb_thread_create (embb_thread_t ∗ *thread,* const embb_core_set_t ∗ *core_set,* embb_thread_start_t *function,* void ∗ *arg*)

Creates and runs a thread.

Precondition

The given thread is not running and has not yet been successfully joined.

Postcondition

On success, the given thread has started to run.

Returns

EMBB_SUCCESS if the thread could be created.
EMBB_NOMEM if there was insufficient amount of memory
EMBB_ERROR otherwise.

Dynamic memory allocation

Dynamically allocates a small constant amount of memory to store the function and argument pointers. This memory is freed when the thread is joined.

Concurrency

Not thread-safe

See also

embb_thread_join()

Parameters

out	*thread*	Thread to be run
in	*core_set*	Set of cores on which the thread shall be executed. Can be NULL to indicate automatic thread scheduling by the OS.
in	*function*	Function which is executed by the thread when started. Has to be of type embb_thread_start_t.
in,out	*arg*	Argument to thread start function. Can be NULL.

5.54.4.6 int embb_thread_create_with_priority (**embb_thread_t** ∗ *thread,* const **embb_core_set_t** ∗ *core_set,* **embb_thread_priority_t** *priority,* **embb_thread_start_t** *function,* void ∗ *arg*)

Creates and runs a thread.

Precondition

The given thread is not running and has not yet been successfully joined.

Postcondition

On success, the given thread has started to run.

Returns

EMBB_SUCCESS if the thread could be created.
EMBB_NOMEM if there was insufficient amount of memory
EMBB_ERROR otherwise.

Dynamic memory allocation

Dynamically allocates a small constant amount of memory to store the function and argument pointers. This memory is freed when the thread is joined.

Concurrency

Not thread-safe

See also

embb_thread_join()

Parameters

out	*thread*	Thread to be run
in	*core_set*	Set of cores on which the thread shall be executed. Can be NULL to indicate automatic thread scheduling by the OS.
in	*priority*	Priority to run the thread at.
in	*function*	Function which is executed by the thread when started. Has to be of type embb_thread_start_t.
in,out	*arg*	Argument to thread start function. Can be NULL.

5.54.4.7 int embb_thread_join (**embb_thread_t** ∗ *thread,* int ∗ *result_code*)

Waits until the given thread has finished execution.

Precondition

The given thread has been successfully created using embb_thread_create().

Postcondition

If successful, the thread has finished execution and all memory associated to the thread has been freed.

Returns

EMBB_SUCCESS if thread was joined
EMBB_ERROR otherwise

Concurrency

Not thread-safe

See also

embb_thread_create()

Parameters

`in,out`	*thread*	Thread to be joined
`out`	*result_code*	Memory location (or NULL) for thread result code

5.54.4.8 int embb_thread_equal (const **embb_thread_t** ∗ *lhs,* const **embb_thread_t** ∗ *rhs*)

Compares two threads represented by their handles for equality.

Returns

Non-zero, if equal
0, otherwise

Concurrency

Not thread-safe

Parameters

`in`	*lhs*	First thread (left-hand side of equality sign)
`in`	*rhs*	Second thread (right-hand side of equality sign)

5.55 Thread-Specific Storage

Thread-specific storage.

Typedefs

- typedef opaque_type embb_tss_t

  Opaque type representing a TSS.

Functions

- int embb_tss_create (embb_tss_t ∗tss)

  Creates thread-specific storage (TSS) pointer slots.
- int embb_tss_set (embb_tss_t ∗tss, void ∗value)

  Sets thread-specific slot value of the current thread.
- void ∗ embb_tss_get (const embb_tss_t ∗tss)

  Gets thread-specific TSS slot value of the current thread.
- void embb_tss_delete (embb_tss_t ∗tss)

  Deletes all slots of the given TSS.

5.55.1 Detailed Description

Thread-specific storage.

Implements thread-specific storage (TSS), that is, memory locations that are individual for each thread. Each thread has its own slot for a memory address that can point to a (thread-specific) value. The value pointed to has to be managed, i.e., created and deleted, by the user.

5.55.2 Typedef Documentation

5.55.2.1 typedef opaque_type **embb_tss_t**

Opaque type representing a TSS.

5.55.3 Function Documentation

5.55.3.1 int embb_tss_create (**embb_tss_t** ∗ ***tss***)

Creates thread-specific storage (TSS) pointer slots.

Precondition

The given TSS has not yet been created or has already been deleted.

Returns

EMBB_SUCCESS if storage could be created
EMBB_NOMEM if not enough memory was available

Dynamic memory allocation

embb_thread_get_max_count() pointers

Concurrency

Not thread-safe

See also

embb_tss_delete(), embb_thread_get_max_count()

Parameters

out	*tss*	Pointer to TSS

5.55.3.2 int embb_tss_set (embb_tss_t ∗ *tss*, void ∗ *value*)

Sets thread-specific slot value of the current thread.

The value pointed to needs to be managed (created, deleted) by the user.

Precondition

The given TSS has been created

Returns

EMBB_SUCCESS if value could be set
EMBB_ERROR if no thread index could be obtained, that is, the maximum number of threads has been exceeded.

Concurrency

Thread-safe and lock-free

See also

embb_tss_get(), embb_thread_get_max_count()

Parameters

in,out	*tss*	Pointer to TSS
in	*value*	Pointer to be stored in TSS

5.55.3.3 void* embb_tss_get (const embb_tss_t * tss)

Gets thread-specific TSS slot value of the current thread.

Precondition

The given TSS has been created

Returns

Thread-specific value if embb_tss_set() has previously been called with a valid address. NULL, if no value was set or the calling thread could not obtain a thread-specific index.

Concurrency

Thread-safe and lock-free

See also

embb_tss_set()

Parameters

in	*tss*	Pointer to TSS

5.55.3.4 void embb_tss_delete (embb_tss_t * tss)

Deletes all slots of the given TSS.

Does not delete the values pointed to.

Precondition

`tss` has been created successfully and is not NULL.

Postcondition

All slots are deleted

Concurrency

Not thread-safe

See also

embb_tss_create()

Parameters

in,out	*tss*	Pointer to TSS

5.56 MTAPI OpenCL Plugin

Provides functionality to execute tasks on OpenCL devices.

Functions

- void mtapi_opencl_plugin_initialize (MTAPI_OUT mtapi_status_t *status)

    Initializes the MTAPI OpenCL environment on a previously initialized MTAPI node.
- void mtapi_opencl_plugin_finalize (MTAPI_OUT mtapi_status_t *status)

    Finalizes the MTAPI OpenCL environment on the local MTAPI node.
- mtapi_action_hndl_t mtapi_opencl_action_create (MTAPI_IN mtapi_job_id_t job_id, MTAPI_IN char *kernel_source, MTAPI_IN char *kernel_name, MTAPI_IN mtapi_size_t local_work_size, MTAPI_IN mtapi_size_t element_size, MTAPI_IN void *node_local_data, MTAPI_IN mtapi_size_t node_local_data_size, MTAPI_OUT mtapi_status_t *status)

    This function creates an OpenCL action.
- cl_context mtapi_opencl_get_context (MTAPI_OUT mtapi_status_t *status)

    Retrieves the handle of the OpenCL context used by the plugin.

5.56.1 Detailed Description

Provides functionality to execute tasks on OpenCL devices.

5.56.2 Function Documentation

5.56.2.1 void mtapi_opencl_plugin_initialize (MTAPI_OUT mtapi_status_t * *status*)

Initializes the MTAPI OpenCL environment on a previously initialized MTAPI node.

It must be called on all nodes using the MTAPI OpenCL plugin.

Application software using MTAPI OpenCL must call mtapi_opencl_plugin_initialize() once per node. It is an error to call mtapi_opencl_plugin_initialize() multiple times from a given node, unless mtapi_opencl_plugin_finalize() is called in between.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI OpenCL couldn't be initialized.

See also

mtapi_opencl_plugin_finalize()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.56.2.2 void mtapi_opencl_plugin_finalize (MTAPI_OUT mtapi_status_t ∗ *status*)

Finalizes the MTAPI OpenCL environment on the local MTAPI node.

It has to be called by each node using MTAPI OpenCL. It is an error to call mtapi_opencl_plugin_finalize() without first calling mtapi_opencl_plugin_initialize(). An MTAPI node can call mtapi_opencl_plugin_finalize() once for each call to mtapi_opencl_plugin_initialize(), but it is an error to call mtapi_opencl_plugin_finalize() multiple times from a given node unless mtapi_opencl_plugin_initialize() has been called prior to each mtapi_opencl_plugin_finalize() call.

All OpenCL tasks that have not completed and that have been started on the node where mtapi_opencl_plugin_finalize() is called will be canceled (see mtapi_task_cancel()). mtapi_opencl_plugin_finalize() blocks until all tasks that have been started on the same node return. Tasks that execute actions on the node where mtapi_opencl_plugin_finalize() is called, also block finalization of the MTAPI OpenCL system on that node.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI OpenCL couldn't be finalized.

See also

mtapi_opencl_plugin_initialize(), mtapi_task_cancel()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.56.2.3 mtapi_action_hndl_t mtapi_opencl_action_create (MTAPI_IN mtapi_job_id_t *job_id,* MTAPI_IN char ∗ *kernel_source,* MTAPI_IN char ∗ *kernel_name,* MTAPI_IN mtapi_size_t *local_work_size,* MTAPI_IN mtapi_size_t *element_size,* MTAPI_IN void ∗ *node_local_data,* MTAPI_IN mtapi_size_t *node_local_data_size,* MTAPI_OUT mtapi_status_t ∗ *status*)

This function creates an OpenCL action.

It is called on the node where the user wants to execute an action on an OpenCL device. An OpenCL action contains a reference to a local job, the kernel source to compile and execute on the OpenCL device, the name of the kernel function, a local work size (see OpenCL specification for details) and the size of one element in the result buffer. After an OpenCL action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`.

An OpenCL action's life-cycle begins with mtapi_opencl_action_create(), and ends when mtapi_action_delete() or mtapi_finalize() is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

If `node_local_data_size` is not zero, `node_local_data` specifies the start of node local data shared by kernel functions executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

On success, an action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the action already exists, `status` will be set to `MTAPI_↩ ERR_ACTION_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The `job_id` is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
`MTAPI_ERR_ACTION_EXISTS`	This action is already created.
`MTAPI_ERR_ACTION_LIMIT`	Exceeded maximum number of actions allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_UNKNOWN`	The kernel could not be compiled or no OpenCL device was available.

See also

mtapi_action_delete(), mtapi_finalize()

Returns

Handle to newly created OpenCL action, invalid handle on error

Concurrency

Thread-safe

Parameters

`in`	*job_id*	Job id
`in`	*kernel_source*	Pointer to kernel source
`in`	*kernel_name*	Name of the kernel function
`in`	*local_work_size*	Size of local work group
`in`	*element_size*	Size of one element in the result buffer
`in`	*node_local_data*	Data shared across tasks
`in`	*node_local_data_size*	Size of shared data
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.56.2.4 cl_context mtapi_opencl_get_context (MTAPI_OUT mtapi_status_t ∗ *status*)

Retrieves the handle of the OpenCL context used by the plugin.

Returns

cl_context used by the plugin

Concurrency

Thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.57 MTAPI Network Plugin

Provides functionality to distribute tasks across nodes in a TCP/IP network.

Functions

- void mtapi_network_plugin_initialize (MTAPI_IN char ∗host, MTAPI_IN mtapi_uint16_t port, MTAPI_IN mtapi_uint16_t max_connections, MTAPI_IN mtapi_size_t buffer_size, MTAPI_OUT mtapi_status_t ∗status)

    Initializes the MTAPI network environment on a previously initialized MTAPI node.
- void mtapi_network_plugin_finalize (MTAPI_OUT mtapi_status_t ∗status)

    Finalizes the MTAPI network environment on the local MTAPI node.
- mtapi_action_hndl_t mtapi_network_action_create (MTAPI_IN mtapi_domain_t domain_id, MTAPI_IN mtapi_job_id_t local_job_id, MTAPI_IN mtapi_job_id_t remote_job_id, MTAPI_IN char ∗host, MTAPI_IN mtapi_uint16_t port, MTAPI_OUT mtapi_status_t ∗status)

    This function creates a network action.

5.57.1 Detailed Description

Provides functionality to distribute tasks across nodes in a TCP/IP network.

5.57.2 Function Documentation

5.57.2.1 void mtapi_network_plugin_initialize (MTAPI_IN char ∗ *host,* MTAPI_IN mtapi_uint16_t *port,* MTAPI_IN mtapi_uint16_t *max_connections,* MTAPI_IN mtapi_size_t *buffer_size,* MTAPI_OUT mtapi_status_t ∗ *status*)

Initializes the MTAPI network environment on a previously initialized MTAPI node.

It must be called on all nodes using the MTAPI network plugin.

Application software using MTAPI network must call mtapi_network_plugin_initialize() once per node. It is an error to call mtapi_network_plugin_initialize() multiple times from a given node, unless mtapi_network_plugin_finalize() is called in between.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI network couldn't be initialized.

See also

mtapi_network_plugin_finalize()

Concurrency

Not thread-safe

Parameters

in	*host*	The interface to listen on, if MTAPI_NULL is given the plugin will listen on all available interfaces.
in	*port*	The port to listen on.
in	*max_connections*	Maximum concurrent connections accepted by the plugin.
in	*buffer_size*	Capacity of the transfer buffers, this should be chosen big enough to hold argument and result buffers.
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.57.2.2 void mtapi_network_plugin_finalize (MTAPI_OUT mtapi_status_t * *status*)

Finalizes the MTAPI network environment on the local MTAPI node.

It has to be called by each node using MTAPI network. It is an error to call mtapi_network_plugin_finalize() without first calling mtapi_network_plugin_initialize(). An MTAPI node can call mtapi_network_plugin_finalize() once for each call to mtapi_network_plugin_initialize(), but it is an error to call mtapi_network_plugin_finalize() multiple times from a given node unless mtapi_network_plugin_initialize() has been called prior to each mtapi_network_plugin_finalize() call.

All network tasks that have not completed and that have been started on the node where mtapi_network_plugin_finalize() is called will be canceled (see mtapi_task_cancel()). mtapi_network_plugin_finalize() blocks until all tasks that have been started on the same node return. Tasks that execute actions on the node where mtapi_network_plugin_finalize() is called, also block finalization of the MTAPI network system on that node.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI network couldn't be finalized.

See also

mtapi_network_plugin_initialize(), mtapi_task_cancel()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.57.2.3 mtapi_action_hndl_t mtapi_network_action_create (MTAPI_IN mtapi_domain_t *domain_id*, MTAPI_IN mtapi_job_id_t *local_job_id*, MTAPI_IN mtapi_job_id_t *remote_job_id*, MTAPI_IN char * *host*, MTAPI_IN mtapi_uint16_t *port*, MTAPI_OUT mtapi_status_t * *status*)

This function creates a network action.

It is called on the node where the user wants to execute an action on a remote node where the actual action is implemented. A network action contains a reference to a local job, a remote job and a remote domain as well as a host and port to connect to. After a network action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`. A network action's life-cycle begins with mtapi_network_action_create(), and ends when mtapi_action_delete() or mtapi_finalize() is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

A network action defines no node local data, instead the node local data of the remote action is used. The user has to make sure that the remote node local data matches what he expects the remote action to use if invoked through the network.

On success, an action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the action already exists, `status` will be set to `MTAPI_ERR_ACTION_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The `job_id` is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
`MTAPI_ERR_ACTION_EXISTS`	This action is already created.
`MTAPI_ERR_ACTION_LIMIT`	Exceeded maximum number of actions allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_UNKNOWN`	The remote node could not be reached or there was no local interface available.

See also

mtapi_action_delete(), mtapi_finalize()

Returns

Handle to newly created network action, invalid handle on error

Concurrency

Thread-safe

Parameters

`in`	*domain_id*	The domain the action is associated with
`in`	*local_job_id*	The ID of the local job
`in`	*remote_job_id*	The ID of the remote job
`in`	*host*	The host to connect to
`in`	*port*	The port the host is listening on
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

5.58 MTAPI CUDA Plugin

Provides functionality to execute tasks on CUDA devices.

Functions

- void mtapi_cuda_plugin_initialize (MTAPI_OUT mtapi_status_t ∗status)
  Initializes the MTAPI CUDA environment on a previously initialized MTAPI node.
- void mtapi_cuda_plugin_finalize (MTAPI_OUT mtapi_status_t ∗status)
  Finalizes the MTAPI CUDA environment on the local MTAPI node.
- mtapi_action_hndl_t mtapi_cuda_action_create (MTAPI_IN mtapi_job_id_t job_id, MTAPI_IN char ∗kernel_source, MTAPI_IN char ∗kernel_name, MTAPI_IN mtapi_size_t local_work_size, MTAPI_IN mtapi_size_t element_size, MTAPI_IN void ∗node_local_data, MTAPI_IN mtapi_size_t node_local_data_size, MTAPI_OUT mtapi_status_t ∗status)
  This function creates a CUDA action.
- CUcontext mtapi_cuda_get_context (MTAPI_OUT mtapi_status_t ∗status)
  Retrieves the handle of the CUDA context used by the plugin.

5.58.1 Detailed Description

Provides functionality to execute tasks on CUDA devices.

5.58.2 Function Documentation

5.58.2.1 void mtapi_cuda_plugin_initialize (MTAPI_OUT mtapi_status_t ∗ *status*)

Initializes the MTAPI CUDA environment on a previously initialized MTAPI node.

It must be called on all nodes using the MTAPI CUDA plugin.

Application software using MTAPI CUDA must call mtapi_cuda_plugin_initialize() once per node. It is an error to call mtapi_cuda_plugin_initialize() multiple times from a given node, unless mtapi_cuda_plugin_finalize() is called in between.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI CUDA couldn't be initialized.

See also

mtapi_cuda_plugin_finalize()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.58.2.2 void mtapi_cuda_plugin_finalize (MTAPI_OUT mtapi_status_t * *status*)

Finalizes the MTAPI CUDA environment on the local MTAPI node.

It has to be called by each node using MTAPI CUDA. It is an error to call mtapi_cuda_plugin_finalize() without first calling mtapi_cuda_plugin_initialize(). An MTAPI node can call mtapi_cuda_plugin_finalize() once for each call to mtapi_cuda_plugin_initialize(), but it is an error to call mtapi_cuda_plugin_finalize() multiple times from a given node unless mtapi_cuda_plugin_initialize() has been called prior to each mtapi_cuda_plugin_finalize() call.

All CUDA tasks that have not completed and that have been started on the node where mtapi_cuda_plugin_finalize() is called will be canceled (see mtapi_task_cancel()). mtapi_cuda_plugin_finalize() blocks until all tasks that have been started on the same node return. Tasks that execute actions on the node where mtapi_cuda_plugin_finalize() is called, also block finalization of the MTAPI CUDA system on that node.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_UNKNOWN`	MTAPI CUDA couldn't be finalized.

See also

mtapi_cuda_plugin_initialize(), mtapi_task_cancel()

Concurrency

Not thread-safe

Parameters

out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.58.2.3 mtapi_action_hndl_t mtapi_cuda_action_create (MTAPI_IN mtapi_job_id_t *job_id*, MTAPI_IN char * *kernel_source*, MTAPI_IN char * *kernel_name*, MTAPI_IN mtapi_size_t *local_work_size*, MTAPI_IN mtapi_size_t *element_size*, MTAPI_IN void * *node_local_data*, MTAPI_IN mtapi_size_t *node_local_data_size*, MTAPI_OUT mtapi_status_t * *status*)

This function creates a CUDA action.

It is called on the node where the user wants to execute an action on an CUDA device. A CUDA action contains a reference to a local job, the kernel source to compile and execute on the CUDA device, the name of the kernel function, a local work size (see CUDA specification for details) and the size of one element in the result buffer. After a CUDA action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`. A CUDA action's life-cycle begins with mtapi_cuda_action_create(), and ends when mtapi_action_delete() or mtapi_finalize() is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

If `node_local_data_size` is not zero, `node_local_data` specifies the start of node local data shared by kernel functions executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

On success, an action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the action already exists, `status` will be set to `MTAPI_ERR_ACTION_EXISTS` and the handle returned will not be a valid handle.

Error code	Description
`MTAPI_ERR_JOB_INVALID`	The `job_id` is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
`MTAPI_ERR_ACTION_EXISTS`	This action is already created.
`MTAPI_ERR_ACTION_LIMIT`	Exceeded maximum number of actions allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_UNKNOWN`	The kernel could not be compiled or no CUDA device was available.

See also

mtapi_action_delete(), mtapi_finalize()

Returns

Handle to newly created CUDA action, invalid handle on error

Concurrency

Thread-safe

Parameters

in	*job_id*	Job id
in	*kernel_source*	Pointer to kernel source
in	*kernel_name*	Name of the kernel function
in	*local_work_size*	Size of local work group
in	*element_size*	Size of one element in the result buffer
in	*node_local_data*	Data shared across tasks
in	*node_local_data_size*	Size of shared data
out	*status*	Pointer to error code, may be `MTAPI_NULL`

5.58.2.4 CUcontext mtapi_cuda_get_context (MTAPI_OUT mtapi_status_t * *status*)

Retrieves the handle of the CUDA context used by the plugin.

Returns

CUcontext used by the plugin

Concurrency

Thread-safe

Parameters

out	*status*	Pointer to error code, may be MTAPI_NULL

Chapter 6

Class Documentation

6.1 embb::mtapi::Action Class Reference

Holds the actual worker function used to execute a Task.

```
#include &lt;action.h&gt;
```

Public Member Functions

- Action ()
  Constructs an Action.
- Action (Action const &other)
  Copies an Action.
- Action & operator= (Action const &other)
  Copies an Action.
- void Delete ()
  Deletes an Action.
- mtapi_action_hndl_t GetInternal () const
  Returns the internal representation of this object.

6.1.1 Detailed Description

Holds the actual worker function used to execute a Task.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 embb::mtapi::Action::Action ( )

Constructs an Action.

The Action object will be invalid.

Concurrency

Thread-safe and wait-free

6.1.2.2 embb::mtapi::Action::Action (Action const & *other*)

Copies an Action.

Concurrency

Thread-safe and wait-free

Parameters

other	Action to copy

6.1.3 Member Function Documentation

6.1.3.1 Action& embb::mtapi::Action::operator= (Action const & *other*)

Copies an Action.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

other	Action to copy

6.1.3.2 void embb::mtapi::Action::Delete ( )

Deletes an Action.

Concurrency

Thread-safe

6.1.3.3 mtapi_action_hndl_t embb::mtapi::Action::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_action_hndl_t.

Concurrency

Thread-safe and wait-free

6.2 embb::mtapi::ActionAttributes Class Reference

Contains attributes of an Action.

```
#include <action_attributes.h>
```

Public Member Functions

- ActionAttributes ()

  Constructs an ActionAttributes object.
- ActionAttributes & SetGlobal (bool state)

  Sets the global property of an Action.
- ActionAttributes & SetAffinity (Affinity const &affinity)

  Sets the affinity of an Action.
- ActionAttributes & SetDomainShared (bool state)

  Sets the domain shared property of an Action.
- mtapi_action_attributes_t const & GetInternal () const

  Returns the internal representation of this object.

6.2.1 Detailed Description

Contains attributes of an Action.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 embb::mtapi::ActionAttributes::ActionAttributes ( )

Constructs an ActionAttributes object.

Concurrency

Thread-safe and wait-free

6.2.3 Member Function Documentation

6.2.3.1 ActionAttributes& embb::mtapi::ActionAttributes::SetGlobal (bool *state*)

Sets the global property of an Action.

This determines whether the object will be visible across nodes.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

state	The state to set

6.2.3.2 ActionAttributes& embb::mtapi::ActionAttributes::SetAffinity (Affinity const & *affinity*)

Sets the affinity of an Action.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

affinity	The Affinity to set.

6.2.3.3 ActionAttributes& embb::mtapi::ActionAttributes::SetDomainShared (bool *state*)

Sets the domain shared property of an Action.

This determines whether the object will be visible across domains.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

state	The state to set

6.2.3.4 mtapi_action_attributes_t const& embb::mtapi::ActionAttributes::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A reference to the internal mtapi_action_attributes_t structure.

Concurrency

Thread-safe and wait-free

6.3 embb::base::AdoptLockTag Struct Reference

Tag type for adopt UniqueLock constructor.

`#include <mutex.h>`

6.3.1 Detailed Description

Tag type for adopt UniqueLock constructor.

Use the adopt_lock variable in constructor calls.

6.4 embb::mtapi::Affinity Class Reference

Describes the affinity of an Action or Task to a worker thread of a Node.

`#include <affinity.h>`

Public Member Functions

- Affinity ()
  Constructs an Affinity object.
- Affinity (Affinity const &other)
  Copies an Affinity object.
- void operator= (Affinity const &other)
  Copies an Affinity object.
- Affinity (bool initial_affinity)
  Constructs an Affinity object with the given initial affinity.
- void Init (bool initial_affinity)
  Initializes an Affinity object with the given initial affinity.
- void Set (mtapi_uint_t worker, bool state)
  Sets affinity to the given worker.
- bool Get (mtapi_uint_t worker)
  Gets affinity to the given worker.
- mtapi_affinity_t GetInternal () const
  Returns the internal representation of this object.

6.4.1 Detailed Description

Describes the affinity of an Action or Task to a worker thread of a Node.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 embb::mtapi::Affinity::Affinity ( )

Constructs an Affinity object.

Concurrency

Not thread-safe

6.4.2.2 embb::mtapi::Affinity::Affinity (Affinity const & *other*)

Copies an Affinity object.

Concurrency

Thread-safe and wait-free

Parameters

other	The Affinity to copy from

6.4.2.3 embb::mtapi::Affinity::Affinity (bool *initial_affinity*)

Constructs an Affinity object with the given initial affinity.

If `initial_affinity` is `true` the Affinity will map to all worker threads, otherwise it will map to no worker threads.

Concurrency

Not thread-safe

Parameters

initial_affinity	The initial affinity to set.

6.4.3 Member Function Documentation

6.4.3.1 void embb::mtapi::Affinity::operator= (Affinity const & *other*)

Copies an Affinity object.

Concurrency

Thread-safe and wait-free

Parameters

other	The Affinity to copy from

6.4.3.2 void embb::mtapi::Affinity::Init (bool *initial_affinity*)

Initializes an Affinity object with the given initial affinity.

If `initial_affinity` is `true` the Affinity will map to all worker threads, otherwise it will map to no worker threads.

Concurrency

Not thread-safe

Parameters

initial_affinity	The initial affinity to set.

6.4.3.3 void embb::mtapi::Affinity::Set (mtapi_uint_t *worker,* bool *state*)

Sets affinity to the given worker.

Concurrency

Not thread-safe

Parameters

worker	The worker to set affinity to.
state	The state of the affinity.

6.4.3.4 bool embb::mtapi::Affinity::Get (mtapi_uint_t *worker*)

Gets affinity to the given worker.

Returns

true, if the Affinity maps to the worker, false otherwise.

Concurrency

Thread-safe and wait-free

Parameters

worker	The worker to get affinity of.

6.4.3.5 mtapi_affinity_t embb::mtapi::Affinity::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_affinity_t.

Concurrency

Thread-safe and wait-free

6.5 embb::base::Allocatable Class Reference

Overloaded new/delete operators.

```
#include <memory_allocation.h>
```

Static Public Member Functions

- static void * operator new (size_t size)

  New operator.
- static void operator delete (void *ptr, size_t size)

  Delete operator.
- static void * operator new[ ] (size_t size)

  Array new operator.
- static void operator delete[ ] (void *ptr, size_t size)

  Array delete operator.

6.5.1 Detailed Description

Overloaded new/delete operators.

Classes that derive from this class will use the EMBB methods for dynamic allocation and deallocation of memory (Allocation::Allocate() and Allocation::Free()). In debug mode, memory consumption is tracked in order to detect memory leaks.

See also

CacheAlignedAllocatable

6.5.2 Member Function Documentation

6.5.2.1 static void∗ embb::base::Allocatable::operator new (size_t *size*) [static]

New operator.

Allocates `size` bytes of memory. Must not be called directly!

Returns

Pointer to allocated block of memory

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Concurrency

Thread-safe

Dynamic memory allocation

See Allocation::Allocate()

See also

operator delete()

Parameters

in	*size*	Size of the memory block in bytes

6.5.2.2 static void embb::base::Allocatable::operator delete (void ∗ *ptr*, size_t *size*) [static]

Delete operator.

Deletes `size` bytes of memory pointed to by `ptr`. Must not be called directly!

Concurrency

Thread-safe

See also

operator new()

Parameters

`in,out`	*ptr*	Pointer to memory block to be freed
`in`	*size*	Size of the memory block in bytes

6.5.2.3 static void* embb::base::Allocatable::operator new[] (size_t *size*) `[static]`

Array new operator.

Allocates an array of `size` bytes. Must not be called directly!

Remarks

Note that the global new[], calling this function, might return a different address. This is stated in the standard (5.3.4 New [expr.new]):
"A new-expression passes the amount of space requested to the allocation function as the first argument of type std::size_t. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array."
So, even if the returned pointer of this function is aligned, the pointer to the array returned by global new[] need not be. For example, when using GCC 4.8.3 (64 bit), the size of the array is kept in the first 8 bytes of the allocated memory.

Returns

Pointer to allocated block of memory

Concurrency

Thread-safe

Dynamic memory allocation

See Allocation::Allocate()

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

See also

operator delete[ ]()

Parameters

`in`	*size*	Size of the array in bytes

6.5.2.4 static void embb::base::Allocatable::operator delete[ ] (void ∗ *ptr*, size_t *size*) [static]

Array delete operator.

Deletes array of `size` bytes pointed to by `ptr`. Must not be called directly!

Concurrency

Thread-safe

See also

operator new[ ]()

Parameters

in,out	*ptr*	Pointer to the array to be freed
in	*size*	Size of the array in bytes

6.6 embb::base::Allocation Class Reference

Common (static) functionality for unaligned and aligned memory allocation.

`#include <memory_allocation.h>`

Static Public Member Functions

- template<typename Type >
  static Type ∗ New ()

  Allocates memory for an instance of type `Type` and default-initializes it.
- template<typename Type , typename Arg1 , ... >
  static Type ∗ New (Arg1 argument1,...)

  Allocates memory unaligned for an instance of type `Type` and initializes it with the specified arguments.
- template<typename Type >
  static void Delete (Type ∗to_delete)

  Destructs an instance of type `Type` and frees the allocated memory.
- static size_t AllocatedBytes ()

  Returns the total number of bytes currently allocated.
- static void ∗ Allocate (size_t size)

  Allocates `size` bytes of memory (unaligned).
- static void Free (void ∗ptr)

  Frees memory that has been allocated by Allocation::Allocate() for some pointer `ptr`.
- static void ∗ AllocateAligned (size_t alignment, size_t size)

  Allocates `size` bytes of memory with alignment `alignment`.
- static void FreeAligned (void ∗ptr)

  Frees memory that has been allocated by Allocation::AllocateAligned() or Allocation::AllocateCacheAligned() for some pointer `ptr`.
- static void ∗ AllocateCacheAligned (size_t size)

  Allocates `size` bytes of cache-aligned memory.

6.6.1 Detailed Description

Common (static) functionality for unaligned and aligned memory allocation.

This class is a wrapper for the functions in embb/base/c/memory_allocation.h

6.6.2 Member Function Documentation

6.6.2.1 template<typename Type > static Type* embb::base::Allocation::New ( ) [static]

Allocates memory for an instance of type `Type` and default-initializes it.

Keeps track of allocated memory in debug mode.

Returns

Pointer to new instance of type `Type`

Exceptions

embb::base::NoMemoryException	if not enough memory is available for the given type.

See also

Delete()

Dynamic memory allocation

`size+3*sizeof(size_t)` bytes in debug mode, otherwise `size` bytes

Concurrency

Thread-safe

Template Parameters

Type	Type of the object to be allocated

6.6.2.2 template<typename Type , typename Arg1 , ... > static Type* embb::base::Allocation::New (Arg1 *argument1*, ...) [static]

Allocates memory unaligned for an instance of type `Type` and initializes it with the specified arguments.

Keeps track of allocated memory in debug mode.

Returns

Pointer to new instance of type `Type`

Exceptions

embb::base::NoMemoryException	if not enough memory is available for the given type.

See also

Delete()

Dynamic memory allocation

`size+3*sizeof(size_t)` bytes in debug mode, otherwise `size` bytes

Concurrency

Thread-safe

Template Parameters

Type	Type of the instance to be allocated
Arg1	Type of (first) constructor argument

Parameters

`in`	*argument1*	(First) argument for constructor of `Type`

6.6.2.3 template<typename Type > static void embb::base::Allocation::Delete (Type * *to_delete*) [static]

Destructs an instance of type `Type` and frees the allocated memory.

Template Parameters

Type	Type of instance to be deleted

Parameters

`in,out`	*to_delete*	Instance to be deleted

6.6.2.4 static size_t embb::base::Allocation::AllocatedBytes ( ) [static]

Returns the total number of bytes currently allocated.

Wrapper for C function embb_get_bytes_allocated().

Returns

Number of currently allocated bytes in debug mode, otherwise 0.

Concurrency

Thread-safe and wait-free

6.6.2.5 static void* embb::base::Allocation::Allocate (size_t *size*) [static]

Allocates `size` bytes of memory (unaligned).

Wrapper for C function embb_allocate().

Keeps track of allocated memory in debug mode.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Dynamic memory allocation

`size+3*sizeof(size_t)` bytes in debug mode, otherwise `size` bytes

Concurrency

Thread-safe

Note

Memory allocated using this function must be freed using Allocation::Free().

See also

AllocateAligned(), AllocateCacheAligned(), Free()

Parameters

in	*size*	Size of memory block to be allocated in bytes

6.6.2.6 static void embb::base::Allocation::Free (void * *ptr*) [static]

Frees memory that has been allocated by Allocation::Allocate() for some pointer `ptr`.

Wrapper for C function embb_free().

Keeps track of freed memory in debug mode.

Concurrency

Thread-safe

See also

Allocate()

Parameters

in,out	*ptr*	Pointer to memory block to be freed

6.6.2.7 static void* embb::base::Allocation::AllocateAligned (size_t *alignment,* size_t *size*) [static]

Allocates size bytes of memory with alignment alignment.

Wrapper for C function embb_alloc_aligned().

This function can be used to align objects to certain boundaries such as cache lines, memory pages, etc.

Keeps track of allocated memory in debug mode.

It is not required that size is a multiple of alignment as, e.g., for the aligned_alloc function of the C11 Standard.

Precondition

The alignment has to be power of 2 and a multiple of size(void*).

Postcondition

The returned pointer is a multiple of alignment.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Dynamic memory allocation

Debug mode: Let n be the number of aligned cells necessary to fit the payload. Then, (n+1)*alignment+3*size↩
_of(size_t)-1 bytes are allocated.
Release mode: size bytes are requested using the functions provided by the operating systems.

Concurrency

Thread-safe

Note

Memory allocated using this function must be freed using Allocation::FreeAligned().

See also

Allocate(), AllocateCacheAligned(), FreeAligned()

Parameters

in	*alignment*	Alignment in bytes
in	*size*	Size of memory block to be allocated in bytes

6.6.2.8 static void embb::base::Allocation::FreeAligned (void * *ptr*) [static]

Frees memory that has been allocated by Allocation::AllocateAligned() or Allocation::AllocateCacheAligned() for some pointer `ptr`.

Wrapper for C function embb_free_aligned().

Keeps track of freed memory in debug mode.

Concurrency

Thread-safe

See also

AllocateAligned(), AllocateCacheAligned()

Parameters

in,out	*ptr*	Pointer to memory block to be freed

6.6.2.9 static void* embb::base::Allocation::AllocateCacheAligned (size_t *size*) [static]

Allocates `size` bytes of cache-aligned memory.

Wrapper for C function embb_alloc_cache_aligned().

Specialized version of Allocation::AllocateAligned(). The alignment is chosen automatically (usually 64 bytes).

Keeps track of allocated memory in debug mode.

Postcondition

The returned pointer is a multiple of the cache line size.

Returns

NULL in case of failure, otherwise address of allocated memory block.

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Dynamic memory allocation

See Allocation::AllocateAligned()

Concurrency

Thread-safe

Note

Memory allocated using this function must be freed using Allocation::FreeAligned().

See also

Allocate(), AllocateAligned(), FreeAligned()

Parameters

in	*size*	Size of memory block to be allocated in bytes

6.7 embb::base::Allocator< Type > Class Template Reference

Allocator according to the C++ standard.

```
#include <memory_allocation.h>
```

Classes

- struct rebind

  Rebind allocator to type OtherType.

Public Types

- typedef size_t size_type

  Quantity of elements type.
- typedef ptrdiff_t difference_type

  Difference between two pointers type.
- typedef Type ∗ pointer

  Pointer to element type.
- typedef const Type ∗ const_pointer

  Pointer to constant element type.
- typedef Type & reference

  Reference to element type.
- typedef const Type & const_reference

  Reference to constant element type.
- typedef Type value_type

  Element type.

Public Member Functions

- Allocator () throw ()

    Constructs allocator object.
- Allocator (const Allocator &) throw ()

    Copies allocator object.
- template< typename OtherType >
  Allocator (const Allocator< OtherType > &) throw ()

    Constructs allocator object.
- ∼Allocator () throw ()

    Destructs allocator object.
- pointer address (reference x) const

    Gets address of an object.
- const_pointer address (const_reference x) const

    Gets address of a constant object.
- pointer allocate (size_type n, const void ∗=0)

    Allocates but doesn't initialize storage for elements of type Type.
- void deallocate (pointer p, size_type)

    Deallocates storage of destroyed elements.
- size_type max_size () const throw ()

    Allocation maximum
- void construct (pointer p, const value_type &val)

    Initializes elements of allocated storage with specified value.
- void destroy (pointer p)

    Destroys elements of initialized storage.

6.7.1 Detailed Description

template<typename Type>
class embb::base::Allocator< Type >

Allocator according to the C++ standard.

For memory allocation and deallocation, embb::base::Allocation::Allocate() and embb::base::Allocation::Free() are used, respectively.

In debug mode, leak checking is active. The function embb::base::Allocation::AllocatedBytes() returns the number of currently allocated bytes.

6.7.2 Member Typedef Documentation

6.7.2.1 template<typename Type> typedef size_t embb::base::Allocator< Type >::size_type

Quantity of elements type.

6.7.2.2 template<typename Type> typedef ptrdiff_t embb::base::Allocator< Type >::difference_type

Difference between two pointers type.

6.7.2.3 template<typename Type> typedef Type* embb::base::Allocator< Type >::pointer

Pointer to element type.

6.7.2.4 template<typename Type> typedef const Type* embb::base::Allocator< Type >::const_pointer

Pointer to constant element type.

6.7.2.5 template<typename Type> typedef Type& embb::base::Allocator< Type >::reference

Reference to element type.

6.7.2.6 template<typename Type> typedef const Type& embb::base::Allocator< Type >::const_reference

Reference to constant element type.

6.7.2.7 template<typename Type> typedef Type embb::base::Allocator< Type >::value_type

Element type.

6.7.3 Constructor & Destructor Documentation

6.7.3.1 template<typename Type> embb::base::Allocator< Type >::Allocator ( ) throw)

Constructs allocator object.

6.7.3.2 template<typename Type> embb::base::Allocator< Type >::Allocator (const Allocator< Type > &) throw)

Copies allocator object.

6.7.3.3 template<typename Type> template<typename OtherType > embb::base::Allocator< Type >::Allocator (const Allocator< OtherType > &) throw)

Constructs allocator object.

Allows construction from allocators for different types (rebind)

6.7.3.4 template<typename Type> embb::base::Allocator< Type >::~Allocator ( ) throw)

Destructs allocator object.

6.7.4 Member Function Documentation

6.7.4.1 template<typename Type> pointer embb::base::Allocator< Type >::address (reference *x*) const

Gets address of an object.

Returns

Address of object

Concurrency

Thread-safe and wait-free

Parameters

in	*x*	Reference to object

6.7.4.2 template<typename Type> const_pointer embb::base::Allocator< Type >::address (const_reference *x*) const

Gets address of a constant object.

Returns

Address of object

Concurrency

Thread-safe and wait-free

Parameters

in	*x*	Reference to constant object

6.7.4.3 template<typename Type> pointer embb::base::Allocator< Type >::allocate (size_type *n*, const void * = 0)

Allocates but doesn't initialize storage for elements of type Type.

Concurrency

Thread-safe

Returns

Pointer to allocated storage

Dynamic memory allocation

See Allocation::Allocate()

Parameters

in	*n*	Number of elements to allocate

6.7.4.4 template<typename Type> void embb::base::Allocator< Type >::deallocate (pointer *p*, size_type)

Deallocates storage of destroyed elements.

Concurrency

Thread-safe

Parameters

`in,out`	*p*	Pointer to allocated storage

6.7.4.5 template<typename Type> size_type embb::base::Allocator< Type >::max_size ( ) const throw)

Allocation maximum

Returns

Maximum number of elements that can be allocated

Concurrency

Thread-safe and wait-free

6.7.4.6 template<typename Type> void embb::base::Allocator< Type >::construct (pointer *p*, const value_type & *val*)

Initializes elements of allocated storage with specified value.

Concurrency

Thread-safe

Parameters

`in,out`	*p*	Pointer to allocated storage
`in`	*val*	Value

6.7.4.7 template<typename Type> void embb::base::Allocator< Type >::destroy (pointer *p*)

Destroys elements of initialized storage.

Concurrency

Thread-safe

Parameters

`in,out`	*p*	Pointer to allocated storage

6.8 embb::base::AllocatorCacheAligned< Type > Class Template Reference

Allocator according to the C++ standard.

```
#include <memory_allocation.h>
```

Classes

- struct rebind

  Rebind allocator to type OtherType.

Public Types

- typedef size_t size_type

  Quantity of elements type.
- typedef ptrdiff_t difference_type

  Difference between two pointers type.
- typedef Type ∗ pointer

  Pointer to element type.
- typedef const Type ∗ const_pointer

  Pointer to constant element type.
- typedef Type & reference

  Reference to element type.
- typedef const Type & const_reference

  Reference to constant element type.
- typedef Type value_type

  Element type.

Public Member Functions

- AllocatorCacheAligned () throw ()

  Constructs allocator object.
- AllocatorCacheAligned (const AllocatorCacheAligned &a) throw ()

  Copies allocator object.
- template<typename OtherType >
  AllocatorCacheAligned (const AllocatorCacheAligned< OtherType > &) throw ()

  Constructs allocator object.
- ∼AllocatorCacheAligned () throw ()

  Destructs allocator object.
- pointer allocate (size_type n, const void ∗=0)

  Allocates but doesn't initialize storage for elements of type Type.
- void deallocate (pointer p, size_type)

  Deallocates storage of destroyed elements.
- pointer address (reference x) const

  Gets address of an object.
- const_pointer address (const_reference x) const

  Gets address of a constant object.
- size_type max_size () const throw ()

  Allocation maximum
- void construct (pointer p, const value_type &val)

  Initializes elements of allocated storage with specified value.
- void destroy (pointer p)

  Destroys elements of initialized storage.

6.8.1 Detailed Description

template<typename Type>
class embb::base::AllocatorCacheAligned< Type >

Allocator according to the C++ standard.

Allocates memory cache-aligned.

For memory allocation and deallocation, embb::base::Allocation::AllocateCacheAligned() and embb::base::Allocation::FreeAligned() are used, respectively.

In debug mode, leak checking is active. The function embb::base::Allocation::AllocatedBytes() returns the number of currently allocated bytes.

6.8.2 Member Typedef Documentation

6.8.2.1 template<typename Type> typedef size_t embb::base::AllocatorCacheAligned< Type >::size_type

Quantity of elements type.

6.8.2.2 template<typename Type> typedef ptrdiff_t embb::base::AllocatorCacheAligned< Type >::difference_type

Difference between two pointers type.

6.8.2.3 template<typename Type> typedef Type* embb::base::AllocatorCacheAligned< Type >::pointer

Pointer to element type.

6.8.2.4 template<typename Type> typedef const Type* embb::base::AllocatorCacheAligned< Type >::const_pointer

Pointer to constant element type.

6.8.2.5 template<typename Type> typedef Type& embb::base::AllocatorCacheAligned< Type >::reference

Reference to element type.

6.8.2.6 template<typename Type> typedef const Type& embb::base::AllocatorCacheAligned< Type >::const_reference

Reference to constant element type.

6.8.2.7 template<typename Type> typedef Type embb::base::AllocatorCacheAligned< Type >::value_type

Element type.

6.8.3 Constructor & Destructor Documentation

6.8.3.1 template<typename Type> embb::base::AllocatorCacheAligned< Type >::AllocatorCacheAligned ( ) throw)

Constructs allocator object.

6.8.3.2 template<typename Type> embb::base::AllocatorCacheAligned< Type >::AllocatorCacheAligned (const AllocatorCacheAligned< Type > & *a*) throw)

Copies allocator object.

Parameters

in	*a*	Other allocator object

6.8.3.3 template<typename Type> template<typename OtherType > embb::base::AllocatorCacheAligned< Type >::AllocatorCacheAligned (const AllocatorCacheAligned< OtherType > &) throw)

Constructs allocator object.

Allows construction from allocators for different types (rebind)

6.8.3.4 template<typename Type> embb::base::AllocatorCacheAligned< Type >::~AllocatorCacheAligned ( ) throw)

Destructs allocator object.

6.8.4 Member Function Documentation

6.8.4.1 template<typename Type> pointer embb::base::AllocatorCacheAligned< Type >::allocate (size_type *n*, const void * = 0)

Allocates but doesn't initialize storage for elements of type Type.

Concurrency

Thread-safe

Returns

Pointer to allocated storage

Dynamic memory allocation

see Allocation::Allocate()

Parameters

in	*n*	Number of elements to allocate

6.8.4.2 template<typename Type> void embb::base::AllocatorCacheAligned< Type >::deallocate (pointer *p*, size_type)

Deallocates storage of destroyed elements.

Concurrency

Thread-safe

Parameters

in,out	*p*	Pointer to allocated storage

6.8.4.3 template<typename Type> pointer embb::base::Allocator< Type >::address (reference *x*) const [inherited]

Gets address of an object.

Returns

Address of object

Concurrency

Thread-safe and wait-free

Parameters

x	Reference to object

6.8.4.4 template<typename Type> const_pointer embb::base::Allocator< Type >::address (const_reference *x*) const [inherited]

Gets address of a constant object.

Returns

Address of object

Concurrency

Thread-safe and wait-free

Parameters

x	Reference to constant object

6.8.4.5 template< typename Type > size_type embb::base::Allocator< Type >::max_size ( ) const throw) [inherited]

Allocation maximum

Returns

Maximum number of elements that can be allocated

Concurrency

Thread-safe and wait-free

6.8.4.6 template< typename Type > void embb::base::Allocator< Type >::construct (pointer *p*, const value_type & *val*) [inherited]

Initializes elements of allocated storage with specified value.

Concurrency

Thread-safe

Parameters

p	Pointer to allocated storage
val	Value

6.8.4.7 template< typename Type > void embb::base::Allocator< Type >::destroy (pointer *p*) [inherited]

Destroys elements of initialized storage.

Concurrency

Thread-safe

Parameters

p	Pointer to allocated storage

6.9 embb::base::Atomic< BaseType > Class Template Reference

Class representing atomic variables.

```
#include <atomic.h>
```

Public Member Functions

- Atomic ()
  Default constructor.
- Atomic (BaseType val)
  Valued-based constructor.
- BaseType operator= (BaseType val)
  Assignment operator.
- operator BaseType () const
  Type conversion.
- bool IsArithmetic () const
  Predicate representing support for arithmetic operations.
- bool IsInteger () const
  Predicate representing integers.
- bool IsPointer () const
  Predicate representing pointers.
- void Store (BaseType val)
  Store operation.
- BaseType Load () const
  Load operation.
- BaseType Swap (BaseType val)
  Swap operation.
- bool CompareAndSwap (BaseType &expected, BaseType desired)
  Compare-and-Swap operation (CAS).

Arithmetic members

The following members are only available if `BaseType` supports arithmetic operations (integer and non-void pointer types).

- BaseType FetchAndAdd (BaseType val)
  Fetch-and-Add operation.
- BaseType FetchAndSub (BaseType val)
  Fetch-and-Sub operation.
- BaseType operator++ (int)
  Post-increment operation.
- BaseType operator-- (int)
  Post-decrement operation.
- BaseType operator++ ()
  Pre-increment operation.
- BaseType operator-- ()
  Pre-decrement operation.
- BaseType operator+= (BaseType val)
  Assignment by sum operation.
- BaseType operator-= (BaseType val)
  Assignment by difference operation.

Integer members

The following members are only available if `BaseType` is an integer type.

- void operator&= (BaseType val)
  Assignment by bitwise AND.
- void operator|= (BaseType val)
  Assignment by bitwise OR.
- void operator^= (BaseType val)
  Assignment by bitwise XOR.

Pointer members

The following members are only available if `BaseType` is a non-void pointer type.

- BaseType ∗ operator-> ()
  Structure dereference operation.
- BaseType & operator∗ ()
  Dereference operation.

6.9.1 Detailed Description

template<typename BaseType>
class embb::base::Atomic< BaseType >

Class representing atomic variables.

The current implementation guarantees sequential consistency (full fences) for all atomic operations. Relaxed memory models may be added in the future.

Template Parameters

BaseType	Underlying type

6.9.2 Constructor & Destructor Documentation

6.9.2.1 template<typename BaseType> embb::base::Atomic< BaseType >::Atomic ( )

Default constructor.

Constructs an atomic variable holding zero.

Concurrency

Thread-safe and wait-free

See also

Atomic(BaseType)

6.9.2.2 template<typename BaseType> embb::base::Atomic< BaseType >::Atomic (BaseType *val*) [explicit]

Valued-based constructor.

Constructs an atomic variable holding the passed value.

Parameters

val	Initial value

Concurrency

Thread-safe and wait-free

Note

There is intentionally no copy constructor, since two different memory locations cannot be manipulated atomically.

See also

Atomic()

6.9.3 Member Function Documentation

6.9.3.1 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator= (BaseType *val*)

Assignment operator.

Assigns the passed value to the object.

Concurrency

Thread-safe and wait-free

Parameters

val	The value to assign

Returns

A shallow copy of this object

6.9.3.2 template<typename BaseType> embb::base::Atomic< BaseType >::operator BaseType ( ) const

Type conversion.

Returns the value of the object. Equivalent to Load().

Concurrency

Thread-safe and wait-free

Returns

Stored value

See also

Load()

6.9.3.3 template<typename BaseType> bool embb::base::Atomic< BaseType >::IsArithmetic ( ) const

Predicate representing support for arithmetic operations.

Returns `true` if type `BaseType` supports arithmetic operations, otherwise `false`. Only integers and non-void pointers support arithmetic operations.

Concurrency

Thread-safe and wait-free

Returns

Boolean value indicating support for arithmetic operations

See also

IsInteger(), IsPointer()

6.9.3.4 template<typename BaseType> bool embb::base::Atomic< BaseType >::IsInteger ( ) const

Predicate representing integers.

Returns `true` if `BaseType` is an integer type, otherwise `false`.

Concurrency

Thread-safe and wait-free

Returns

Boolean value indicating whether `BaseType` is an integer

See also

IsArithmetic(), IsPointer()

6.9.3.5 template<typename BaseType> bool embb::base::Atomic< BaseType >::IsPointer ( ) const

Predicate representing pointers.

Returns `true` if `BaseType` is a non-void pointer type, otherwise `false`.

Concurrency

Thread-safe and wait-free

Returns

Boolean value indicating whether `BaseType` is a non-void pointer type

See also

IsArithmetic(), IsInteger()

6.9.3.6 template<typename BaseType> void embb::base::Atomic< BaseType >::Store (BaseType *val*)

Store operation.

Stores the passed value in the object. Equivalent to assignment operator, except that `Store` does not return anything.

Concurrency

Thread-safe and wait-free

Parameters

val	Value to be stored

See also

Load()

6.9.3.7 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::Load ( ) const

Load operation.

Loads and returns the stored value. Equivalent to type conversion.

Concurrency

Thread-safe and wait-free

Returns

Stored value

See also

Store()

6.9.3.8 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::Swap (BaseType *val*)

Swap operation.

Stores the given value in the object and returns the old value.

Concurrency

Thread-safe and wait-free

Parameters

val	New value

Returns

Old value

See also

CompareAndSwap()

6.9.3.9 template<typename BaseType> bool embb::base::Atomic< BaseType >::CompareAndSwap (BaseType & *expected*, BaseType *desired*)

Compare-and-Swap operation (CAS).

Stores `desired` if the current value is equal to `expected`. Otherwise, stores the current value in `expected`.

Concurrency

Thread-safe and wait-free

Parameters

expected	Expected value
desired	Desired value

Returns

`true` if CAS succeeded, otherwise `false`

See also

Swap()

6.9.3.10 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::FetchAndAdd (BaseType *val*)

Fetch-and-Add operation.

Adds the passed value and returns the old value.

Concurrency

Thread-safe and wait-free

Parameters

val	Addend

Returns

Old value

See also

FetchAndSub()

6.9.3.11 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::FetchAndSub (BaseType *val*)

Fetch-and-Sub operation.

Subtracts the passed value and returns the old value.

Concurrency

Thread-safe and wait-free

Parameters

val	Subtrahend

Returns

Old value

See also

FetchAndAdd()

6.9.3.12 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator++ (int)

Post-increment operation.

Increments the value and returns the old value.

Concurrency

Thread-safe and wait-free

Returns

Old value

See also

operator++()

6.9.3.13 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator-- (int)

Post-decrement operation.

Decrements the value and returns the old value.

Concurrency

Thread-safe and wait-free

Returns

Old value

See also

operator--()

6.9.3.14 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator++ ( )

Pre-increment operation.

Increments the value and returns the new value.

Concurrency

Thread-safe and wait-free

Returns

New value

See also

operator++(int)

6.9.3.15 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator-- ( )

Pre-decrement operation.

Decrements the value and returns the new value.

Concurrency

Thread-safe and wait-free

Returns

New value

See also

operator--(int)

6.9.3.16 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator+= (BaseType *val*)

Assignment by sum operation.

Adds the passed value and returns the new value.

Parameters

val	Addend

Returns

New value

Concurrency

Thread-safe and wait-free

See also

operator-=()

6.9.3.17 template<typename BaseType> BaseType embb::base::Atomic< BaseType >::operator-= (BaseType *val*)

Assignment by difference operation.

Subtracts the passed value and returns the new value.

Parameters

val	Subtrahend

Returns

New value

Concurrency

Thread-safe and wait-free

See also

operator+=()

6.9.3.18 template<typename BaseType> void embb::base::Atomic< BaseType >::operator&= (BaseType *val*)

Assignment by bitwise AND.

Stores the result of the bitwise AND in the current object. Does not return anything, since this cannot be implemented atomically on all architectures.

Concurrency

Thread-safe and wait-free

Parameters

val	Second operand of bitwise AND

See also

operator|=(), operator^=()

6.9.3.19 template<typename BaseType> void embb::base::Atomic< BaseType >::operator|= (BaseType *val*)

Assignment by bitwise OR.

Stores the result of the bitwise OR in the current object. Does not return anything, since this cannot be implemented atomically on all architectures.

Concurrency

Thread-safe and wait-free

Parameters

val	Second operand of bitwise OR

See also

operator&=(), operator^=()

6.9.3.20 template<typename BaseType> void embb::base::Atomic< BaseType >::operator^= (BaseType *val*)

Assignment by bitwise XOR.

Stores the result of the bitwise XOR in the current object. Does not return anything, since this cannot be implemented atomically on all architectures.

Parameters

val	Second operand of bitwise XOR

Concurrency

Thread-safe and wait-free

See also

operator&=(), operator|=()

6.9.3.21 template<typename BaseType> BaseType* embb::base::Atomic< BaseType >::operator-> ( )

Structure dereference operation.

Used to access an element of an instance of a class or a structure pointed to by the stored pointer.

Returns

Stored pointer

Concurrency

Thread-safe and wait-free

See also

operator*()

6.9.3.22 template<typename BaseType> BaseType& embb::base::Atomic< BaseType >::operator* ( )

Dereference operation.

Used to access the object pointed to by the stored pointer.

Returns

Reference to the object

Concurrency

Thread-safe and wait-free

See also

operator->()

6.10 embb::base::CacheAlignedAllocatable Class Reference

Overloaded new/delete operators.

```
#include <memory_allocation.h>
```

Static Public Member Functions

- static void * operator new (size_t size)

  New operator.
- static void operator delete (void *ptr, size_t size)

  Delete operator.
- static void * operator new[ ] (size_t size)

  Array new operator.
- static void operator delete[ ] (void *ptr, size_t size)

  Array delete operator.

6.10.1 Detailed Description

Overloaded new/delete operators.

Classes that derive from this class will use the EMBB methods for dynamic, cache-aligned allocation and deallocation of memory (Allocation::AllocateCacheAligned() and Allocation::FreeAligned()). In debug mode, memory consumption is tracked in order to detect memory leaks.

Note

When using the new[] operator, not each object in the array is aligned, but only the constructed array as a whole.

See also

Allocatable

6.10.2 Member Function Documentation

6.10.2.1 static void* embb::base::CacheAlignedAllocatable::operator new (size_t *size*) [static]

New operator.

Allocates `size` bytes of memory. Must not be called directly!

Returns

Pointer to allocated block of memory

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Concurrency

Thread-safe

Dynamic memory allocation

See Allocation::AllocateCacheAligned()

See also

operator delete()

Parameters

`in`	*size*	Size of the memory block in bytes

6.10.2.2 static void embb::base::CacheAlignedAllocatable::operator delete (void * *ptr*, size_t *size*) [static]

Delete operator.

Deletes `size` bytes of memory pointed to by `ptr`. Must not be called directly!

Concurrency

Thread-safe

Parameters

`in,out`	*ptr*	Pointer to memory block to be freed
`in`	*size*	Size of the memory block in bytes

6.10.2.3 static void∗ embb::base::CacheAlignedAllocatable::operator new[ ] (size_t *size*) [static]

Array new operator.

Allocates an array of `size` bytes. Must not be called directly!

Returns

Pointer to allocated block of memory

Exceptions

embb::base::NoMemoryException	if not enough memory is available.

Dynamic memory allocation

See Allocation::AllocateCacheAligned()

Concurrency

Thread-safe

See also

operator delete[ ]()

Parameters

`in`	*size*	size of bytes to allocate for the array

6.10.2.4 static void embb::base::CacheAlignedAllocatable::operator delete[ ] (void ∗ *ptr*, size_t *size*) [static]

Array delete operator.

Deletes array of `size` bytes pointed to by `ptr`. Must not be called directly!

Concurrency

Thread-safe

See also

operator new[ ]()

Parameters

`in,out`	*ptr*	Pointer to the array to be freed
`in`	*size*	Size of the array in bytes

6.11 embb::base::ConditionVariable Class Reference

Represents a condition variable for thread synchronization.

```
#include <condition_variable.h>
```

Public Member Functions

- ConditionVariable ()

  Creates a condition variable.
- void NotifyOne ()

  Wakes up one waiting thread.
- void NotifyAll ()

  Wakes up all waiting threads.
- void Wait (UniqueLock< Mutex > &lock)

  Releases the lock and waits until the thread is woken up.
- bool WaitUntil (UniqueLock< Mutex > &lock, const Time &time)

  Releases the lock and waits until the thread is woken up or the specified time point has passed.
- template<typename Tick >
  bool WaitFor (UniqueLock< Mutex > &lock, const Duration< Tick > &duration)

  Releases the lock and waits until the thread is woken up or the specified duration has passed.

6.11.1 Detailed Description

Represents a condition variable for thread synchronization.

Provides an abstraction from platform-specific condition variable implementations. Condition variables can be waited for with timeouts using relative durations and absolute time points.

This class is essentially a wrapper for the underlying C implementation.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 embb::base::ConditionVariable::ConditionVariable ( )

Creates a condition variable.

Exceptions

embb::base::ErrorException	if initialization failed

Dynamic memory allocation

Potentially allocates dynamic memory

Concurrency

Not thread-safe

6.11.3 Member Function Documentation

6.11.3.1 void embb::base::ConditionVariable::NotifyOne ( )

Wakes up one waiting thread.

Exceptions

embb::base::ErrorException	if notification failed

Concurrency

Thread-safe

See also

NotifyAll(), Wait()

6.11.3.2 void embb::base::ConditionVariable::NotifyAll ( )

Wakes up all waiting threads.

Exceptions

embb::base::ErrorException	if notification failed

Concurrency

Thread-safe

See also

NotifyOne(), Wait()

6.11.3.3 void embb::base::ConditionVariable::Wait (UniqueLock< Mutex > & *lock*)

Releases the lock and waits until the thread is woken up.

Precondition

The lock has been acquired by the calling thread.

Postcondition

The lock has been re-acquired by the calling thread.

Exceptions

embb::base::ErrorException	if waiting failed

Concurrency

Thread-safe

See also

NotifyOne(), NotifyAll()

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*lock*	Lock to be released and re-acquired

6.11.3.4 bool embb::base::ConditionVariable::WaitUntil (UniqueLock< Mutex > & *lock*, const Time & *time*)

Releases the lock and waits until the thread is woken up or the specified time point has passed.

Precondition

The lock has been acquired by the calling thread.

Postcondition

The lock has been re-acquired by the calling thread.

Returns

`true` if the thread was woken up before the specified time point has passed, otherwise `false`.

Exceptions

embb::base::ErrorException	if an error occurred

Concurrency

Thread-safe

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*lock*	Lock to be released and re-acquired
`in`	*time*	Absolute time point until which the thread maximally waits

6.11.3.5 template<typename Tick > bool embb::base::ConditionVariable::WaitFor (UniqueLock< Mutex > & *lock,* const Duration< Tick > & *duration*)

Releases the lock and waits until the thread is woken up or the specified duration has passed.

Precondition

The lock has been acquired by the calling thread.

Postcondition

The lock has been re-acquired by the calling thread.

Returns

`true` if the thread was woken up before the specified duration has passed, otherwise `false`.

Exceptions

embb::base::ErrorException	if an error occurred

Concurrency

Thread-safe

Template Parameters

Tick	Type of tick of the duration. See Duration.

Note

It is strongly recommended checking the condition in a loop in order to deal with spurious wakeups and situations where another thread has locked the mutex between notification and wakeup.

Parameters

`in,out`	*lock*	Lock to be released and re-acquired
`in`	*duration*	Relative time duration the thread maximally waits

6.12 embb::dataflow::Network::ConstantSource< Type > Class Template Reference

Constant source process template.

`#include <network.h>`

Public Types

- typedef Outputs< OUTPUT_TYPE_LIST > OutputsType
  Output port type list.

Public Member Functions

- ConstantSource (Network &network, Type value)
  Constructs a ConstantSource with a value to emit on each token.
- ConstantSource (Network &network, Type value, embb::mtapi::ExecutionPolicy const &policy)
  Constructs a ConstantSource with a value to emit on each token.
- virtual bool HasInputs () const
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)
  Connects output port 0 to input port 0 of `target`.

6.12.1 Detailed Description

template<typename Type>
class embb::dataflow::Network::ConstantSource< Type >

Constant source process template.

A constant source has one output port and emits a constant value given at construction time for each token.

Template Parameters

Type	The type of output port 0.

6.12.2 Member Typedef Documentation

6.12.2.1 template<typename Type > typedef Outputs<OUTPUT_TYPE_LIST> embb::dataflow::Network::Constant↩Source< Type >::OutputsType

Output port type list.

6.12.3 Constructor & Destructor Documentation

6.12.3.1 template< typename Type > embb::dataflow::Network::ConstantSource< Type >::ConstantSource (Network & *network,* Type *value*)

Constructs a ConstantSource with a value to emit on each token.

Parameters

network	The network this node is going to be part of.
value	The value to emit.

6.12.3.2 template< typename Type > embb::dataflow::Network::ConstantSource< Type >::ConstantSource (Network & *network,* Type *value,* embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a ConstantSource with a value to emit on each token.

Parameters

network	The network this node is going to be part of.
value	The value to emit.
policy	The execution policy of the process.

6.12.4 Member Function Documentation

6.12.4.1 template< typename Type > virtual bool embb::dataflow::Network::ConstantSource< Type >::HasInputs ( ) const [virtual]

Returns

Always `false`.

6.12.4.2 template< typename Type > virtual bool embb::dataflow::Network::ConstantSource< Type >::HasOutputs ( ) const [virtual]

Returns

Always `true`.

6.12.4.3 template< typename Type > OutputsType& embb::dataflow::Network::ConstantSource< Type >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.12.4.4 template<typename Type > template<int Index> OutputsType::Types<Index>::Result& embb::dataflow::Network::ConstantSource< Type >::GetOutput ( )

Returns

Output port at Index.

6.12.4.5 template<typename Type > template<typename T > void embb::dataflow::Network::ConstantSource< Type >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.13 embb::base::CoreSet Class Reference

Represents a set of processor cores, used to set thread-to-core affinities.

```
#include <core_set.h>
```

Public Member Functions

- CoreSet ()
  Constructs an empty core set.
- CoreSet (bool value)
  Constructs a core set with all or no cores.
- CoreSet (const CoreSet &to_copy)
  Constructs a copy of the specified core set.
- CoreSet & operator= (const CoreSet &to_assign)
  Assigns an existing core set.
- void Reset (bool value)
  Resets the core set according to the specified value.
- void Add (unsigned int core)
  Adds one core to the core set.
- void Remove (unsigned int core)
  Removes one core from the core set.
- bool IsContained (unsigned int core) const
  Checks whether the specified core is included in the set.
- unsigned int Count () const
  Counts the number of cores in the set.

- CoreSet operator& (const CoreSet &rhs) const

    Intersects this core set with the specified one.
- CoreSet operator| (const CoreSet &rhs) const

    Unites this core set with the specified one.
- CoreSet & operator&= (const CoreSet &rhs)

    Intersects this core set with the specified one and overwrites this core set.
- CoreSet & operator|= (const CoreSet &rhs)

    Unites this core set with the specified one an overwrites this core set.
- embb_core_set_t const & GetInternal () const

    Provides access to internal representation to use it with C API.

Static Public Member Functions

- static unsigned int CountAvailable ()

    Returns the number of available processor cores.

6.13.1 Detailed Description

Represents a set of processor cores, used to set thread-to-core affinities.

An instance of this type represents a subset of processor cores. Core sets can be used to set thread-to-core affinities. A core in a core set might just represent a logical core (hyper-thread), depending on the underlying hardware. Each core is identified by a unique integer starting with 0. For example, the cores of a quad-core system are represented by the set {0,1,2,3}.

This class is essentially a wrapper for the underlying C implementation.

Concurrency

Not thread-safe

6.13.2 Constructor & Destructor Documentation

6.13.2.1 embb::base::CoreSet::CoreSet ( )

Constructs an empty core set.

6.13.2.2 embb::base::CoreSet::CoreSet (bool *value*) [explicit]

Constructs a core set with all or no cores.

Parameters

in	*value*	`true` includes all cores in the set, `false` excludes all

6.13.2.3 embb::base::CoreSet::CoreSet (const CoreSet & *to_copy*)

Constructs a copy of the specified core set.

Parameters

in	*to_copy*	Core set to copy

6.13.3 Member Function Documentation

6.13.3.1 static unsigned int embb::base::CoreSet::CountAvailable ( ) [static]

Returns the number of available processor cores.

If the processor supports hyper-threading, each hyper-thread is treated as a separate processor core.

Returns

Number of cores including hyper-threads

6.13.3.2 CoreSet& embb::base::CoreSet::operator= (const CoreSet & *to_assign*)

Assigns an existing core set.

Returns

Reference to `*this`

Parameters

in	*to_assign*	Core set to assign

6.13.3.3 void embb::base::CoreSet::Reset (bool *value*)

Resets the core set according to the specified value.

Parameters

in	*value*	`true` includes all cores in the set, `false` excludes all

6.13.3.4 void embb::base::CoreSet::Add (unsigned int *core*)

Adds one core to the core set.

Parameters

in	*core*	Core to add (from 0 to number of cores - 1)

6.13.3.5 void embb::base::CoreSet::Remove (unsigned int *core*)

Removes one core from the core set.

Parameters

in	*core*	Core to remove (from 0 to number of cores - 1)

6.13.3.6 bool embb::base::CoreSet::IsContained (unsigned int *core*) const

Checks whether the specified core is included in the set.

Returns

`true` if core is included, otherwise `false`

Parameters

in	*core*	Core to check (from 0 to number of cores - 1)

6.13.3.7 unsigned int embb::base::CoreSet::Count ( ) const

Counts the number of cores in the set.

Returns

Number of cores in the set

6.13.3.8 CoreSet embb::base::CoreSet::operator& (const CoreSet & *rhs*) const

Intersects this core set with the specified one.

This core set is not modified by the operation.

Returns

Copy of the result

Parameters

in	*rhs*	Core set on right-hand side of intersection operation

6.13.3.9 CoreSet embb::base::CoreSet::operator\| (const CoreSet & *rhs*) const

Unites this core set with the specified one.

This core set is not modified by the operation.

Returns

Copy of the result

Parameters

in	*rhs*	Core set on right-hand side of union operation

6.13.3.10 CoreSet& embb::base::CoreSet::operator&= (const CoreSet & *rhs*)

Intersects this core set with the specified one and overwrites this core set.

Returns

Reference to `*this`

Parameters

in	*rhs*	Core set on right-hand side of intersection operation

6.13.3.11 CoreSet& embb::base::CoreSet::operator\|= (const CoreSet & *rhs*)

Unites this core set with the specified one an overwrites this core set.

Returns

Reference to `*this`

Parameters

in	*rhs*	Core set on right-hand side of union operation

6.13.3.12 embb_core_set_t const& embb::base::CoreSet::GetInternal ( ) const

Provides access to internal representation to use it with C API.

Returns

A reference to the internal embb_core_set_t structure.

6.14 embb::base::DeferLockTag Struct Reference

Tag type for deferred UniqueLock construction.

`#include <mutex.h>`

6.14.1 Detailed Description

Tag type for deferred UniqueLock construction.

Use the defer_lock variable in constructor calls.

6.15 embb::base::Duration< Tick > Class Template Reference

Represents a relative time duration for a given tick type.

`#include <duration.h>`

Public Member Functions

- Duration ()
  Constructs a duration of length zero.
- Duration (unsigned long long ticks)
  Constructs a duration with given number of ticks.
- Duration (const Duration< Tick > &to_copy)
  Constructs a duration by copying from an existing duration.
- Duration< Tick > & operator= (const Duration< Tick > &to_assign)
  Assigns an existing duration.
- unsigned long long Count () const
  Returns the number of ticks of the duration.
- Duration< Tick > & operator+= (const Duration< Tick > &rhs)
  Assignment by addition of another duration with same tick type.

Static Public Member Functions

- static const Duration< Tick > & Zero ()
  Returns duration of length zero.
- static const Duration< Tick > & Max ()
  Returns duration with maximum ticks representable by implementation.
- static const Duration< Tick > & Min ()
  Returns duration with minimum ticks representable by implementation.

6.15.1 Detailed Description

template<typename Tick>
class embb::base::Duration< Tick >

Represents a relative time duration for a given tick type.

Concurrency

Not thread-safe

Note

The typedefs DurationSeconds, DurationMilliseconds, DurationMicroseconds, and DurationNanoseconds provide directly usable duration types.

Template Parameters

Tick	Possible tick types are Seconds, Milliseconds, Microseconds, Nanoseconds

6.15.2 Constructor & Destructor Documentation

6.15.2.1 template<typename Tick> embb::base::Duration< Tick >::Duration ( )

Constructs a duration of length zero.

6.15.2.2 template<typename Tick> embb::base::Duration< Tick >::Duration (unsigned long long *ticks*) [explicit]

Constructs a duration with given number of ticks.

Parameters

in	*ticks*	Number of ticks

6.15.2.3 template<typename Tick> embb::base::Duration< Tick >::Duration (const Duration< Tick > & *to_copy*)

Constructs a duration by copying from an existing duration.

Parameters

in	*to_copy*	Duration to copy

6.15.3 Member Function Documentation

6.15.3.1 template<typename Tick> static const Duration<Tick>& embb::base::Duration< Tick >::Zero ( ) [static]

Returns duration of length zero.

Returns

Duration of length zero

6.15.3.2 template<typename Tick> static const Duration<Tick>& embb::base::Duration< Tick >::Max ( ) [static]

Returns duration with maximum ticks representable by implementation.

This value depends on the tick type and on the platform.

Returns

Reference to duration with maximum value

6.15.3.3 template<typename Tick> static const Duration<Tick>& embb::base::Duration< Tick >::Min ( ) [static]

Returns duration with minimum ticks representable by implementation.

This value depends on the tick type and on the platform.

Returns

Reference to duration with minimum value

6.15.3.4 template<typename Tick> Duration<Tick>& embb::base::Duration< Tick >::operator= (const Duration< Tick > & *to_assign*)

Assigns an existing duration.

Returns

Reference to `*this`

Parameters

in	*to_assign*	Duration to assign

6.15.3.5 template<typename Tick> unsigned long long embb::base::Duration< Tick >::Count ( ) const

Returns the number of ticks of the duration.

Returns

Number of ticks of the duration

6.15.3.6 template<typename Tick> Duration<Tick>& embb::base::Duration< Tick >::operator+= (const Duration< Tick > & *rhs*)

Assignment by addition of another duration with same tick type.

Returns

Reference to `*this`

Parameters

in	*rhs*	Duration to add to this duration

6.16 embb::base::ErrorException Class Reference

Indicates a general error.

`#include <exceptions.h>`

Public Member Functions

- ErrorException (const char *message)
  Constructs an exception with the specified message.
- virtual int Code () const
  Returns an integer code representing the exception.
- virtual const char * What () const throw ()
  Returns the error message.

6.16.1 Detailed Description

Indicates a general error.

6.16.2 Constructor & Destructor Documentation

6.16.2.1 embb::base::ErrorException::ErrorException (const char * *message*) [explicit]

Constructs an exception with the specified message.

Parameters

in	*message*	Error message

6.16.3 Member Function Documentation

6.16.3.1 virtual int embb::base::ErrorException::Code ( ) const [virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implements embb::base::Exception.

6.16.3.2 virtual const char∗ embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.17 embb::base::Exception Class Reference

Abstract base class for exceptions.

`#include <exceptions.h>`

Public Member Functions

- Exception (const char ∗message)

  Constructs an exception with a custom message.
- virtual ∼Exception () throw ()

  Destructs the exception.
- Exception (const Exception &e)

  Constructs an exception by copying from an existing one.
- Exception & operator= (const Exception &e)

  Assigns an existing exception.
- virtual const char ∗ What () const throw ()

  Returns the error message.
- virtual int Code () const =0

  Returns an integer code representing the exception.

6.17.1 Detailed Description

Abstract base class for exceptions.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 embb::base::Exception::Exception (const char ∗ *message*) [explicit]

Constructs an exception with a custom message.

Parameters

in	*message*	Error message

6.17.2.2 virtual embb::base::Exception::∼Exception ( ) throw) [virtual]

Destructs the exception.

6.17.2.3 embb::base::Exception::Exception (const Exception & *e*)

Constructs an exception by copying from an existing one.

Parameters

in	*e*	Exception to be copied

6.17.3 Member Function Documentation

6.17.3.1 Exception& embb::base::Exception::operator= (const Exception & *e*)

Assigns an existing exception.

Returns

Reference to `*this`

Parameters

in	*e*	Exception to assign

6.17.3.2 virtual const char∗ embb::base::Exception::What ( ) const throw) [virtual]

Returns the error message.

Returns

Pointer to error message

6.17.3.3 virtual int embb::base::Exception::Code ( ) const [pure virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implemented in embb::base::ErrorException, embb::base::OverflowException, embb::base::UnderflowException, embb::base::ResourceBusyException, embb::base::NoMemoryException, and embb::mtapi::StatusException.

6.18 embb::mtapi::ExecutionPolicy Class Reference

Describes the execution policy of a parallel algorithm.

```
#include <execution_policy.h>
```

Public Member Functions

- ExecutionPolicy ()

  Constructs the default execution policy.
- ExecutionPolicy (bool initial_affinity, mtapi_uint_t priority)

  Constructs an execution policy with the specified affinity and priority.
- ExecutionPolicy (mtapi_uint_t priority)

  Constructs an execution policy with the specified priority.
- ExecutionPolicy (bool initial_affinity)

  Constructs an execution policy with the specified affinity.
- void AddWorker (mtapi_uint_t worker)

  Sets affinity to a specific worker thread.
- void RemoveWorker (mtapi_uint_t worker)

  Removes affinity to a specific worker thread.
- bool IsSetWorker (mtapi_uint_t worker)

  Checks if affinity to a specific worker thread is set.
- unsigned int GetCoreCount () const

  Returns the number of cores the policy is affine to.
- mtapi_affinity_t GetAffinity () const

  Returns the affinity.
- mtapi_uint_t GetPriority () const

  Returns the priority.

6.18.1 Detailed Description

Describes the execution policy of a parallel algorithm.

The execution policy comprises

- the affinity of tasks to MTAPI worker threads (not CPU cores) and
- the priority of the spawned tasks.

The priority is a number between 0 (denoting the highest priority) to max_priorities - 1 as given during initialization using Node::Initialize(). The default value of max_priorities is 4.

6.18.2 Constructor & Destructor Documentation

6.18.2.1 embb::mtapi::ExecutionPolicy::ExecutionPolicy ( )

Constructs the default execution policy.

Sets the affinity to all worker threads and the priority to the default value.

Concurrency

Not thread-safe

6.18.2.2 embb::mtapi::ExecutionPolicy::ExecutionPolicy (bool *initial_affinity*, mtapi_uint_t *priority*)

Constructs an execution policy with the specified affinity and priority.

Concurrency

Not thread-safe

Parameters

in	*initial_affinity*	`true` sets the affinity to all worker threads, `false` to no worker threads.
in	*priority*	Priority for the execution policy.

6.18.2.3 embb::mtapi::ExecutionPolicy::ExecutionPolicy (mtapi_uint_t *priority*) [explicit]

Constructs an execution policy with the specified priority.

Sets the affinity to all worker threads.

Concurrency

Not thread-safe

Parameters

in	*priority*	Priority for the execution policy.

6.18.2.4 embb::mtapi::ExecutionPolicy::ExecutionPolicy (bool *initial_affinity*) [explicit]

Constructs an execution policy with the specified affinity.

Sets the priority to the default value.

Concurrency

Not thread-safe

Parameters

in	*initial_affinity*	`true` sets the affinity to all worker threads, `false` to no worker threads.

6.18.3 Member Function Documentation

6.18.3.1 void embb::mtapi::ExecutionPolicy::AddWorker (mtapi_uint_t *worker*)

Sets affinity to a specific worker thread.

Concurrency

Not thread-safe

Parameters

in	*worker*	Worker thread index

6.18.3.2 void embb::mtapi::ExecutionPolicy::RemoveWorker (mtapi_uint_t *worker*)

Removes affinity to a specific worker thread.

Concurrency

Not thread-safe

Parameters

in	*worker*	Worker thread index

6.18.3.3 bool embb::mtapi::ExecutionPolicy::IsSetWorker (mtapi_uint_t *worker*)

Checks if affinity to a specific worker thread is set.

Returns

`true` if affinity is set, otherwise `false`

Concurrency

Thread-safe and wait-free

Parameters

in	*worker*	Worker thread index

6.18.3.4 unsigned int embb::mtapi::ExecutionPolicy::GetCoreCount ( ) const

Returns the number of cores the policy is affine to.

Returns

the number of cores

Concurrency

Thread-safe and wait-free

6.18.3.5 mtapi_affinity_t embb::mtapi::ExecutionPolicy::GetAffinity ( ) const

Returns the affinity.

Returns

the affinity

Concurrency

Thread-safe and wait-free

6.18.3.6 mtapi_uint_t embb::mtapi::ExecutionPolicy::GetPriority ( ) const

Returns the priority.

Returns

the priority

Concurrency

Thread-safe and wait-free

6.19 embb::base::Function< ReturnType,... > Class Template Reference

Wraps function pointers, member function pointers, and functors with up to five arguments.

```
#include <function.h>
```

Public Member Functions

- template<class ClassType >
  Function (ClassType const &obj)

  Constructor from functor.
- Function (ReturnType(∗func)(...))

  Constructor from function pointer with return type ReturnType and up to five arguments.
- template<class ClassType >
  Function (ClassType &obj, ReturnType(ClassType::∗func)(...))

  Constructor from object and member function pointer with return type ReturnType and up to five arguments.
- Function (Function const &func)

  Copy constructor.
- ∼Function ()

  Destructor.
- void operator= (ReturnType(∗func)(...))

  Assigns this object a new function pointer.
- void operator= (Function &func)

  Assigns this object another Function.
- template<class C >
  void operator= (C const &obj)

  Assigns this object a new functor.
- ReturnType operator() (...)

  Calls the wrapped function with the given parameters.

6.19.1 Detailed Description

template<typename ReturnType, ...>
class embb::base::Function< ReturnType,... >

Wraps function pointers, member function pointers, and functors with up to five arguments.

6.19.2 Constructor & Destructor Documentation

6.19.2.1 template<typename ReturnType, ... > template<class ClassType > embb::base::Function< ReturnType,... >::Function (ClassType const & *obj*) [explicit]

Constructor from functor.

Uses operator() with return type ReturnType and up to five arguments. Copies the functor.

Dynamic memory allocation

Allocates memory for the copy of the functor.

Parameters

obj	The functor to wrap.

6.19.2.2 template<typename ReturnType, ... > embb::base::Function< ReturnType,... >::Function (ReturnType(∗)(...) *func*) [explicit]

Constructor from function pointer with return type ReturnType and up to five arguments.

Parameters

func	The function pointer.

6.19.2.3 template<typename ReturnType, ... > template<class ClassType > embb::base::Function< ReturnType,... >::Function (ClassType & *obj*, ReturnType(ClassType::∗)(...) *func*)

Constructor from object and member function pointer with return type ReturnType and up to five arguments.

Parameters

obj	Reference to object.
func	Member function pointer.

6.19.2.4 template<typename ReturnType, ... > embb::base::Function< ReturnType,... >::Function (Function< ReturnType,... > const & *func*)

Copy constructor.

Parameters

func	The Function to copy.

6.19.2.5 template<typename ReturnType, ... > embb::base::Function< ReturnType,... >::∼Function ( )

Destructor.

6.19.3 Member Function Documentation

6.19.3.1 template<typename ReturnType, ... > void embb::base::Function< ReturnType,... >::operator= (ReturnType(∗)(...) *func*)

Assigns this object a new function pointer.

Parameters

func	The function pointer.

6.19.3.2 template<typename ReturnType, ... > void embb::base::Function< ReturnType,... >::operator= (Function< ReturnType,... > & *func*)

Assigns this object another Function.

Parameters

func	The Function.

6.19.3.3 template<typename ReturnType, ... > template<class C > void embb::base::Function< ReturnType,... >::operator= (C const & *obj*)

Assigns this object a new functor.

The functor is copied.

Parameters

obj	The functor.

6.19.3.4 template<typename ReturnType, ... > ReturnType embb::base::Function< ReturnType,... >::operator() (...)

Calls the wrapped function with the given parameters.

Returns

A value generated by the wrapped function.

6.20 embb::mtapi::Group Class Reference

Represents a facility to wait for multiple related Tasks.

```
#include <group.h>
```

Public Member Functions

- Group ()
  Constructs an invalid Group.
- Group (Group const &other)
  Copies a Group.
- Group & operator= (Group const &other)
  Copies a Group.
- void Delete ()
  Deletes a Group object.

- template<typename ARGS , typename RES >
  Task Start (mtapi_task_id_t task_id, Job const &job, const ARGS *arguments, RES *results, TaskAttributes const &attributes)

  Starts a new Task in this Group.
- template<typename ARGS , typename RES >
  Task Start (mtapi_task_id_t task_id, Job const &job, const ARGS *arguments, RES *results)

  Starts a new Task in this Group.
- template<typename ARGS , typename RES >
  Task Start (Job const &job, const ARGS *arguments, RES *results, TaskAttributes const &attributes)

  Starts a new Task in this Group.
- template<typename ARGS , typename RES >
  Task Start (Job const &job, const ARGS *arguments, RES *results)

  Starts a new Task in this Group.
- mtapi_status_t WaitAny (mtapi_timeout_t timeout, void **result)

  Waits for any Task in the Group to finish for `timeout` milliseconds and retrieves the result buffer given in Start().
- mtapi_status_t WaitAny (void **result)

  Waits for any Task in the Group to finish and retrieves the result buffer given in Start().
- mtapi_status_t WaitAny (mtapi_timeout_t timeout)

  Waits for any Task in the Group to finish for `timeout` milliseconds.
- mtapi_status_t WaitAny ()

  Waits for any Task in the Group to finish.
- mtapi_status_t WaitAll (mtapi_timeout_t timeout)

  Waits for all Task in the Group to finish for `timeout` milliseconds.
- mtapi_status_t WaitAll ()

  Waits for all Task in the Group to finish.
- mtapi_group_hndl_t GetInternal () const

  Returns the internal representation of this object.

6.20.1 Detailed Description

Represents a facility to wait for multiple related Tasks.

6.20.2 Constructor & Destructor Documentation

6.20.2.1 embb::mtapi::Group::Group ( )

Constructs an invalid Group.

Concurrency

Thread-safe and wait-free

6.20.2.2 embb::mtapi::Group::Group (Group const & *other*)

Copies a Group.

Concurrency

Thread-safe and wait-free

Parameters

other	Group to copy

6.20.3 Member Function Documentation

6.20.3.1 Group& embb::mtapi::Group::operator= (Group const & *other*)

Copies a Group.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

other	Group to copy

6.20.3.2 void embb::mtapi::Group::Delete ( )

Deletes a Group object.

Concurrency

Thread-safe

6.20.3.3 template<typename ARGS , typename RES > Task embb::mtapi::Group::Start (mtapi_task_id_t *task_id*, Job const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*, TaskAttributes const & *attributes*)

Starts a new Task in this Group.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.20.3.4 template<typename ARGS , typename RES > **Task** embb::mtapi::Group::Start (mtapi_task_id_t *task_id*, **Job** const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*)

Starts a new Task in this Group.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.

6.20.3.5 template<typename ARGS , typename RES > **Task** embb::mtapi::Group::Start (**Job** const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*, **TaskAttributes** const & *attributes*)

Starts a new Task in this Group.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.20.3.6 template<typename ARGS , typename RES > **Task** embb::mtapi::Group::Start (**Job** const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*)

Starts a new Task in this Group.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.

6.20.3.7 mtapi_status_t embb::mtapi::Group::WaitAny (mtapi_timeout_t *timeout*, void ** *result*)

Waits for any Task in the Group to finish for `timeout` milliseconds and retrieves the result buffer given in Start().

Returns

The status of the Task that finished execution, `MTAPI_TIMEOUT` or `MTAPI_ERR_*`

Concurrency

Thread-safe

Parameters

`in`	*timeout*	Timeout duration in milliseconds
`out`	*result*	The result buffer given in Node::Start, Group::Start or Queue::Enqueue

6.20.3.8 mtapi_status_t embb::mtapi::Group::WaitAny (void ** *result*)

Waits for any Task in the Group to finish and retrieves the result buffer given in Start().

Returns

The status of the Task that finished execution or `MTAPI_ERR_*`

Concurrency

Thread-safe

Parameters

`out`	*result*	The result buffer given in Node::Start, Group::Start or Queue::Enqueue

6.20.3.9 mtapi_status_t embb::mtapi::Group::WaitAny (mtapi_timeout_t *timeout*)

Waits for any Task in the Group to finish for `timeout` milliseconds.

Returns

The status of the Task that finished execution

Concurrency

Thread-safe

Parameters

`in`	*timeout*	Timeout duration in milliseconds

6.20.3.10 mtapi_status_t embb::mtapi::Group::WaitAny ( )

Waits for any Task in the Group to finish.

Returns

The status of the Task that finished execution

Concurrency

Thread-safe

6.20.3.11 mtapi_status_t embb::mtapi::Group::WaitAll (mtapi_timeout_t *timeout*)

Waits for all Task in the Group to finish for `timeout` milliseconds.

Returns

`MTAPI_SUCCESS`, `MTAPI_TIMEOUT`, `MTAPI_ERR_*` or the status of any failed Task

Concurrency

Thread-safe

Parameters

`in`	*timeout*	Timeout duration in milliseconds

6.20.3.12 mtapi_status_t embb::mtapi::Group::WaitAll ( )

Waits for all Task in the Group to finish.

Returns

MTAPI_SUCCESS, MTAPI_TIMEOUT, MTAPI_ERR_* or the status of any failed Task

Concurrency

Thread-safe

6.20.3.13 mtapi_group_hndl_t embb::mtapi::Group::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_group_hndl_t.

Concurrency

Thread-safe and wait-free

6.21 embb::mtapi::GroupAttributes Class Reference

Contains attributes of a Group.

`#include <group_attributes.h>`

Public Member Functions

- GroupAttributes ()

  Constructs a GroupAttributes object.
- mtapi_group_attributes_t const & GetInternal () const

  Returns the internal representation of this object.

6.21.1 Detailed Description

Contains attributes of a Group.

6.21.2 Constructor & Destructor Documentation

6.21.2.1 embb::mtapi::GroupAttributes::GroupAttributes ( )

Constructs a GroupAttributes object.

Concurrency

Not thread-safe

6.21.3 Member Function Documentation

6.21.3.1 mtapi_group_attributes_t const& embb::mtapi::GroupAttributes::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A reference to the internal mtapi_group_attributes_t structure.

Concurrency

Thread-safe and wait-free

6.22 embb::base::Thread::ID Class Reference

Unique ID of a thread that can be compared with other IDs.

```
#include <thread.h>
```

Public Member Functions

- ID ()

  Constructs an empty (invalid) thread ID.

Friends

- template<class CharT , class Traits >
  std::basic_ostream< CharT, Traits > & operator<< (std::basic_ostream< CharT, Traits > &os, Thread::ID id)

  The streaming operator needs to access the internal ID representation.
- bool operator== (Thread::ID lhs, Thread::ID rhs)

  Comparison operators need to access the internal ID representation.
- bool operator!= (Thread::ID lhs, Thread::ID rhs)

  Compares two thread IDs for inequality.

6.22.1 Detailed Description

Unique ID of a thread that can be compared with other IDs.

6.22.2 Constructor & Destructor Documentation

6.22.2.1 embb::base::Thread::ID::ID ( )

Constructs an empty (invalid) thread ID.

6.22.3 Friends And Related Function Documentation

6.22.3.1 template<class CharT , class Traits > std::basic_ostream<CharT, Traits>& operator<< (std::basic_ostream< CharT, Traits > & *os*, Thread::ID *id*) [friend]

The streaming operator needs to access the internal ID representation.

Returns

Reference to the stream

Parameters

in,out	*os*	Stream to which thread ID is written
in	*id*	Thread ID to be written

6.22.3.2 bool operator== (Thread::ID *lhs*, Thread::ID *rhs*) [friend]

Comparison operators need to access the internal ID representation.

Returns

`true` if thread IDs are equivalent, otherwise `false`

Parameters

in	*lhs*	Left-hand side of equality sign
in	*rhs*	Right-hand side of equality sign

6.22.3.3 bool operator!= (Thread::ID *lhs*, Thread::ID *rhs*) [friend]

Compares two thread IDs for inequality.

Returns

`true` if thread IDs are not equivalent, otherwise `false`

Parameters

in	*lhs*	Left-hand side of inequality sign
in	*rhs*	Left-hand side of inequality sign

6.23 embb::algorithms::Identity Struct Reference

Unary identity functor.

`#include <identity.h>`

Public Member Functions

- template<typename Type >
  Type & operator() (Type &value)

  Returns `value` *unchanged.*
- template<typename Type >
  const Type & operator() (const Type &value)

  Returns `value` *unchanged.*

6.23.1 Detailed Description

Unary identity functor.

6.23.2 Member Function Documentation

6.23.2.1 template<typename Type > Type& embb::algorithms::Identity::operator() (Type & *value*)

Returns `value` unchanged.

Returns

`value`

Template Parameters

Type	Any type

Parameters

`in`	*value*	Value that is returned unchanged

6.23.2.2 template<typename Type > const Type& embb::algorithms::Identity::operator() (const Type & *value*)

Returns `value` unchanged.

Returns

`value`

Template Parameters

Type	Any type

Parameters

in	*value*	Value that is returned unchanged

6.24 embb::dataflow::Network::In< Type > Class Template Reference

Input port class.

```
#include <network.h>
```

6.24.1 Detailed Description

template<typename Type>
class embb::dataflow::Network::In< Type >

Input port class.

6.25 embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 > Struct Template Reference

Provides the input port types for a process.

```
#include <network.h>
```

Classes

- struct Types
  Type list used to derive input port types from Index.

Public Member Functions

- template<int Index>
  Types< Index >::Result & Get ()

6.25.1 Detailed Description

template<typename T1, typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil>
struct embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 >

Provides the input port types for a process.

Template Parameters

T1	Type of first port.
T2	Optional type of second port.
T3	Optional type of third port.
T4	Optional type of fourth port.
T5	Optional type of fifth port.

6.25.2 Member Function Documentation

6.25.2.1 template<typename T1 , typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil> template<int Index> Types<Index>::Result& embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 >::Get ( )

Returns

Reference to input port at Index.

6.26 embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator Class Reference

Forward iterator to iterate over the allocated elements of the pool.

```
#include <wait_free_array_value_pool.h>
```

Public Member Functions

- Iterator ()

  Constructs an invalid iterator.
- Iterator (Iterator const &other)

  Copies an iterator.
- Iterator & operator= (Iterator const &other)

  Copies an iterator.
- Iterator & operator++ ()

  Pre-increments an iterator.
- Iterator operator++ (int)

  Post-increments an iterator.
- bool operator== (Iterator const &rhs)

  Compares two iterators for equality.
- bool operator!= (Iterator const &rhs)

  Compares two iterators for inequality.
- std::pair< int, Type > operator* ()

  Dereferences the iterator.

6.26.1 Detailed Description

template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >>
class embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator

Forward iterator to iterate over the allocated elements of the pool.

Note

Iterators are invalidated by any change to the pool (Allocate and Free calls).

6.26.2 Constructor & Destructor Documentation

6.26.2.1 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::Iterator ( )

Constructs an invalid iterator.

Concurrency

Thread-safe and wait-free

6.26.2.2 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::Iterator (Iterator const & *other*)

Copies an iterator.

Concurrency

Thread-safe and wait-free

Parameters

in	*other*	Iterator to copy.

6.26.3 Member Function Documentation

6.26.3.1 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> Iterator& embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator= (Iterator const & *other*)

Copies an iterator.

Returns

Reference to this iterator.

Concurrency

Thread-safe and wait-free

Parameters

in	*other*	Iterator to copy.

6.26.3.2 template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic< Type > > > Iterator& embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator++ ( )

Pre-increments an iterator.

Returns

Reference to this iterator.

Concurrency

Not thread-safe

6.26.3.3 template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic< Type > > > Iterator embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator++ (int)

Post-increments an iterator.

Returns

Copy of this iterator before increment.

Concurrency

Not thread-safe

6.26.3.4 template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic< Type > > > bool embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator== (Iterator const & *rhs*)

Compares two iterators for equality.

Returns

`true`, if the two iterators are equal, `false` otherwise.

Concurrency

Thread-safe and wait-free

Parameters

in	*rhs*	Iterator to compare to.

6.26.3.5 template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> bool embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator!= (Iterator const & *rhs*)

Compares two iterators for inequality.

Returns

`true`, if the two iterators are not equal, `false` otherwise.

Concurrency

Thread-safe and wait-free

Parameters

in	*rhs*	Iterator to compare to.

6.26.3.6 template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> std::pair<int, Type> embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Iterator::operator∗ ( )

Dereferences the iterator.

Returns

A pair consisting of index and value of the element pointed to.

Concurrency

Thread-safe and wait-free

6.27 embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator Class Reference

Forward iterator to iterate over the allocated elements of the pool.

`#include <lock_free_tree_value_pool.h>`

Public Member Functions

- Iterator ()

  Constructs an invalid iterator.
- Iterator (Iterator const &other)

  Copies an iterator.
- Iterator & operator= (Iterator const &other)

  Copies an iterator.
- Iterator & operator++ ()

  Pre-increments an iterator.
- Iterator operator++ (int)

  Post-increments an iterator.
- bool operator== (Iterator const &rhs)

  Compares two iterators for equality.
- bool operator!= (Iterator const &rhs)

  Compares two iterators for inequality.
- std::pair< int, Type > operator* ()

  Dereferences the iterator.

6.27.1 Detailed Description

template<typename Type, Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >>
class embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator

Forward iterator to iterate over the allocated elements of the pool.

Note

Iterators are invalidated by any change to the pool (Allocate and Free calls).

6.27.2 Constructor & Destructor Documentation

6.27.2.1 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::Iterator ( )

Constructs an invalid iterator.

Concurrency

Thread-safe and wait-free

6.27.2.2 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::Iterator (Iterator const & *other*)

Copies an iterator.

Concurrency

Thread-safe and wait-free

Parameters

in	*other*	Iterator to copy.

6.27.3 Member Function Documentation

6.27.3.1 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> Iterator& embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator= (Iterator const & *other*)

Copies an iterator.

Returns

Reference to this iterator.

Concurrency

Thread-safe and wait-free

Parameters

in	*other*	Iterator to copy.

6.27.3.2 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> Iterator& embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator++ ( )

Pre-increments an iterator.

Returns

Reference to this iterator.

Concurrency

Not thread-safe

6.27.3.3 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> Iterator embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator++ (int)

Post-increments an iterator.

Returns

Copy of this iterator before increment.

Concurrency

Not thread-safe

6.27.3.4 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> bool embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator== (Iterator const & *rhs*)

Compares two iterators for equality.

Returns

`true`, if the two iterators are equal, `false` otherwise.

Concurrency

Thread-safe and wait-free

Parameters

`in`	*rhs*	Iterator to compare to.

6.27.3.5 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> bool embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator!= (Iterator const & *rhs*)

Compares two iterators for inequality.

Returns

`true`, if the two iterators are not equal, `false` otherwise.

Concurrency

Thread-safe and wait-free

Parameters

`in`	*rhs*	Iterator to compare to.

6.27.3.6 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> std::pair<int, Type> embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Iterator::operator* ( )

Dereferences the iterator.

Returns

A pair consisting of index and value of the element pointed to.

Concurrency

Thread-safe and wait-free

6.28 embb::mtapi::Job Class Reference

Represents a collection of Actions.

```
#include <job.h>
```

Public Member Functions

- Job ()
  Constructs a Job.
- Job (Job const &other)
  Copies a Job object.
- void operator= (Job const &other)
  Copies a Job object.
- mtapi_job_hndl_t GetInternal () const
  Returns the internal representation of this object.

6.28.1 Detailed Description

Represents a collection of Actions.

6.28.2 Constructor & Destructor Documentation

6.28.2.1 embb::mtapi::Job::Job ( )

Constructs a Job.

The Job object will be invalid.

Concurrency

Thread-safe and wait-free

6.28.2.2 embb::mtapi::Job::Job (Job const & *other*)

Copies a Job object.

Concurrency

Thread-safe and wait-free

Parameters

other	The Job to copy from

6.28.3 Member Function Documentation

6.28.3.1 void embb::mtapi::Job::operator= (Job const & *other*)

Copies a Job object.

Concurrency

Thread-safe and wait-free

Parameters

other	The Job to copy from

6.28.3.2 mtapi_job_hndl_t embb::mtapi::Job::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_job_hndl_t.

Concurrency

Thread-safe and wait-free

6.29 embb::containers::LockFreeMPMCQueue< Type, ValuePool > Class Template Reference

Lock-free queue for multiple producers and multiple consumers.

```
#include <lock_free_mpmc_queue.h>
```

Public Member Functions

- LockFreeMPMCQueue (size_t capacity)
  Creates a queue with the specified capacity.
- ∼LockFreeMPMCQueue ()
  Destroys the queue.
- size_t GetCapacity ()
  Returns the capacity of the queue.
- bool TryEnqueue (Type const &element)
  Tries to enqueue an element into the queue.
- bool TryDequeue (Type &element)
  Tries to dequeue an element from the queue.

6.29.1 Detailed Description

template< typename Type, typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >>
class embb::containers::LockFreeMPMCQueue< Type, ValuePool >

Lock-free queue for multiple producers and multiple consumers.

Implemented concepts:

Queue Concept

See also

WaitFreeSPSCQueue

Template Parameters

Type	Type of the queue elements
ValuePool	Type of the value pool used as basis for the ObjectPool which stores the elements.

6.29.2 Constructor & Destructor Documentation

6.29.2.1 template< typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> embb::containers::LockFreeMPMCQueue< Type, ValuePool >::LockFreeMPMCQueue (size_t *capacity*)

Creates a queue with the specified capacity.

Dynamic memory allocation

Let `t` be the maximum number of threads and `x` be `2.5*t+1`. Then, `x*(3*t+1)` elements of size `sizeof(void*)`, `x` elements of size `sizeof(Type)`, and `capacity+1` elements of size `sizeof(Type)` are allocated.

Concurrency

Not thread-safe

See also

Queue Concept

Parameters

`in`	*capacity*	Capacity of the queue

6.29.2.2 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> embb::containers::LockFreeMPMCQueue< Type, ValuePool >::~LockFreeMPMCQueue ( )

Destroys the queue.

Concurrency

Not thread-safe

6.29.3 Member Function Documentation

6.29.3.1 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> size_t embb::containers::LockFreeMPMCQueue< Type, ValuePool >::GetCapacity ( )

Returns the capacity of the queue.

Returns

Number of elements the queue can hold.

Concurrency

Thread-safe and wait-free

6.29.3.2 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> bool embb::containers::LockFreeMPMCQueue< Type, ValuePool >::TryEnqueue (Type const & *element*)

Tries to enqueue an element into the queue.

Returns

`true` if the element could be enqueued, `false` if the queue is full.

Concurrency

Thread-safe and lock-free

Note

It might be possible to enqueue more elements into the queue than its capacity permits.

See also

Queue Concept

Parameters

in	*element*	Const reference to the element that shall be enqueued

6.29.3.3 template< typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> bool embb::containers::LockFreeMPMCQueue< Type, ValuePool >::TryDequeue (Type & *element*)

Tries to dequeue an element from the queue.

Returns

`true` if an element could be dequeued, `false` if the queue is empty.

Concurrency

Thread-safe and lock-free

See also

Queue Concept

Parameters

`in,out`	*element*	Reference to the dequeued element. Unchanged, if the operation was not successful.

6.30 embb::containers::LockFreeStack< Type, ValuePool > Class Template Reference

Lock-free stack.

`#include <lock_free_stack.h>`

Public Member Functions

- LockFreeStack (size_t capacity)
  Creates a stack with the specified capacity.
- size_t GetCapacity ()
  Returns the capacity of the stack.
- ∼LockFreeStack ()
  Destroys the stack.
- bool TryPush (Type const &element)
  Tries to push an element onto the stack.
- bool TryPop (Type &element)
  Tries to pop an element from the stack.

6.30.1 Detailed Description

template< typename Type, typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >>
class embb::containers::LockFreeStack< Type, ValuePool >

Lock-free stack.

Implemented concepts:

Stack Concept

Template Parameters

Type	Type of the stack elements
ValuePool	Type of the value pool used as basis for the ObjectPool which stores the elements.

6.30.2 Constructor & Destructor Documentation

6.30.2.1 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> embb::containers::LockFreeStack< Type, ValuePool >::LockFreeStack (size_t *capacity*)

Creates a stack with the specified capacity.

Dynamic memory allocation

Let `t` be the maximum number of threads and `x` be `1.25*t+1`. Then, `x*(3*t+1)` elements of size `sizeof(void*)`, `x` elements of size `sizeof(Type)`, and `capacity` elements of size `sizeof(Type)` are allocated.

Concurrency

Not thread-safe

See also

Stack Concept

Parameters

in	*capacity*	Capacity of the stack

6.30.2.2 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> embb::containers::LockFreeStack< Type, ValuePool >::~LockFreeStack ( )

Destroys the stack.

Concurrency

Not thread-safe

6.30.3 Member Function Documentation

6.30.3.1 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> size_t embb::containers::LockFreeStack< Type, ValuePool >::GetCapacity ( )

Returns the capacity of the stack.

Returns

Number of elements the stack can hold.

Concurrency

Thread-safe and wait-free

6.30.3.2 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> bool embb::containers::LockFreeStack< Type, ValuePool >::TryPush (Type const & *element*)

Tries to push an element onto the stack.

Returns

`true` if the element could be pushed, `false` if the stack is full.

Concurrency

Thread-safe and lock-free

Note

It might be possible to push more elements onto the stack than its capacity permits.

See also

Stack Concept

Parameters

`in`	*element*	Const reference to the element that shall be pushed

6.30.3.3 template<typename Type , typename ValuePool = embb::containers::LockFreeTreeValuePool < bool, false >> bool embb::containers::LockFreeStack< Type, ValuePool >::TryPop (Type & *element*)

Tries to pop an element from the stack.

Returns

`true` if an element could be popped, `false` if the stack is empty.

Concurrency

Thread-safe and lock-free

See also

Stack Concept

Parameters

in,out	*element*	Reference to the popped element. Unchanged, if the operation was not successful.

6.31 embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator > Class Template Reference

Lock-free value pool using binary tree construction.

`#include <lock_free_tree_value_pool.h>`

Classes

- class Iterator

  Forward iterator to iterate over the allocated elements of the pool.

Public Member Functions

- Iterator Begin ()

  Gets a forward iterator to the first allocated element in the pool.
- Iterator End ()

  Gets a forward iterator pointing after the last allocated element in the pool.
- template<typename ForwardIterator >
  LockFreeTreeValuePool (ForwardIterator first, ForwardIterator last)

  Constructs a pool and fills it with the elements in the specified range.
- ~LockFreeTreeValuePool ()

  Destructs the pool.
- int Allocate (Type &element)

  Allocates an element from the pool.
- void Free (Type element, int index)

  Returns an element to the pool.

Static Public Member Functions

- static size_t GetMinimumElementCountForGuaranteedCapacity (size_t capacity)

  Due to concurrency effects, a pool might provide less elements than managed by it.

6.31.1 Detailed Description

template<typename Type, Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> > >
class embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >

Lock-free value pool using binary tree construction.

Implemented concepts:

Value Pool Concept

See also

WaitFreeArrayValuePool

Template Parameters

Type	Element type (must support atomic operations such as `int`).
Undefined	Bottom element (cannot be stored in the pool)
PoolAllocator	Allocator used to allocate the pool array
TreeAllocator	Allocator used to allocate the array representing the binary tree.

6.31.2 Constructor & Destructor Documentation

6.31.2.1 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> template<typename ForwardIterator > embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::LockFreeTreeValuePool (ForwardIterator *first*, ForwardIterator *last*)

Constructs a pool and fills it with the elements in the specified range.

Dynamic memory allocation

Let `n = std::distance(first, last))` and `k` be the minimum number such that `n <= 2^k holds`. Then, `((2^k)-1) * sizeof(embb::Atomic<int>) + n*sizeof(embb::Atomic<Type>)` bytes of memory are allocated.

Concurrency

Not thread-safe

See also

Value Pool Concept

Parameters

in	*first*	Iterator pointing to the first element of the range the pool is filled with
in	*last*	Iterator pointing to the last plus one element of the range the pool is filled with

6.31.2.2 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::~LockFreeTreeValuePool ( )

Destructs the pool.

Concurrency

Not thread-safe

6.31.3 Member Function Documentation

6.31.3.1 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> Iterator embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Begin ( )

Gets a forward iterator to the first allocated element in the pool.

Returns

a forward iterator pointing to the first allocated element.

Concurrency

Thread-safe and wait-free

6.31.3.2 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> Iterator embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::End ( )

Gets a forward iterator pointing after the last allocated element in the pool.

Returns

a forward iterator pointing after the last allocated element.

Concurrency

Thread-safe and wait-free

6.31.3.3 template<typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic<Type> >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic<int> >> static size_t embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::GetMinimumElementCountForGuaranteedCapacity (size_t *capacity*) [static]

Due to concurrency effects, a pool might provide less elements than managed by it.

However, usually one wants to guarantee a minimal capacity. The count of elements that must be given to the pool when to guarantee `capacity` elements is computed using this function.

Returns

count of indices the pool has to be initialized with

Parameters

in	*capacity*	count of indices that shall be guaranteed

6.31.3.4 template< typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic< Type > >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic< int > > > int embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Allocate (Type & *element*)

Allocates an element from the pool.

Returns

Index of the element if the pool is not empty, otherwise `-1`.

Concurrency

Thread-safe and lock-free

See also

Value Pool Concept

Parameters

`in,out`	*element*	Reference to the allocated element. Unchanged, if the operation was not successful.

6.31.3.5 template< typename Type , Type Undefined, class PoolAllocator = embb::base::Allocator< embb::base::Atomic< Type > >, class TreeAllocator = embb::base::Allocator < embb::base::Atomic< int > > > void embb::containers::LockFreeTreeValuePool< Type, Undefined, PoolAllocator, TreeAllocator >::Free (Type *element*, int *index*)

Returns an element to the pool.

Note

The element must have been allocated with Allocate().

Concurrency

Thread-safe and lock-free

See also

Value Pool Concept

Parameters

`in`	*element*	Element to be returned to the pool
`in`	*index*	Index of the element as obtained by Allocate()

6.32 embb::base::LockGuard< Mutex > Class Template Reference

Scoped lock (according to the RAII principle) using a mutex.

```
#include <mutex.h>
```

Public Member Functions

- LockGuard (Mutex &mutex)

  Creates the lock and locks the mutex.
- ∼LockGuard ()

  Unlocks the mutex.

6.32.1 Detailed Description

template<typename Mutex = embb::base::Mutex>
class embb::base::LockGuard< Mutex >

Scoped lock (according to the RAII principle) using a mutex.

The mutex is locked on construction and unlocked on leaving the scope of the lock.

Template Parameters

Mutex	Used mutex type. Has to fulfil the Mutex Concept.

See also

UniqueLock

6.32.2 Constructor & Destructor Documentation

6.32.2.1 template<typename Mutex = embb::base::Mutex> embb::base::LockGuard< Mutex >::LockGuard (Mutex & *mutex*) [explicit]

Creates the lock and locks the mutex.

Precondition

The given mutex is unlocked

Concurrency

Not thread-safe

Parameters

in	*mutex*	Mutex to be guarded

6.32.2.2 template<typename Mutex = embb::base::Mutex> embb::base::LockGuard< Mutex >::∼LockGuard ( )

Unlocks the mutex.

6.33 embb::base::Log Class Reference

Simple logging facilities.

```
#include <log.h>
```

Static Public Member Functions

- static void SetLogLevel (embb_log_level_t log_level)
  Sets the global log level.
- static void SetLogFunction (void ∗context, embb_log_function_t func)
  Sets the global logging function.
- static void Write (char const ∗channel, embb_log_level_t log_level, char const ∗message,...)
  Logs a message to the given channel with the specified log level.
- static void Trace (char const ∗channel, char const ∗message,...)
  Logs a message to the given channel with EMBB_LOG_LEVEL_TRACE.
- static void Info (char const ∗channel, char const ∗message,...)
  Logs a message to the given channel with EMBB_LOG_LEVEL_INFO.
- static void Warning (char const ∗channel, char const ∗message,...)
  Logs a message to the given channel with EMBB_LOG_LEVEL_WARNING.
- static void Error (char const ∗channel, char const ∗message,...)
  Logs a message to the given channel with EMBB_LOG_LEVEL_ERROR.

6.33.1 Detailed Description

Simple logging facilities.

6.33.2 Member Function Documentation

6.33.2.1 static void embb::base::Log::SetLogLevel (embb_log_level_t *log_level*) [static]

Sets the global log level.

This determines what messages will be shown, messages with a more detailed log level will be filtered out. The default log level is EMBB_LOG_LEVEL_NONE.

Concurrency

Not thread-safe

Parameters

in	*log_level*	Log level to use for filtering

6.33.2.2 static void embb::base::Log::SetLogFunction (void * *context,* embb_log_function_t *func*) [static]

Sets the global logging function.

The logging function implements the mechanism for transferring log messages to their destination. `context` is a pointer to data the user needs in the function to determine where the messages should go (may be NULL if no additional data is needed). The default logging function is embb_log_write_file() with context set to `stdout`.

See also

embb_log_function_t

Concurrency

Not thread-safe

Parameters

in	*context*	User context to supply as the first parameter of the logging function
in	*func*	The logging function

6.33.2.3 static void embb::base::Log::Write (char const * *channel,* embb_log_level_t *log_level,* char const * *message,* ...) [static]

Logs a message to the given channel with the specified log level.

If the log level is greater than the configured log level for the channel, the message will be ignored.

See also

embb::base::Log::SetLogLevel, embb::base::Log::SetLogFunction

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id for filtering the log later on. Might be NULL, channel identifier will be "global" in that case
in	*log_level*	Log level to use
in	*message*	Message to convey, may use `printf` style formatting

6.33.2.4 static void embb::base::Log::Trace (char const * *channel*, char const * *message*, ...) [static]

Logs a message to the given channel with EMBB_LOG_LEVEL_TRACE.

In non-debug builds, this function does nothing.

See also

embb::base::Log::Write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

6.33.2.5 static void embb::base::Log::Info (char const * *channel*, char const * *message*, ...) [static]

Logs a message to the given channel with EMBB_LOG_LEVEL_INFO.

In non-debug builds, this function does nothing.

See also

embb::base::Log::Write

Concurrency

Thread-safe

Parameters

in	*channel*	User specified channel id
in	*message*	Message to convey, may use `printf` style formatting

6.33.2.6 static void embb::base::Log::Warning (char const * *channel*, char const * *message*, ...) [static]

Logs a message to the given channel with EMBB_LOG_LEVEL_WARNING.

See also

embb::base::Log::Write

Concurrency

Thread-safe

Parameters

`in`	*channel*	User specified channel id
`in`	*message*	Message to convey, may use `printf` style formatting

6.33.2.7 static void embb::base::Log::Error (char const ∗ *channel*, char const ∗ *message*, ...) `[static]`

Logs a message to the given channel with EMBB_LOG_LEVEL_ERROR.

See also

embb::base::Log::Write

Concurrency

Thread-safe

Parameters

`in`	*channel*	User specified channel id
`in`	*message*	Message to convey, may use `printf` style formatting

6.34 mtapi_action_attributes_struct Struct Reference

Action attributes.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_action_attributes_struct mtapi_action_attributes_t

  Action attributes type.

Public Member Functions

- void mtapi_actionattr_init (mtapi_action_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function initializes an action attributes object.
- void mtapi_actionattr_set (mtapi_action_attributes_t ∗attributes, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  This function sets action attribute values in an action attributes object.

Public Attributes

- mtapi_boolean_t global
  stores MTAPI_ACTION_GLOBAL
- mtapi_affinity_t affinity
  stores MTAPI_ACTION_AFFINITY
- mtapi_boolean_t domain_shared
  stores MTAPI_ACTION_DOMAIN_SHARED

6.34.1 Detailed Description

Action attributes.

6.34.2 Member Typedef Documentation

6.34.2.1 typedef struct mtapi_action_attributes_struct mtapi_action_attributes_t

Action attributes type.

6.34.3 Member Function Documentation

6.34.3.1 void mtapi_actionattr_init (mtapi_action_attributes_t * *attributes*, mtapi_status_t * *status*)

This function initializes an action attributes object.

A action attributes object is a container of action attributes, optionally passed to mtapi_action_create() to create an action with non-default attributes.

The application is responsible for allocating the `mtapi_action_attributes_t` object and initializing it with a call to mtapi_actionattr_init(). The application may then call mtapi_actionattr_set() to specify action attribute values. Calls to mtapi_actionattr_init() have no effect on action attributes after the action has been created with mtapi_↩ action_create(). The `mtapi_action_attributes_t` object may safely be deleted by the application after the call to mtapi_action_create().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_actionattr_set(), mtapi_action_create()

Concurrency

Not thread-safe

Parameters

out	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

6.34.3.2 void mtapi_actionattr_set (mtapi_action_attributes_t ∗ *attributes*, const mtapi_uint_t *attribute_num*, const void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

This function sets action attribute values in an action attributes object.

An action attributes object is a container of action attributes, optionally passed to mtapi_action_create() to create an action with non-default attributes.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

Calls to mtapi_actionattr_set() have no effect on action attributes after the action has been created. The `mtapi_action_attributes_t` object may safely be deleted by the application after the call to mtapi_action_create().

MTAPI-defined action attributes:

Attribute num	Description	Data Type	Default
MTAPI_ACTION_GLOBAL	Indicates whether or not this is a globally visible action. Local actions are not shared with other nodes.	mtapi_boolean_t	MTAPI_TRUE
MTAPI_ACTION_AFFINITY	Core affinity of action code.	mtapi_affinity_t	all cores set
MTAPI_DOMAIN_SHARED	Indicates whether or not the action is shareable across domains.	mtapi_boolean_t	MTAPI_TRUE

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_action_create()

Concurrency

Not thread-safe

Parameters

`in,out`	*attributes*	Pointer to attributes
`in`	*attribute_num*	Attribute id
`in`	*attribute*	Pointer to attribute value
`in`	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

6.34.4 Member Data Documentation

6.34.4.1 mtapi_boolean_t mtapi_action_attributes_struct::global

stores MTAPI_ACTION_GLOBAL

6.34.4.2 mtapi_affinity_t mtapi_action_attributes_struct::affinity

stores MTAPI_ACTION_AFFINITY

6.34.4.3 mtapi_boolean_t mtapi_action_attributes_struct::domain_shared

stores MTAPI_ACTION_DOMAIN_SHARED

6.35 mtapi_action_hndl_struct Struct Reference

Action handle.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_action_hndl_struct mtapi_action_hndl_t

  Action handle type.

Public Attributes

- mtapi_uint_t tag

  version of this handle
- mtapi_action_id_t id

  pool index of this handle

6.35.1 Detailed Description

Action handle.

6.35.2 Member Typedef Documentation

6.35.2.1 typedef struct mtapi_action_hndl_struct mtapi_action_hndl_t

Action handle type.

6.35.3 Member Data Documentation

6.35.3.1 mtapi_uint_t mtapi_action_hndl_struct::tag

version of this handle

6.35.3.2 mtapi_action_id_t mtapi_action_hndl_struct::id

pool index of this handle

6.36 mtapi_ext_job_attributes_struct Struct Reference

Job attributes.

```
#include <mtapi_ext.h>
```

Public Types

- typedef struct mtapi_ext_job_attributes_struct mtapi_ext_job_attributes_t

  Job attributes type.

Public Attributes

- mtapi_ext_problem_size_function_t problem_size_func

  stores MTAPI_JOB_PROBLEM_SIZE_FUNCTION
- mtapi_uint_t default_problem_size

  stores MTAPI_JOB_DEFAULT_PROBLEM_SIZE_SIZE

6.36.1 Detailed Description

Job attributes.

6.36.2 Member Typedef Documentation

6.36.2.1 typedef struct mtapi_ext_job_attributes_struct mtapi_ext_job_attributes_t

Job attributes type.

6.36.3 Member Data Documentation

6.36.3.1 mtapi_ext_problem_size_function_t mtapi_ext_job_attributes_struct::problem_size_func

stores MTAPI_JOB_PROBLEM_SIZE_FUNCTION

6.36.3.2 mtapi_uint_t mtapi_ext_job_attributes_struct::default_problem_size

stores MTAPI_JOB_DEFAULT_PROBLEM_SIZE_SIZE

6.37 mtapi_group_attributes_struct Struct Reference

Group attributes.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_group_attributes_struct mtapi_group_attributes_t

  Group attributes type.

Public Member Functions

- void mtapi_groupattr_init (mtapi_group_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function initializes a group attributes object.
- void mtapi_groupattr_set (mtapi_group_attributes_t ∗attributes, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  This function sets group attribute values in a group attributes object.

Public Attributes

- mtapi_int_t some_value

  just a placeholder

6.37.1 Detailed Description

Group attributes.

6.37.2 Member Typedef Documentation

6.37.2.1 typedef struct mtapi_group_attributes_struct mtapi_group_attributes_t

Group attributes type.

6.37.3 Member Function Documentation

6.37.3.1 void mtapi_groupattr_init (mtapi_group_attributes_t ∗ *attributes*, mtapi_status_t ∗ *status*)

This function initializes a group attributes object.

A group attributes object is a container of group attributes. It is an optional argument passed to mtapi_group_create() to specify non-default group attributes when creating a task group.

To set group attributes to non-default values, the application must allocate a group attributes object of type `mtapi_group_attributes_t` and initialize it with a call to mtapi_groupattr_init(). The application may call mtapi_groupattr_set() to specify attribute values. Calls to mtapi_groupattr_init() have no effect on group attributes after the group has been created. The `mtapi_group_attributes_t` object may safely be deleted by the application after the call to mtapi_group_create().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_group_create(), mtapi_groupattr_set()

Concurrency

Not thread-safe

Parameters

out	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

6.37.3.2 void mtapi_groupattr_set (mtapi_group_attributes_t ∗ *attributes*, const mtapi_uint_t *attribute_num*, const void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

This function sets group attribute values in a group attributes object.

A group attributes object is a container of group attributes, optionally passed to mtapi_group_create() to specify non-default group attributes when creating a task group.

`attributes` is a pointer to a group attributes object that was previously initialized with a call to mtapi_groupattr_init(). Calls to mtapi_groupattr_set() have no effect on group attributes after the group has been created. The group attributes object may safely be deleted by the application after the call to mtapi_group_create().

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_group_create(), mtapi_groupattr_init()

Concurrency

Not thread-safe

Parameters

in,out	*attributes*	Pointer to attributes
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be MTAPI_NULL

6.37.4 Member Data Documentation

6.37.4.1 mtapi_int_t mtapi_group_attributes_struct::some_value

just a placeholder

6.38 mtapi_group_hndl_struct Struct Reference

Group handle.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_group_hndl_struct mtapi_group_hndl_t

  Group handle type.

Public Attributes

- mtapi_uint_t tag

  version of this handle
- mtapi_group_id_t id

  pool index of this handle

6.38.1 Detailed Description

Group handle.

6.38.2 Member Typedef Documentation

6.38.2.1 typedef struct mtapi_group_hndl_struct mtapi_group_hndl_t

Group handle type.

6.38.3 Member Data Documentation

6.38.3.1 mtapi_uint_t mtapi_group_hndl_struct::tag

version of this handle

6.38.3.2 mtapi_group_id_t mtapi_group_hndl_struct::id

pool index of this handle

6.39 mtapi_info_struct Struct Reference

Info structure.

```
#include <mtapi.h>
```

Public Types

- typedef struct mtapi_info_struct mtapi_info_t
  Info type.

Public Attributes

- mtapi_uint_t mtapi_version
  The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.
- mtapi_uint_t organization_id
  Implementation vendor or organization ID.
- mtapi_uint_t implementation_version
  The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.
- mtapi_uint_t number_of_domains
  Number of domains allowed by the implementation.
- mtapi_uint_t number_of_nodes
  Number of nodes allowed by the implementation.
- mtapi_uint_t hardware_concurrency
  Number of CPU cores available.
- mtapi_uint_t used_memory
  Bytes of memory used by MTAPI.

6.39.1 Detailed Description

Info structure.

6.39.2 Member Typedef Documentation

6.39.2.1 typedef struct mtapi_info_struct mtapi_info_t

Info type.

6.39.3 Member Data Documentation

6.39.3.1 mtapi_uint_t mtapi_info_struct::mtapi_version

The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.

6.39.3.2 mtapi_uint_t mtapi_info_struct::organization_id

Implementation vendor or organization ID.

6.39.3.3 mtapi_uint_t mtapi_info_struct::implementation_version

The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.

6.39.3.4 mtapi_uint_t mtapi_info_struct::number_of_domains

Number of domains allowed by the implementation.

6.39.3.5 mtapi_uint_t mtapi_info_struct::number_of_nodes

Number of nodes allowed by the implementation.

6.39.3.6 mtapi_uint_t mtapi_info_struct::hardware_concurrency

Number of CPU cores available.

6.39.3.7 mtapi_uint_t mtapi_info_struct::used_memory

Bytes of memory used by MTAPI.

6.40 mtapi_job_hndl_struct Struct Reference

Job handle.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_job_hndl_struct mtapi_job_hndl_t
    Job handle type.

Public Attributes

- mtapi_uint_t tag
    version of this handle
- mtapi_job_id_t id
    pool index of this handle

6.40.1 Detailed Description

Job handle.

6.40.2 Member Typedef Documentation

6.40.2.1 typedef struct mtapi_job_hndl_struct mtapi_job_hndl_t

Job handle type.

6.40.3 Member Data Documentation

6.40.3.1 mtapi_uint_t mtapi_job_hndl_struct::tag

version of this handle

6.40.3.2 mtapi_job_id_t mtapi_job_hndl_struct::id

pool index of this handle

6.41 mtapi_node_attributes_struct Struct Reference

Node attributes.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_node_attributes_struct mtapi_node_attributes_t

  Node attributes type.

Public Member Functions

- void mtapi_nodeattr_init (mtapi_node_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function initializes a node attributes object.
- void mtapi_nodeattr_set (mtapi_node_attributes_t ∗attributes, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  This function sets node attribute values in a node attributes object.

Public Attributes

- embb_core_set_t core_affinity

  stores MTAPI_NODE_CORE_AFFINITY
- mtapi_uint_t num_cores

  stores MTAPI_NODE_NUMCORES
- mtapi_uint_t type

  stores MTAPI_NODE_TYPE
- mtapi_uint_t max_tasks

  stores MTAPI_NODE_MAX_TASKS
- mtapi_uint_t max_actions

  stores MTAPI_NODE_MAX_ACTIONS
- mtapi_uint_t max_groups

  stores MTAPI_NODE_MAX_GROUPS
- mtapi_uint_t max_queues

  stores MTAPI_NODE_MAX_QUEUES
- mtapi_uint_t queue_limit

  stores MTAPI_NODE_QUEUE_LIMIT
- mtapi_uint_t max_jobs

  stores MTAPI_NODE_MAX_JOBS
- mtapi_uint_t max_actions_per_job

  stores MTAPI_NODE_MAX_ACTIONS_PER_JOB
- mtapi_uint_t max_priorities

  stores MTAPI_NODE_MAX_PRIORITIES
- mtapi_boolean_t reuse_main_thread

  stores MTAPI_NODE_REUSE_MAIN_THREAD
- mtapi_worker_priority_entry_t ∗ worker_priorities

  stores MTAPI_NODE_WORKER_PRIORITIES

6.41.1 Detailed Description

Node attributes.

6.41.2 Member Typedef Documentation

6.41.2.1 typedef struct mtapi_node_attributes_struct mtapi_node_attributes_t

Node attributes type.

6.41.3 Member Function Documentation

6.41.3.1 void mtapi_nodeattr_init (mtapi_node_attributes_t ∗ *attributes*, mtapi_status_t ∗ *status*)

This function initializes a node attributes object.

A node attributes object is a container of node attributes. It is an optional argument passed to mtapi_initialize() to specify non-default node attributes when creating a node.

To set node attributes to non-default values, the application must allocate a node attributes object of type mtapi_↩ node_attributes_t and initialize it with a call to mtapi_nodeattr_init(). The application may call mtapi_nodeattr_set() to specify attribute values. Calls to mtapi_nodeattr_init() have no effect on node attributes after the node has been created and initialized with mtapi_initialize(). The mtapi_node_attributes_t object may safely be deleted by the application after the call to mtapi_nodeattr_init().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.

See also

mtapi_initialize(), mtapi_nodeattr_set()

Concurrency

Not thread-safe

Parameters

out	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

6.41.3.2 void mtapi_nodeattr_set (mtapi_node_attributes_t ∗ *attributes*, const mtapi_uint_t *attribute_num*, const void ∗ *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t ∗ *status*)

This function sets node attribute values in a node attributes object.

A node attributes object is a container of node attributes, optionally passed to mtapi_initialize() to specify non-default node attributes when creating a node.

attributes is a pointer to a node attributes object that was previously initialized with a call to mtapi_nodeattr_init(). Calls to mtapi_nodeattr_set() have no effect on node attributes after the node has been created and initialized with mtapi_initialize(). The node attributes object may safely be deleted by the application after the call to mtapi_initialize().

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set attribute_size to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

MTAPI-defined node attributes:

Attribute num	Description	Data Type	Default
MTAPI_NODES_NUMCORES	(Read-only) number of processor cores of the node.	mtapi_uint_t	(none)

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.

See also

mtapi_nodeattr_init(), mtapi_initialize()

Concurrency

Not thread-safe

Parameters

in,out	*attributes*	Pointer to attributes
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be MTAPI_NULL

6.41.4 Member Data Documentation

6.41.4.1 embb_core_set_t mtapi_node_attributes_struct::core_affinity

stores MTAPI_NODE_CORE_AFFINITY

6.41.4.2 mtapi_uint_t mtapi_node_attributes_struct::num_cores

stores MTAPI_NODE_NUMCORES

6.41.4.3 mtapi_uint_t mtapi_node_attributes_struct::type

stores MTAPI_NODE_TYPE

6.41.4.4 mtapi_uint_t mtapi_node_attributes_struct::max_tasks

stores MTAPI_NODE_MAX_TASKS

6.41.4.5 mtapi_uint_t mtapi_node_attributes_struct::max_actions

stores MTAPI_NODE_MAX_ACTIONS

6.41.4.6 mtapi_uint_t mtapi_node_attributes_struct::max_groups

stores MTAPI_NODE_MAX_GROUPS

6.41.4.7 mtapi_uint_t mtapi_node_attributes_struct::max_queues

stores MTAPI_NODE_MAX_QUEUES

6.41.4.8 mtapi_uint_t mtapi_node_attributes_struct::queue_limit

stores MTAPI_NODE_QUEUE_LIMIT

6.41.4.9 mtapi_uint_t mtapi_node_attributes_struct::max_jobs

stores MTAPI_NODE_MAX_JOBS

6.41.4.10 mtapi_uint_t mtapi_node_attributes_struct::max_actions_per_job

stores MTAPI_NODE_MAX_ACTIONS_PER_JOB

6.41.4.11 mtapi_uint_t mtapi_node_attributes_struct::max_priorities

stores MTAPI_NODE_MAX_PRIORITIES

6.41.4.12 mtapi_boolean_t mtapi_node_attributes_struct::reuse_main_thread

stores MTAPI_NODE_REUSE_MAIN_THREAD

6.41.4.13 mtapi_worker_priority_entry_t∗ mtapi_node_attributes_struct::worker_priorities

stores MTAPI_NODE_WORKER_PRIORITIES

6.42 mtapi_queue_attributes_struct Struct Reference

Queue attributes.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_queue_attributes_struct mtapi_queue_attributes_t

  Queue attributes type.

Public Member Functions

- void mtapi_queueattr_init (mtapi_queue_attributes_t ∗attributes, mtapi_status_t ∗status)

  This function initializes a queue attributes object.
- void mtapi_queueattr_set (mtapi_queue_attributes_t ∗attributes, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

  This function sets queue attribute values in a queue attributes object.

Public Attributes

- mtapi_boolean_t global

  stores MTAPI_QUEUE_GLOBAL
- mtapi_uint_t priority

  stores MTAPI_QUEUE_PRIORITY
- mtapi_uint_t limit

  stores MTAPI_QUEUE_LIMIT
- mtapi_boolean_t ordered

  stores MTAPI_QUEUE_ORDERED
- mtapi_boolean_t retain

  stores MTAPI_QUEUE_RETAIN
- mtapi_boolean_t domain_shared

  stores MTAPI_QUEUE_DOMAIN_SHARED

6.42.1 Detailed Description

Queue attributes.

6.42.2 Member Typedef Documentation

6.42.2.1 typedef struct mtapi_queue_attributes_struct mtapi_queue_attributes_t

Queue attributes type.

6.42.3 Member Function Documentation

6.42.3.1 void mtapi_queueattr_init (mtapi_queue_attributes_t ∗ *attributes,* mtapi_status_t ∗ *status*)

This function initializes a queue attributes object.

A queue attributes object is a container of queue attributes, optionally passed to mtapi_queue_create() to create a queue with non-default attributes.

The application is responsible for allocating the `mtapi_queue_attributes_t` object and initializing it with a call to mtapi_queueattr_init(). The application may then call mtapi_queueattr_set() to specify queue attribute values. Calls to mtapi_queueattr_init() have no effect on queue attributes after the queue has been created. To change an attribute of an existing queue, see mtapi_queue_set_attribute(). The `mtapi_queue_attributes_t` object may safely be deleted by the application after the call to mtapi_queue_create().

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_queue_create(), mtapi_queueattr_set(), mtapi_queue_set_attribute()

Concurrency

Not thread-safe

Parameters

`out`	*attributes*	Pointer to attributes
`out`	*status*	Pointer to error code, may be `MTAPI_NULL`

6.42.3.2 void mtapi_queueattr_set (mtapi_queue_attributes_t ∗ *attributes,* const mtapi_uint_t *attribute_num,* const void ∗ *attribute,* const mtapi_size_t *attribute_size,* mtapi_status_t ∗ *status*)

This function sets queue attribute values in a queue attributes object.

A queue attributes object is a container of queue attributes, optionally passed to mtapi_queue_create() to create a queue with non-default attributes.

attributes must be a pointer to a queue attributes object previously initialized by mtapi_queueattr_init().

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set attribute_size to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

Calls to mtapi_queueattr_set() have no effect on queue attributes once the queue has been created. The mtapi↵_queue_attributes_t object may safely be deleted by the application after the call to mtapi_queue_create().

MTAPI-defined queue attributes:

Attribute num	Description	Data Type	Default
MTAPI_QUEUE_GLO↵BAL	Indicates if this is a globally visible queue. Only global queues are shared with other nodes.	mtapi_boolean↵_t	MTAPI_TRUE
MTAPI_QUEUE_PRI↵ORITY	Priority of the queue.	mtapi_uint_t	0(default priority)
MTAPI_QUEUE_LIMIT	Max. number of elements in the queue; the queue blocks on queuing more items.	mtapi_uint_t	0(0 stands for 'unlimited')
MTAPI_QUEUE_ORD↵ERED	Specify if the queue is order-preserving.	mtapi_boolean↵_t	MTAPI_TRUE
MTAPI_QUEUE_RET↵AIN	Allow enqueuing of jobs when queue is disabled.	mtapi_boolean↵_t	MTAPI_FALSE
MTAPI_DOMAIN_SH↵ARED	Indicates if the queue is shareable across domains.	mtapi_boolean↵_t	MTAPI_TRUE

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_queue_create(), mtapi_queueattr_init()

Concurrency

Not thread-safe

Parameters

in,out	*attributes*	Pointer to attributes
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be MTAPI_NULL

6.42.4 Member Data Documentation

6.42.4.1 mtapi_boolean_t mtapi_queue_attributes_struct::global

stores MTAPI_QUEUE_GLOBAL

6.42.4.2 mtapi_uint_t mtapi_queue_attributes_struct::priority

stores MTAPI_QUEUE_PRIORITY

6.42.4.3 mtapi_uint_t mtapi_queue_attributes_struct::limit

stores MTAPI_QUEUE_LIMIT

6.42.4.4 mtapi_boolean_t mtapi_queue_attributes_struct::ordered

stores MTAPI_QUEUE_ORDERED

6.42.4.5 mtapi_boolean_t mtapi_queue_attributes_struct::retain

stores MTAPI_QUEUE_RETAIN

6.42.4.6 mtapi_boolean_t mtapi_queue_attributes_struct::domain_shared

stores MTAPI_QUEUE_DOMAIN_SHARED

6.43 mtapi_queue_hndl_struct Struct Reference

Queue handle.

```
#include <mtapi.h>
```

Public Types

- typedef struct mtapi_queue_hndl_struct mtapi_queue_hndl_t

    Queue handle type.

Public Attributes

- mtapi_uint_t tag

    version of this handle
- mtapi_queue_id_t id

    pool index of this handle

6.43.1 Detailed Description

Queue handle.

6.43.2 Member Typedef Documentation

6.43.2.1 typedef struct mtapi_queue_hndl_struct mtapi_queue_hndl_t

Queue handle type.

6.43.3 Member Data Documentation

6.43.3.1 mtapi_uint_t mtapi_queue_hndl_struct::tag

version of this handle

6.43.3.2 mtapi_queue_id_t mtapi_queue_hndl_struct::id

pool index of this handle

6.44 mtapi_task_attributes_struct Struct Reference

Task attributes.

`#include <mtapi.h>`

Public Types

- typedef struct mtapi_task_attributes_struct mtapi_task_attributes_t

    Task attributes type.

Public Member Functions

- void mtapi_taskattr_init (mtapi_task_attributes_t ∗attributes, mtapi_status_t ∗status)

    This function initializes a task attributes object.
- void mtapi_taskattr_set (mtapi_task_attributes_t ∗attributes, const mtapi_uint_t attribute_num, const void ∗attribute, const mtapi_size_t attribute_size, mtapi_status_t ∗status)

    This function sets task attribute values in a task attributes object.

Public Attributes

- mtapi_boolean_t is_detached

    stores MTAPI_TASK_DETACHED
- mtapi_uint_t num_instances

    stores MTAPI_TASK_INSTANCES
- mtapi_uint_t priority

    stores MTAPI_TASK_PRIORITY
- mtapi_affinity_t affinity

    stores MTAPI_TASK_AFFINITY
- void * user_data

    stores MTAPI_TASK_USER_DATA
- mtapi_task_complete_function_t complete_func

    stores MTAPI_TASK_COMPLETE_FUNCTION
- mtapi_uint_t problem_size

    stores MTAPI_TASK_PROBLEM_SIZE

6.44.1 Detailed Description

Task attributes.

6.44.2 Member Typedef Documentation

6.44.2.1 typedef struct mtapi_task_attributes_struct mtapi_task_attributes_t

Task attributes type.

6.44.3 Member Function Documentation

6.44.3.1 void mtapi_taskattr_init (mtapi_task_attributes_t * *attributes*, mtapi_status_t * *status*)

This function initializes a task attributes object.

A task attributes object is a container of task attributes. It is an optional argument passed to mtapi_task_start() or mtapi_task_enqueue() to specify non-default task attributes when starting a task.

To set task attributes to non-default values, the application must allocate a task attributes object of type `mtapi_task_attributes_t` and initialize it with a call to mtapi_taskattr_init(). The application may call mtapi_taskattr_set() to specify attribute values. Calls to mtapi_taskattr_init() have no effect on task attributes after the task has started. The `mtapi_task_attributes_t` object may safely be deleted by the application after the task has started.

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

Error code	Description
`MTAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

See also

mtapi_task_start(), mtapi_task_enqueue(), mtapi_taskattr_set()

Concurrency

Not thread-safe

Parameters

out	*attributes*	Pointer to attributes
out	*status*	Pointer to error code, may be `MTAPI_NULL`

6.44.3.2 void mtapi_taskattr_set (mtapi_task_attributes_t * *attributes*, const mtapi_uint_t *attribute_num*, const void * *attribute*, const mtapi_size_t *attribute_size*, mtapi_status_t * *status*)

This function sets task attribute values in a task attributes object.

A task attributes object is a container of task attributes, optionally passed to mtapi_task_start() or mtapi_task_↩ enqueue() to specify non-default task attributes when starting a task.

attributes is a pointer to a task attributes object that was previously initialized with a call to mtapi_taskattr_init(). Calls to mtapi_taskattr_set() have no effect on task attributes after the task has been created. The task attributes object may safely be deleted by the application after the task has started.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set attribute_size to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

MTAPI-defined task attributes:

Attribute num	Description	Data Type	Default
`MTAPI_TASK_DETACH↩ ED`	Indicates if this is a detached task. A detached task is deleted by MTA↩ PI runtime after execution. The task handle of detached tasks must not be used, i.e., it is not possible to wait for completion of dedicated detached tasks. But it is possible to add detached tasks to a group and wait for completion of the group.	`mtapi_boolean_t`	`MTAPI_FALSE`
`MTAPI_TASK_INSTAN↩ CES`	Indicates how many parallel instances of task shall be started by MTAPI. The default case is that each task is executed exactly once. Setting this value to n, the corresponding action code will be executed n times, in parallel, if the underlying hardware allows it. (see chapter 4.1.7 Multi-Instance Tasks, page 107)	`mtapi_uint_t`	1

Attribute num	Description	Data Type	Default
MTAPI_TASK_PRIORI↩ TY	Indicates the prority this task should be run at. Priorities range from zero to one minus the maximum number of priorities specified at the call to mtapi_initialize().	mtapi_uint_t	0 (default priority)
MTAPI_TASK_AFFINI↩ TY	Indicates the affinity of this task. Affinities are manipulated by the matpi_affinity↩ _init() and mtapi_affinity_↩ set() calls.	mtapi_affinity_t	all workers
MTAPI_TASK_USER_D↩ ATA	Provides a pointer to some data required by the user during scheduling (e.g. in a MTAPI plugin).	void*	MTAPI_NULL
MTAPI_TASK_COMPLE↩ TE_FUNCTION	Pointer to a function being called when the task finishes.	mtapi_task_↩ complete_function↩ _t	MTAPI_NULL

On success, *status is set to MTAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

Error code	Description
MTAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MTAPI_ERR_PARAMETER	Invalid attribute parameter.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

See also

mtapi_task_start(), mtapi_task_enqueue(), mtapi_taskattr_init()

Concurrency

Not thread-safe

Parameters

in,out	*attributes*	Pointer to attributes
in	*attribute_num*	Attribute id
in	*attribute*	Pointer to attribute value
in	*attribute_size*	Size of attribute value. may be 0, attribute is interpreted as value in that case
out	*status*	Pointer to error code, may be MTAPI_NULL

6.44.4 Member Data Documentation

6.44.4.1 mtapi_boolean_t mtapi_task_attributes_struct::is_detached

stores MTAPI_TASK_DETACHED

6.44.4.2 mtapi_uint_t mtapi_task_attributes_struct::num_instances

stores MTAPI_TASK_INSTANCES

6.44.4.3 mtapi_uint_t mtapi_task_attributes_struct::priority

stores MTAPI_TASK_PRIORITY

6.44.4.4 mtapi_affinity_t mtapi_task_attributes_struct::affinity

stores MTAPI_TASK_AFFINITY

6.44.4.5 void* mtapi_task_attributes_struct::user_data

stores MTAPI_TASK_USER_DATA

6.44.4.6 mtapi_task_complete_function_t mtapi_task_attributes_struct::complete_func

stores MTAPI_TASK_COMPLETE_FUNCTION

6.44.4.7 mtapi_uint_t mtapi_task_attributes_struct::problem_size

stores MTAPI_TASK_PROBLEM_SIZE

6.45 mtapi_task_hndl_struct Struct Reference

Task handle.

```
#include <mtapi.h>
```

Public Types

- typedef struct mtapi_task_hndl_struct mtapi_task_hndl_t
  Task handle type.

Public Attributes

- mtapi_uint_t tag
  version of this handle
- mtapi_task_id_t id
  pool index of this handle

6.45.1 Detailed Description

Task handle.

6.45.2 Member Typedef Documentation

6.45.2.1 typedef struct mtapi_task_hndl_struct mtapi_task_hndl_t

Task handle type.

6.45.3 Member Data Documentation

6.45.3.1 mtapi_uint_t mtapi_task_hndl_struct::tag

version of this handle

6.45.3.2 mtapi_task_id_t mtapi_task_hndl_struct::id

pool index of this handle

6.46 mtapi_worker_priority_entry_struct Struct Reference

Describes the default priority of all workers or the priority of a specific worker.

```
#include <mtapi.h>
```

Public Attributes

- mtapi_worker_priority_type_t type
  default or specific worker
- embb_thread_priority_t priority
  priority to set

6.46.1 Detailed Description

Describes the default priority of all workers or the priority of a specific worker.

6.46.2 Member Data Documentation

6.46.2.1 mtapi_worker_priority_type_t mtapi_worker_priority_entry_struct::type

default or specific worker

6.46.2.2 embb_thread_priority_t mtapi_worker_priority_entry_struct::priority

priority to set

6.47 embb::base::Mutex Class Reference

Non-recursive, exclusive mutex.

```
#include <mutex.h>
```

Public Member Functions

- Mutex ()

  Creates a mutex which is in unlocked state.
- void Lock ()

  Waits until the mutex can be locked and locks it.
- bool TryLock ()

  Tries to lock the mutex and returns immediately.
- void Unlock ()

  Unlocks the mutex.

6.47.1 Detailed Description

Non-recursive, exclusive mutex.

Mutexes of this type cannot be locked recursively, that is, multiple times by the same thread with unlocking it in between. Moreover, it cannot be copied or assigned.

See also

RecursiveMutex

Implemented concepts:

Mutex Concept

6.47.2 Constructor & Destructor Documentation

6.47.2.1 embb::base::Mutex::Mutex ( )

Creates a mutex which is in unlocked state.

Dynamic memory allocation

Potentially allocates dynamic memory

Concurrency

Not thread-safe

6.47.3 Member Function Documentation

6.47.3.1 void embb::base::Mutex::Lock ( )

Waits until the mutex can be locked and locks it.

Precondition

The mutex is not locked by the current thread.

Postcondition

The mutex is locked

Concurrency

Thread-safe

See also

TryLock(), Unlock()

6.47.3.2 bool embb::base::Mutex::TryLock ( )

Tries to lock the mutex and returns immediately.

Precondition

The mutex is not locked by the current thread.

Postcondition

If successful, the mutex is locked.

Returns

`true` if the mutex could be locked, otherwise `false`.

Concurrency

Thread-safe

See also

Lock(), Unlock()

6.47.3.3 void embb::base::Mutex::Unlock ( )

Unlocks the mutex.

Precondition

The mutex is locked by the current thread

Postcondition

The mutex is unlocked

Concurrency

Thread-safe

See also

Lock(), TryLock()

6.48 embb::dataflow::Network Class Reference

Represents a set of processes that are connected by communication channels.

```
#include <network.h>
```

Classes

- class ConstantSource

    Constant source process template.
- class In

    Input port class.
- struct Inputs

    Provides the input port types for a process.
- class Out

    Output port class.
- struct Outputs

    Provides the output port types for a process.
- class ParallelProcess

    Generic parallel process template.
- class Select

    Select process template.
- class SerialProcess

    Generic serial process template.
- class Sink

    Sink process template.
- class Source

    Source process template.
- class Switch

    Switch process template.

Public Member Functions

- Network ()

    Constructs an empty network.
- Network (int slices)

    Constructs an empty network.
- Network (embb::mtapi::ExecutionPolicy const &policy)

    Constructs an empty network.
- Network (int slices, embb::mtapi::ExecutionPolicy const &policy)

    Constructs an empty network.
- bool IsValid ()

    Checks whether the network is completely connected and free of cycles.
- void operator() ()

    Executes the network until one of the the sources returns `false`.

6.48.1 Detailed Description

Represents a set of processes that are connected by communication channels.

6.48.2 Constructor & Destructor Documentation

6.48.2.1 embb::dataflow::Network::Network ( )

Constructs an empty network.

Note

The number of concurrent tokens will automatically be derived from the structure of the network on the first call to operator(), and the corresponding resources will be allocated then.
When using parallel algorithms inside a dataflow network, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

6.48.2.2 embb::dataflow::Network::Network (int *slices*) [explicit]

Constructs an empty network.

Parameters

slices	Number of concurrent tokens allowed in the network.

Note

The number of slices might be reduced internally if the task limit of the underlying MTAPI node would be exceeded.
When using parallel algorithms inside a dataflow network, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

6.48.2.3 embb::dataflow::Network::Network (**embb::mtapi::ExecutionPolicy** const & *policy*) [explicit]

Constructs an empty network.

Parameters

policy	Default execution policy of the processes in the network.

Note

The number of concurrent tokens will automatically be derived from the structure of the network on the first call to operator(), and the corresponding resources will be allocated then.
When using parallel algorithms inside a dataflow network, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

6.48.2.4 embb::dataflow::Network::Network (int *slices*, **embb::mtapi::ExecutionPolicy** const & *policy*)

Constructs an empty network.

Parameters

slices	Number of concurrent tokens allowed in the network.
policy	Default execution policy of the processes in the network.

Note

The number of slices might be reduced internally if the task limit of the underlying MTAPI node would be exceeded.
When using parallel algorithms inside a dataflow network, the task limit may be exceeded. In that case, increase the task limit of the MTAPI node.

6.48.3 Member Function Documentation

6.48.3.1 bool embb::dataflow::Network::IsValid ( )

Checks whether the network is completely connected and free of cycles.

Returns

`true` if everything is in order, `false` if not.

Note

Executing an invalid network results in an exception. For this reason, it is recommended to first check the network using IsValid().

6.48.3.2 void embb::dataflow::Network::operator() ( )

Executes the network until one of the the sources returns `false`.

Note

If the network was default constructed, the number of concurrent tokens will automatically be derived from the structure of the network on the first call of the operator, and the corresponding resources will be allocated then.

Note

Executing an invalid network results in an exception. For this reason, it is recommended to first check the network using IsValid().

6.49 embb::mtapi::Node Class Reference

A singleton representing the MTAPI runtime.

`#include <node.h>`

Public Types

- typedef embb::base::Function< void, TaskContext & > SMPFunction

  Function type for simple SMP interface.

Public Member Functions

- ∼Node ()

  Destroys the runtime singleton.
- mtapi_uint_t GetCoreCount () const

  Returns the number of available cores.
- mtapi_uint_t GetWorkerThreadCount () const

  Returns the number of worker threads.
- mtapi_uint_t GetQueueCount () const

  Returns the number of available queues.
- mtapi_uint_t GetGroupCount () const

  Returns the number of available groups.
- mtapi_uint_t GetTaskLimit () const

  Returns the number of available tasks.
- Task Start (SMPFunction const &func)

  Starts a new Task.
- Task Start (SMPFunction const &func, ExecutionPolicy const &policy)

  Starts a new Task with a given affinity and priority.
- template<typename ARGS , typename RES >
  Task Start (mtapi_task_id_t task_id, Job const &job, const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes)

  Starts a new Task.
- template<typename ARGS , typename RES >
  Task Start (mtapi_task_id_t task_id, Job const &job, const ARGS ∗arguments, RES ∗results)

Starts a new Task.

- template<typename ARGS , typename RES >
  Task Start (Job const &job, const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes)

  Starts a new Task.

- template<typename ARGS , typename RES >
  Task Start (Job const &job, const ARGS ∗arguments, RES ∗results)

  Starts a new Task.

- Job GetJob (mtapi_job_id_t job_id)

  Retrieves a handle to the Job identified by `job_id` within the domain of the local Node.

- Job GetJob (mtapi_job_id_t job_id, mtapi_domain_t domain_id)

  Retrieves a handle to the Job identified by `job_id` and `domain_id`.

- Action CreateAction (mtapi_job_id_t job_id, mtapi_action_function_t func, const void ∗node_local_data, mtapi_size_t node_local_data_size, ActionAttributes const &attributes)

  Constructs an Action.

- Action CreateAction (mtapi_job_id_t job_id, mtapi_action_function_t func, const void ∗node_local_data, mtapi_size_t node_local_data_size)

  Constructs an Action.

- Action CreateAction (mtapi_job_id_t job_id, mtapi_action_function_t func, ActionAttributes const &attributes)

  Constructs an Action.

- Action CreateAction (mtapi_job_id_t job_id, mtapi_action_function_t func)

  Constructs an Action.

- Group CreateGroup ()

  Constructs a Group object with default attributes.

- Group CreateGroup (mtapi_group_id_t id)

  Constructs a Group object with default attributes and the given ID.

- Group CreateGroup (GroupAttributes const &group_attr)

  Constructs a Group object using the given Attributes.

- Group CreateGroup (mtapi_group_id_t id, GroupAttributes const &group_attr)

  Constructs a Group object with given attributes and ID.

- Queue CreateQueue (Job &job)

  Constructs a Queue with the given Job and default attributes.

- Queue CreateQueue (Job const &job, QueueAttributes const &attr)

  Constructs a Queue with the given Job and QueueAttributes.

- Task Start (mtapi_task_id_t task_id, mtapi_job_hndl_t job, const void ∗arguments, mtapi_size_t arguments←_size, void ∗results, mtapi_size_t results_size, mtapi_task_attributes_t const ∗attributes)

  Starts a new Task.

- void YieldToScheduler ()

  This function yields execution to the MTAPI scheduler for at most one task.

Static Public Member Functions

- static void Initialize (mtapi_domain_t domain_id, mtapi_node_t node_id)

  Initializes the runtime singleton using default values:

- static void Initialize (mtapi_domain_t domain_id, mtapi_node_t node_id, NodeAttributes const &attributes)

  Initializes the runtime singleton.

- static bool IsInitialized ()

  Checks if runtime is initialized.

- static Node & GetInstance ()

  Gets the instance of the runtime system.

- static void Finalize ()

  Shuts the runtime system down.

6.49.1 Detailed Description

A singleton representing the MTAPI runtime.

6.49.2 Member Typedef Documentation

6.49.2.1 typedef embb::base::Function<void, TaskContext &> embb::mtapi::Node::SMPFunction

Function type for simple SMP interface.

6.49.3 Constructor & Destructor Documentation

6.49.3.1 embb::mtapi::Node::∼Node ( )

Destroys the runtime singleton.

Concurrency

Not thread-safe

6.49.4 Member Function Documentation

6.49.4.1 static void embb::mtapi::Node::Initialize (mtapi_domain_t *domain_id*, mtapi_node_t *node_id*) [static]

Initializes the runtime singleton using default values:

- all available cores will be used
- maximum number of tasks is 1024
- maximum number of groups is 128
- maximum number of queues is 16
- maximum queue capacity is 1024
- maximum number of priorities is 4.

Concurrency

Not thread-safe

Exceptions

ErrorException	if the singleton was already initialized or the Node could not be initialized.

Dynamic memory allocation

Allocates about 200kb of memory.

Parameters

in	*domain_id*	The domain id to use
in	*node_id*	The node id to use

6.49.4.2 static void embb::mtapi::Node::Initialize (mtapi_domain_t *domain_id,* mtapi_node_t *node_id,* NodeAttributes const & *attributes*) [static]

Initializes the runtime singleton.

Concurrency

Not thread-safe

Exceptions

ErrorException	if the singleton was already initialized or the Node could not be initialized.

Dynamic memory allocation

Allocates some memory depending on the values given.

Parameters

in	*domain_id*	The domain id to use
in	*node_id*	The node id to use
in	*attributes*	Attributes to use

6.49.4.3 static bool embb::mtapi::Node::IsInitialized ( ) [static]

Checks if runtime is initialized.

Returns

`true` if the Node singleton is already initialized, false otherwise

Concurrency

Thread-safe and wait-free

6.49.4.4 static Node& embb::mtapi::Node::GetInstance ( ) [static]

Gets the instance of the runtime system.

Returns

Reference to the Node singleton

Concurrency

Thread-safe

6.49.4.5 static void embb::mtapi::Node::Finalize ( ) [static]

Shuts the runtime system down.

Exceptions

ErrorException	if the singleton is not initialized.

Concurrency

Not thread-safe

6.49.4.6 mtapi_uint_t embb::mtapi::Node::GetCoreCount ( ) const

Returns the number of available cores.

Returns

The number of available cores

Concurrency

Thread-safe and wait-free

6.49.4.7 mtapi_uint_t embb::mtapi::Node::GetWorkerThreadCount ( ) const

Returns the number of worker threads.

Returns

The number of worker threads.

Concurrency

Thread-safe and wait-free

6.49.4.8 mtapi_uint_t embb::mtapi::Node::GetQueueCount ( ) const

Returns the number of available queues.

Returns

The number of available queues

Concurrency

Thread-safe and wait-free

6.49.4.9 mtapi_uint_t embb::mtapi::Node::GetGroupCount ( ) const

Returns the number of available groups.

Returns

The number of available groups

Concurrency

Thread-safe and wait-free

6.49.4.10 mtapi_uint_t embb::mtapi::Node::GetTaskLimit ( ) const

Returns the number of available tasks.

Returns

The number of available tasks

Concurrency

Thread-safe and wait-free

6.49.4.11 Task embb::mtapi::Node::Start (SMPFunction const & *func*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

func	Function to use for the task.

6.49.4.12 Task embb::mtapi::Node::Start (SMPFunction const & *func*, ExecutionPolicy const & *policy*)

Starts a new Task with a given affinity and priority.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

func	Function to use for the task.
policy	Affinity and priority of the task.

6.49.4.13 template<typename ARGS , typename RES > Task embb::mtapi::Node::Start (mtapi_task_id_t *task_id*, Job const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*, TaskAttributes const & *attributes*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.49.4.14 template<typename ARGS , typename RES > Task embb::mtapi::Node::Start (mtapi_task_id_t *task_id*, Job const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.

6.49.4.15 template<typename ARGS , typename RES > Task embb::mtapi::Node::Start (Job const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*, TaskAttributes const & *attributes*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.49.4.16 template<typename ARGS , typename RES > Task embb::mtapi::Node::Start (Job const & *job*, const ARGS ∗ *arguments*, RES ∗ *results*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

job	The Job to execute.
arguments	Pointer to the arguments.
results	Pointer to the results.

6.49.4.17 Job embb::mtapi::Node::GetJob (mtapi_job_id_t *job_id*)

Retrieves a handle to the Job identified by `job_id` within the domain of the local Node.

Returns

The handle to the requested Job.

Concurrency

Thread-safe and wait-free

Parameters

in	*job_id*	The id of the job

6.49.4.18 Job embb::mtapi::Node::GetJob (mtapi_job_id_t *job_id*, mtapi_domain_t *domain_id*)

Retrieves a handle to the Job identified by `job_id` and `domain_id`.

Returns

The handle to the requested Job.

Concurrency

Thread-safe and wait-free

Parameters

in	*job_id*	The id of the job
in	*domain_id*	The domain id to use

6.49.4.19 Action embb::mtapi::Node::CreateAction (mtapi_job_id_t *job_id*, mtapi_action_function_t *func*, const void * *node_local_data*, mtapi_size_t *node_local_data_size*, ActionAttributes const & *attributes*)

Constructs an Action.

Returns

The handle to the new Action.

Concurrency

Thread-safe and lock-free

Parameters

job_id	Job ID the Action belongs to
func	The action function
node_local_data	Node local data available to all Tasks using this Action
node_local_data_size	Size of node local data
attributes	Attributes of the Action

6.49.4.20 Action embb::mtapi::Node::CreateAction (mtapi_job_id_t *job_id*, mtapi_action_function_t *func*, const void ∗ *node_local_data*, mtapi_size_t *node_local_data_size*)

Constructs an Action.

Returns

The handle to the new Action.

Concurrency

Thread-safe and lock-free

Parameters

job_id	Job ID the Action belongs to
func	The action function
node_local_data	Node local data available to all Tasks using this Action
node_local_data_size	Size of node local data

6.49.4.21 Action embb::mtapi::Node::CreateAction (mtapi_job_id_t *job_id*, mtapi_action_function_t *func*, ActionAttributes const & *attributes*)

Constructs an Action.

Returns

The handle to the new Action.

Concurrency

Thread-safe and lock-free

Parameters

job_id	Job ID the Action belongs to
func	The action function
attributes	Attributes of the Action

6.49.4.22 Action embb::mtapi::Node::CreateAction (mtapi_job_id_t *job_id*, mtapi_action_function_t *func*)

Constructs an Action.

Returns

The handle to the new Action.

Concurrency

Thread-safe and lock-free

Parameters

job←_id	Job ID the Action belongs to
func	The action function

6.49.4.23 Group embb::mtapi::Node::CreateGroup ( )

Constructs a Group object with default attributes.

Returns

The handle to the new Group.

Concurrency

Thread-safe and lock-free

6.49.4.24 Group embb::mtapi::Node::CreateGroup (mtapi_group_id_t *id*)

Constructs a Group object with default attributes and the given ID.

Returns

The handle to the new Group.

Concurrency

Thread-safe and lock-free

Parameters

id	A user defined ID of the Group.

6.49.4.25 Group embb::mtapi::Node::CreateGroup (GroupAttributes const & *group_attr*)

Constructs a Group object using the given Attributes.

Returns

The handle to the new Group.

Concurrency

Thread-safe and lock-free

Parameters

group_attr	The GroupAttributes to use.

6.49.4.26 Group embb::mtapi::Node::CreateGroup (mtapi_group_id_t *id*, GroupAttributes const & *group_attr*)

Constructs a Group object with given attributes and ID.

Returns

The handle to the new Group.

Concurrency

Thread-safe and lock-free

Parameters

id	A user defined ID of the Group.
group_attr	The GroupAttributes to use.

6.49.4.27 Queue embb::mtapi::Node::CreateQueue (Job & *job*)

Constructs a Queue with the given Job and default attributes.

Returns

The handle to the new Queue.

Concurrency

Thread-safe and lock-free

Parameters

job	The Job to use for the Queue.

6.49.4.28 Queue embb::mtapi::Node::CreateQueue (Job const & *job*, QueueAttributes const & *attr*)

Constructs a Queue with the given Job and QueueAttributes.

Returns

The handle to the new Queue.

Concurrency

Thread-safe and lock-free

Parameters

job	The Job to use for the Queue.
attr	The attributes to use.

6.49.4.29 Task embb::mtapi::Node::Start (mtapi_task_id_t *task_id*, mtapi_job_hndl_t *job*, const void ∗ *arguments*, mtapi_size_t *arguments_size*, void ∗ *results*, mtapi_size_t *results_size*, mtapi_task_attributes_t const ∗ *attributes*)

Starts a new Task.

Returns

The handle to the started Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
job	The Job to execute.
arguments	Pointer to the arguments buffer
arguments_size	Size of the arguments buffer
results	Pointer to the result buffer
results_size	Size of the result buffer
attributes	Attributes to use for the task

6.49.4.30 void embb::mtapi::Node::YieldToScheduler ( )

This function yields execution to the MTAPI scheduler for at most one task.

Concurrency

Thread-safe

6.50 embb::mtapi::NodeAttributes Class Reference

Contains attributes of a Node.

```
#include <node_attributes.h>
```

Public Member Functions

- NodeAttributes ()

  Constructs a NodeAttributes object.
- NodeAttributes (NodeAttributes const &other)

  Copies a NodeAttributes object.
- void operator= (NodeAttributes const &other)

  Copies a NodeAttributes object.
- NodeAttributes & SetCoreAffinity (embb::base::CoreSet const &cores)

  Sets the core affinity of the Node.
- NodeAttributes & SetWorkerPriority (mtapi_worker_priority_entry_t ∗worker_priorities)

  Sets the priority of the specified worker threads.
- NodeAttributes & SetMaxTasks (mtapi_uint_t value)

  Sets the maximum number of concurrently active tasks.
- NodeAttributes & SetMaxActions (mtapi_uint_t value)

  Sets the maximum number of actions.
- NodeAttributes & SetMaxGroups (mtapi_uint_t value)

  Sets the maximum number of groups.
- NodeAttributes & SetMaxQueues (mtapi_uint_t value)

  Sets the maximum number of queues.
- NodeAttributes & SetQueueLimit (mtapi_uint_t value)

  Sets the default limit (capacity) of all queues.
- NodeAttributes & SetMaxJobs (mtapi_uint_t value)

  Sets the maximum number of available jobs.
- NodeAttributes & SetMaxActionsPerJob (mtapi_uint_t value)

  Sets the maximum number of actions per job.
- NodeAttributes & SetMaxPriorities (mtapi_uint_t value)

  Sets the maximum number of available priorities.
- NodeAttributes & SetReuseMainThread (mtapi_boolean_t reuse)

  Enables or disables the reuse of the main thread as a worker.
- mtapi_node_attributes_t const & GetInternal () const

  Returns the internal representation of this object.

6.50.1 Detailed Description

Contains attributes of a Node.

6.50.2 Constructor & Destructor Documentation

6.50.2.1 embb::mtapi::NodeAttributes::NodeAttributes ( )

Constructs a NodeAttributes object.

Concurrency

Thread-safe and wait-free

6.50.2.2 embb::mtapi::NodeAttributes::NodeAttributes (NodeAttributes const & *other*)

Copies a NodeAttributes object.

Concurrency

Thread-safe and wait-free

Parameters

other	The NodeAttributes to copy.

6.50.3 Member Function Documentation

6.50.3.1 void embb::mtapi::NodeAttributes::operator= (NodeAttributes const & *other*)

Copies a NodeAttributes object.

Concurrency

Thread-safe and wait-free

Parameters

other	The NodeAttributes to copy.

6.50.3.2 NodeAttributes& embb::mtapi::NodeAttributes::SetCoreAffinity (embb::base::CoreSet const & *cores*)

Sets the core affinity of the Node.

This also determines the number of worker threads.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

cores	The cores to use.

6.50.3.3 NodeAttributes& embb::mtapi::NodeAttributes::SetWorkerPriority (mtapi_worker_priority_entry_t * *worker_priorities*)

Sets the priority of the specified worker threads.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

worker_priorities	Array of priorities

6.50.3.4 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxTasks (mtapi_uint_t *value*)

Sets the maximum number of concurrently active tasks.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.5 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxActions (mtapi_uint_t *value*)

Sets the maximum number of actions.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.6 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxGroups (mtapi_uint_t *value*)

Sets the maximum number of groups.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.7 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxQueues (mtapi_uint_t *value*)

Sets the maximum number of queues.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.8 NodeAttributes& embb::mtapi::NodeAttributes::SetQueueLimit (mtapi_uint_t *value*)

Sets the default limit (capacity) of all queues.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.9 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxJobs (mtapi_uint_t *value*)

Sets the maximum number of available jobs.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.10 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxActionsPerJob (mtapi_uint_t *value*)

Sets the maximum number of actions per job.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.11 NodeAttributes& embb::mtapi::NodeAttributes::SetMaxPriorities (mtapi_uint_t *value*)

Sets the maximum number of available priorities.

The priority values will range from 0 to `value` - 1 with 0 being the highest priority.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

value	The value to set.

6.50.3.12 NodeAttributes& embb::mtapi::NodeAttributes::SetReuseMainThread (mtapi_boolean_t *reuse*)

Enables or disables the reuse of the main thread as a worker.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

reuse	The state to set.

6.50.3.13 mtapi_node_attributes_t const& embb::mtapi::NodeAttributes::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A reference to the internal mtapi_node_attributes_t structure.

Concurrency

Thread-safe and wait-free

6.51 embb::base::NoMemoryException Class Reference

Indicates lack of memory necessary to allocate a resource.

```
#include <exceptions.h>
```

Public Member Functions

- NoMemoryException (const char *message)

  Constructs an exception with the specified message.
- virtual int Code () const

  Returns an integer code representing the exception.
- virtual const char * What () const throw ()

  Returns the error message.

6.51.1 Detailed Description

Indicates lack of memory necessary to allocate a resource.

6.51.2 Constructor & Destructor Documentation

6.51.2.1 embb::base::NoMemoryException::NoMemoryException (const char * *message*) [explicit]

Constructs an exception with the specified message.

Parameters

in	*message*	Error message

6.51.3 Member Function Documentation

6.51.3.1 virtual int embb::base::NoMemoryException::Code ( ) const [virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implements embb::base::Exception.

6.51.3.2 virtual const char* embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.52 embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator > Class Template Reference

Pool for thread-safe management of arbitrary objects.

```
#include <object_pool.h>
```

Public Member Functions

- ObjectPool (size_t capacity)

  Constructs an object pool with capacity `capacity`*.*
- ∼ObjectPool ()

  Destructs the pool.
- size_t GetCapacity ()

  Returns the capacity of the pool.
- void Free (Type ∗obj)

  Returns an element to the pool.
- Type ∗ Allocate (...)

  Allocates an element from the pool.

6.52.1 Detailed Description

template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>>
class embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >

Pool for thread-safe management of arbitrary objects.

Template Parameters

Type	Element type
ValuePool	Type of the underlying value pool, determines whether the object pool is wait-free or lock-free
ObjectAllocator	Type of allocator used to allocate objects

6.52.2 Constructor & Destructor Documentation

6.52.2.1 template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>> embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >::ObjectPool (size_t *capacity*)

Constructs an object pool with capacity `capacity`.

Dynamic memory allocation

Allocates `capacity` elements of type `Type`.

Concurrency

Not thread-safe

Parameters

in	*capacity*	Number of elements the pool can hold

6.52.2.2 template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>> embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >::∼ObjectPool ( )

Destructs the pool.

Concurrency

Not thread-safe

6.52.3 Member Function Documentation

6.52.3.1 template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>> size_t embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >::GetCapacity ( )

Returns the capacity of the pool.

Returns

Number of elements the pool can hold.

Concurrency

Thread-safe and wait-free

6.52.3.2 template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>> void embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >::Free (Type ∗ *obj*)

Returns an element to the pool.

If the underlying value pool is wait-free/lock-free, this operation is also wait-free/lock-free, respectively.

Note

The element must have been allocated with Allocate().

Parameters

in	*obj*	Pointer to the object to be freed

6.52.3.3 template<class Type, typename ValuePool = embb::containers::WaitFreeArrayValuePool< bool, false >, class ObjectAllocator = embb::base::Allocator<Type>> Type* embb::containers::ObjectPool< Type, ValuePool, ObjectAllocator >::Allocate (...)

Allocates an element from the pool.

If the underlying value pool is wait-free/lock-free, this operation is also wait-free/lock-free, respectively.

Returns

Pointer to the allocated object if successful, otherwise `NULL`.

Parameters

...	Arguments of arbitrary type, passed to the object's constructor

6.53 embb::dataflow::Network::Out< Type > Class Template Reference

Output port class.

`#include <network.h>`

Public Types

- typedef In< Type > InType

  Input port class that can be connected to this output port.

Public Member Functions

- void Connect (InType &input)

  Connects this output port to the input port `input`*.*
- void operator>> (InType &input)

  Connects this output port to the input port `input`*.*

6.53.1 Detailed Description

template<typename Type>
class embb::dataflow::Network::Out< Type >

Output port class.

6.53.2 Member Typedef Documentation

6.53.2.1 template<typename Type > typedef In<Type> embb::dataflow::Network::Out< Type >::InType

Input port class that can be connected to this output port.

6.53.3 Member Function Documentation

6.53.3.1 template<typename Type > void embb::dataflow::Network::Out< Type >::Connect (InType & *input*)

Connects this output port to the input port `input`.

If the input port already was connected to a different output an ErrorException is thrown.

Parameters

input	The input port to connect to.

6.53.3.2 template<typename Type > void embb::dataflow::Network::Out< Type >::operator>> (InType & *input*)

Connects this output port to the input port `input`.

If the input port already was connected to a different output an ErrorException is thrown.

Parameters

input	The input port to connect to.

6.54 embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 > Struct Template Reference

Provides the output port types for a process.

`#include <network.h>`

Classes

- struct Types

  Type list used to derive output port types from Index.

Public Member Functions

- template<int Index>
  Types< Index >::Result & Get ()

6.54.1 Detailed Description

template<typename T1, typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb↩::base::internal::Nil, typename T5 = embb::base::internal::Nil>
struct embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 >

Provides the output port types for a process.

Template Parameters

T1	Type of first port.
T2	Optional type of second port.
T3	Optional type of third port.
T4	Optional type of fourth port.
T5	Optional type of fifth port.

6.54.2 Member Function Documentation

6.54.2.1 template<typename T1 , typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil> template<int Index> Types<Index>::Result& embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 >::Get ( )

Returns

Reference to output port at Index.

6.55 embb::base::OverflowException Class Reference

Indicates a numeric overflow.

```
#include <exceptions.h>
```

Public Member Functions

- OverflowException (const char ∗message)

  Constructs an exception with the specified message.
- virtual int Code () const

  Returns an integer code representing the exception.
- virtual const char ∗ What () const throw ()

  Returns the error message.

6.55.1 Detailed Description

Indicates a numeric overflow.

6.55.2 Constructor & Destructor Documentation

6.55.2.1 embb::base::OverflowException::OverflowException (const char ∗ *message*) [explicit]

Constructs an exception with the specified message.

Parameters

in	*message*	Error message

6.55.3 Member Function Documentation

6.55.3.1 virtual int embb::base::OverflowException::Code ( ) const [virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implements embb::base::Exception.

6.55.3.2 virtual const char∗ embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.56 embb::dataflow::Network::ParallelProcess< Inputs, Outputs > Class Template Reference

Generic parallel process template.

`#include <network.h>`

Public Types

- typedef embb::base::Function< void, INPUT_TYPE_LIST, OUTPUT_TYPE_LIST > FunctionType

    Function type to use when processing tokens.
- typedef Inputs< INPUT_TYPE_LIST > InputsType

    Input port type list.
- typedef Outputs< OUTPUT_TYPE_LIST > OutputsType

    Output port type list.

Public Member Functions

- ParallelProcess (Network &network, FunctionType function)

  Constructs a ParallelProcess with a user specified processing function.
- ParallelProcess (Network &network, embb::mtapi::Job job)

  Constructs a ParallelProcess with a user specified embb::mtapi::Job.
- ParallelProcess (Network &network, FunctionType function, embb::mtapi::ExecutionPolicy const &policy)

  Constructs a ParallelProcess with a user specified processing function.
- ParallelProcess (Network &network, embb::mtapi::Job job, embb::mtapi::ExecutionPolicy const &policy)

  Constructs a ParallelProcess with a user specified embb::mtapi::Job.
- virtual bool HasInputs () const
- InputsType & GetInputs ()
- template<int Index>
  InputsType::Types< Index >::Result & GetInput ()
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)

  Connects output port 0 to input port 0 of `target`.

6.56.1 Detailed Description

template<class Inputs, class Outputs>
class embb::dataflow::Network::ParallelProcess< Inputs, Outputs >

Generic parallel process template.

Implements a generic parallel process in the network that may have one to four input ports and one to four output ports but no more that five total ports. Tokens are processed as soon as all inputs for that token are complete.

See also

Source, SerialProcess, Sink, Switch, Select

Template Parameters

Inputs	Inputs of the process.
Outputs	Outputs of the process.

6.56.2 Member Typedef Documentation

6.56.2.1 template<class Inputs , class Outputs > typedef embb::base::Function<void, INPUT_TYPE_LIST, OUTPUT_TYPE_LIST> embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::FunctionType

Function type to use when processing tokens.

6.56.2.2 template< class Inputs , class Outputs > typedef Inputs< INPUT_TYPE_LIST> embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::InputsType

Input port type list.

6.56.2.3 template< class Inputs , class Outputs > typedef Outputs< OUTPUT_TYPE_LIST> embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::OutputsType

Output port type list.

6.56.3 Constructor & Destructor Documentation

6.56.3.1 template< class Inputs , class Outputs > embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::ParallelProcess (Network & *network*, FunctionType *function*)

Constructs a ParallelProcess with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.

6.56.3.2 template< class Inputs , class Outputs > embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::ParallelProcess (Network & *network*, embb::mtapi::Job *job*)

Constructs a ParallelProcess with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.

6.56.3.3 template< class Inputs , class Outputs > embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::ParallelProcess (Network & *network*, FunctionType *function*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a ParallelProcess with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.
policy	The execution policy of the process.

6.56.3.4 template<class Inputs , class Outputs > embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::ParallelProcess (Network & *network*, embb::mtapi::Job *job*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a ParallelProcess with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.
policy	The execution policy of the process.

6.56.4 Member Function Documentation

6.56.4.1 template<class Inputs , class Outputs > virtual bool embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::HasInputs ( ) const [virtual]

Returns

`true` if the ParallelProcess has any inputs, `false` otherwise.

6.56.4.2 template<class Inputs , class Outputs > InputsType& embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::GetInputs ( )

Returns

Reference to a list of all input ports.

6.56.4.3 template<class Inputs , class Outputs > template<int Index> InputsType::Types<Index>::Result& embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::GetInput ( )

Returns

Input port at Index.

6.56.4.4 template<class Inputs , class Outputs > virtual bool embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::HasOutputs ( ) const [virtual]

Returns

`true` if the ParallelProcess has any outputs, `false` otherwise.

6.56.4.5 template< class Inputs , class Outputs > OutputsType& embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.56.4.6 template< class Inputs , class Outputs > template< int Index> OutputsType::Types<Index>::Result& embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::GetOutput ( )

Returns

Output port at Index.

6.56.4.7 template< class Inputs , class Outputs > template< typename T > void embb::dataflow::Network::ParallelProcess< Inputs, Outputs >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.57 embb::base::Placeholder Class Reference

Provides placeholders for Function arguments used in Bind()

`#include <function.h>`

Static Public Attributes

- static Arg_1 _1

  Placeholder variable to be used in Bind() for keeping one argument unbound.
- static Arg_2 _2

  Placeholder variable to be used in Bind() for keeping one argument unbound.
- static Arg_3 _3

  Placeholder variable to be used in Bind() for keeping one argument unbound.
- static Arg_4 _4

  Placeholder variable to be used in Bind() for keeping one argument unbound.
- static Arg_5 _5

  Placeholder variable to be used in Bind() for keeping one argument unbound.

6.57.1 Detailed Description

Provides placeholders for Function arguments used in Bind()

6.57.2 Member Data Documentation

6.57.2.1 Arg_1 embb::base::Placeholder::_1 [static]

Placeholder variable to be used in Bind() for keeping one argument unbound.

6.57.2.2 Arg_2 embb::base::Placeholder::_2 [static]

Placeholder variable to be used in Bind() for keeping one argument unbound.

6.57.2.3 Arg_3 embb::base::Placeholder::_3 [static]

Placeholder variable to be used in Bind() for keeping one argument unbound.

6.57.2.4 Arg_4 embb::base::Placeholder::_4 [static]

Placeholder variable to be used in Bind() for keeping one argument unbound.

6.57.2.5 Arg_5 embb::base::Placeholder::_5 [static]

Placeholder variable to be used in Bind() for keeping one argument unbound.

6.58 embb::mtapi::Queue Class Reference

Allows for stream processing, either ordered or unordered.

```
#include <queue.h>
```

Public Member Functions

- Queue ()

  Constructs an invalid Queue.
- Queue (Queue const &other)

  Copies a Queue.
- Queue & operator= (Queue const &other)

  Copies a Queue.
- void Delete ()

  Deletes a Queue object.
- void Enable ()

  Enables the Queue.
- void Disable (mtapi_timeout_t timeout)

  Disables the Queue.
- void Disable ()

  Disables the Queue.
- template<typename ARGS , typename RES >
  Task Enqueue (mtapi_task_id_t task_id, const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes, Group const &group)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (mtapi_task_id_t task_id, const ARGS ∗arguments, RES ∗results, Group const &group)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (mtapi_task_id_t task_id, const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (mtapi_task_id_t task_id, const ARGS ∗arguments, RES ∗results)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes, Group const &group)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (const ARGS ∗arguments, RES ∗results, Group const &group)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (const ARGS ∗arguments, RES ∗results, TaskAttributes const &attributes)

  Enqueues a new Task.
- template<typename ARGS , typename RES >
  Task Enqueue (const ARGS ∗arguments, RES ∗results)

  Enqueues a new Task.
- mtapi_queue_hndl_t GetInternal () const

  Returns the internal representation of this object.

6.58.1 Detailed Description

Allows for stream processing, either ordered or unordered.

6.58.2 Constructor & Destructor Documentation

6.58.2.1 embb::mtapi::Queue::Queue ( )

Constructs an invalid Queue.

Concurrency

Thread-safe and wait-free

6.58.2.2 embb::mtapi::Queue::Queue (Queue const & *other*)

Copies a Queue.

Concurrency

Thread-safe and wait-free

Parameters

other	The Queue to copy

6.58.3 Member Function Documentation

6.58.3.1 Queue& embb::mtapi::Queue::operator= (Queue const & *other*)

Copies a Queue.

Returns

Reference to this object.

Concurrency

Thread-safe and wait-free

Parameters

other	The Queue to copy

6.58.3.2 void embb::mtapi::Queue::Delete ( )

Deletes a Queue object.

Concurrency

Thread-safe

6.58.3.3 void embb::mtapi::Queue::Enable ( )

Enables the Queue.

Tasks enqueued while the Queue was disabled are executed.

Concurrency

Thread-safe and wait-free

6.58.3.4 void embb::mtapi::Queue::Disable (mtapi_timeout_t *timeout*)

Disables the Queue.

Running Tasks are canceled. The Queue waits for the Tasks to finish for `timout` milliseconds.

Concurrency

Thread-safe and wait-free

Parameters

timeout	The timeout in milliseconds.

6.58.3.5 void embb::mtapi::Queue::Disable ( )

Disables the Queue.

Running Tasks are canceled. The Queue waits for the Tasks to finish.

Concurrency

Thread-safe and wait-free

6.58.3.6 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (mtapi_task_id_t *task_id,* const ARGS ∗ *arguments,* RES ∗ *results,* TaskAttributes const & *attributes,* Group const & *group*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task
group	The Group to start the Task in

6.58.3.7 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (mtapi_task_id_t *task_id,* const ARGS ∗ *arguments,* RES ∗ *results,* Group const & *group*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
arguments	Pointer to the arguments.
results	Pointer to the results.
group	The Group to start the Task in

6.58.3.8 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (mtapi_task_id_t *task_id,* const ARGS ∗ *arguments,* RES ∗ *results,* TaskAttributes const & *attributes*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.58.3.9 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (mtapi_task_id_t *task_id*, const ARGS * *arguments*, RES * *results*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

task_id	A user defined ID of the Task.
arguments	Pointer to the arguments.
results	Pointer to the results.

6.58.3.10 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (const ARGS * *arguments*, RES * *results*, TaskAttributes const & *attributes*, Group const & *group*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task
group	The Group to start the Task in

6.58.3.11 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (const ARGS * *arguments*, RES * *results*, Group const & *group*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

arguments	Pointer to the arguments.
results	Pointer to the results.
group	The Group to start the Task in

6.58.3.12 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (const ARGS ∗ *arguments*, RES ∗ *results*, TaskAttributes const & *attributes*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

arguments	Pointer to the arguments.
results	Pointer to the results.
attributes	Attributes of the Task

6.58.3.13 template<typename ARGS , typename RES > Task embb::mtapi::Queue::Enqueue (const ARGS ∗ *arguments*, RES ∗ *results*)

Enqueues a new Task.

Returns

The handle to the enqueued Task.

Concurrency

Thread-safe

Parameters

arguments	Pointer to the arguments.
results	Pointer to the results.

6.58.3.14 mtapi_queue_hndl_t embb::mtapi::Queue::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_queue_hndl_t.

Concurrency

Thread-safe and wait-free

6.59 embb::mtapi::QueueAttributes Class Reference

Contains attributes of a Queue.

```
#include <queue_attributes.h>
```

Public Member Functions

- QueueAttributes ()

  Constructs a QueueAttributes object.
- QueueAttributes & SetGlobal (bool state)

  Sets the global property of a Queue.
- QueueAttributes & SetOrdered (bool state)

  Sets the ordered property of a Queue.
- QueueAttributes & SetRetain (bool state)

  Sets the retain property of a Queue.
- QueueAttributes & SetDomainShared (bool state)

  Sets the domain shared property of a Queue.
- QueueAttributes & SetPriority (mtapi_uint_t priority)

  Sets the priority of a Queue.
- QueueAttributes & SetLimit (mtapi_uint_t limit)

  Sets the limit (capacity) of a Queue.
- mtapi_queue_attributes_t const & GetInternal () const

  Returns the internal representation of this object.

6.59.1 Detailed Description

Contains attributes of a Queue.

6.59.2 Constructor & Destructor Documentation

6.59.2.1 embb::mtapi::QueueAttributes::QueueAttributes ( )

Constructs a QueueAttributes object.

Concurrency

Not thread-safe

6.59.3 Member Function Documentation

6.59.3.1 QueueAttributes& embb::mtapi::QueueAttributes::SetGlobal (bool *state*)

Sets the global property of a Queue.

This determines whether the object will be visible across nodes.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

state	The state to set.

6.59.3.2 QueueAttributes& embb::mtapi::QueueAttributes::SetOrdered (bool *state*)

Sets the ordered property of a Queue.

If set to `true`, tasks enqueued will be executed in order.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

state	The state to set.

6.59.3.3 QueueAttributes& embb::mtapi::QueueAttributes::SetRetain (bool *state*)

Sets the retain property of a Queue.

If set to `true`, tasks will be retained while a queue is disabled. Otherwise the will be canceled.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

state	The state to set.

6.59.3.4 QueueAttributes& embb::mtapi::QueueAttributes::SetDomainShared (bool *state*)

Sets the domain shared property of a Queue.

This determines whether the object will be visible across domains.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

state	The state to set.

6.59.3.5 QueueAttributes& embb::mtapi::QueueAttributes::SetPriority (mtapi_uint_t *priority*)

Sets the priority of a Queue.

The priority influences the order in which tasks are chosen for execution.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

priority	The priority to set.

6.59.3.6 QueueAttributes& embb::mtapi::QueueAttributes::SetLimit (mtapi_uint_t *limit*)

Sets the limit (capacity) of a Queue.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

limit	The limit to set.

6.59.3.7 mtapi_queue_attributes_t const& embb::mtapi::QueueAttributes::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A reference to the internal mtapi_queue_attributes_t structure.

Concurrency

Thread-safe and wait-free

6.60 embb::base::Allocator< Type >::rebind< OtherType > Struct Template Reference

Rebind allocator to type OtherType.

`#include <memory_allocation.h>`

Public Types

- typedef Allocator< OtherType > other

  Type to rebind to.

6.60.1 Detailed Description

template<typename Type>
template<typename OtherType>
struct embb::base::Allocator< Type >::rebind< OtherType >

Rebind allocator to type OtherType.

6.60.2 Member Typedef Documentation

6.60.2.1 template<typename Type> template<typename OtherType > typedef Allocator<OtherType> embb::base::Allocator< Type >::rebind< OtherType >::other

Type to rebind to.

6.61 embb::base::AllocatorCacheAligned< Type >::rebind< OtherType > Struct Template Reference

Rebind allocator to type OtherType.

```
#include <memory_allocation.h>
```

Public Types

- typedef Allocator< OtherType > other

    Type to rebind to.

6.61.1 Detailed Description

template<typename Type>
template<typename OtherType>
struct embb::base::AllocatorCacheAligned< Type >::rebind< OtherType >

Rebind allocator to type OtherType.

6.61.2 Member Typedef Documentation

6.61.2.1 template<typename Type> template<typename OtherType > typedef Allocator<OtherType> embb::base::AllocatorCacheAligned< Type >::rebind< OtherType >::other

Type to rebind to.

6.62 embb::base::RecursiveMutex Class Reference

Recursive, exclusive mutex.

```
#include <mutex.h>
```

Public Member Functions

- RecursiveMutex ()

  Creates a mutex which is in unlocked state.
- void Lock ()

  Waits until the mutex can be locked and locks it.
- bool TryLock ()

  Tries to lock the mutex and returns immediately.
- void Unlock ()

  Unlocks a locked mutex.

6.62.1 Detailed Description

Recursive, exclusive mutex.

Mutexes of this type can be locked recursively, that is, multiple times by the same thread without unlocking it in between. It is unlocked only, if the number of unlock operations has reached the number of previous lock operations by the same thread. It cannot be copied or assigned.

See also

Mutex

Implemented concepts:

Mutex Concept

6.62.2 Constructor & Destructor Documentation

6.62.2.1 embb::base::RecursiveMutex::RecursiveMutex ( )

Creates a mutex which is in unlocked state.

Dynamic memory allocation

Potentially allocates dynamic memory

Concurrency

Not thread-safe

6.62.3 Member Function Documentation

6.62.3.1 void embb::base::RecursiveMutex::Lock ( )

Waits until the mutex can be locked and locks it.

Postcondition

The mutex is locked

Concurrency

Thread-safe

See also

TryLock(), Unlock()

6.62.3.2 bool embb::base::RecursiveMutex::TryLock ( )

Tries to lock the mutex and returns immediately.

Postcondition

If successful, the given mutex is locked.

Returns

`true` if the mutex could be locked, otherwise `false`.

Concurrency

Thread-safe

See also

Lock(), Unlock()

6.62.3.3 void embb::base::RecursiveMutex::Unlock ( )

Unlocks a locked mutex.

Precondition

The mutex is locked by the current thread.

Postcondition

The mutex is unlocked if the number of unlock operations has reached the number of lock operations.

Concurrency

Thread-safe

See also

Lock(), TryLock()

6.63 embb::base::ResourceBusyException Class Reference

Indicates business (unavailability) of a required resource.

```
#include <exceptions.h>
```

Public Member Functions

- ResourceBusyException (const char ∗message)

    Constructs an exception with the specified message.
- virtual int Code () const

    Returns an integer code representing the exception.
- virtual const char ∗ What () const throw ()

    Returns the error message.

6.63.1 Detailed Description

Indicates business (unavailability) of a required resource.

6.63.2 Constructor & Destructor Documentation

6.63.2.1 embb::base::ResourceBusyException::ResourceBusyException (const char ∗ *message*) [explicit]

Constructs an exception with the specified message.

Parameters

in	*message*	Error message

6.63.3 Member Function Documentation

6.63.3.1 virtual int embb::base::ResourceBusyException::Code ( ) const [virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implements embb::base::Exception.

6.63.3.2 virtual const char∗ embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.64 embb::dataflow::Network::Select< Type > Class Template Reference

Select process template.

`#include <network.h>`

Public Types

- typedef embb::base::Function< void, bool, Type, Type, Type & > FunctionType

    Function type to use when processing tokens.
- typedef Inputs< bool, Type, Type > InputsType

    Input port type list.
- typedef Outputs< Type > OutputsType

    Output port type list.

Public Member Functions

- Select (Network &network)

    Constructs a Select process.
- Select (Network &network, embb::mtapi::ExecutionPolicy const &policy)

    Constructs a Select process.
- virtual bool HasInputs () const
- InputsType & GetInputs ()
- template<int Index>
  InputsType::Types< Index >::Result & GetInput ()
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)

    Connects output port 0 to input port 0 of `target`.

6.64.1 Detailed Description

template<typename Type>
class embb::dataflow::Network::Select< Type >

Select process template.

A select has 3 inputs and 1 output. Input port 0 is of type boolean and selects which of input port 1 or 2 (of type `Type`) is sent to output port 0 (of type `Type`). If input port 0 is set to true the value of input port 1 is selected, otherwise the value of input port 2 is taken. Tokens are processed as soon as all inputs for that token are complete.

See also

Switch

Template Parameters

Type	The type of input port 1 and 2 and output port 0.

6.64.2 Member Typedef Documentation

6.64.2.1 template< typename Type > typedef embb::base::Function<void, bool, Type, Type, Type &> embb::dataflow::Network::Select< Type >::FunctionType

Function type to use when processing tokens.

6.64.2.2 template< typename Type > typedef Inputs<bool, Type, Type> embb::dataflow::Network::Select< Type >::InputsType

Input port type list.

6.64.2.3 template< typename Type > typedef Outputs<Type> embb::dataflow::Network::Select< Type >::OutputsType

Output port type list.

6.64.3 Constructor & Destructor Documentation

6.64.3.1 template< typename Type > embb::dataflow::Network::Select< Type >::Select (Network & *network*) [explicit]

Constructs a Select process.

Parameters

network	The network this node is going to be part of.

6.64.3.2 template< typename Type > embb::dataflow::Network::Select< Type >::Select (Network & *network*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a Select process.

Parameters

network	The network this node is going to be part of.
policy	The execution policy of the process.

6.64.4 Member Function Documentation

6.64.4.1 template<typename Type > virtual bool embb::dataflow::Network::Select< Type >::HasInputs ( ) const [virtual]

Returns

Always `true`.

6.64.4.2 template<typename Type > InputsType& embb::dataflow::Network::Select< Type >::GetInputs ( )

Returns

Reference to a list of all input ports.

6.64.4.3 template<typename Type > template<int Index> InputsType::Types<Index>::Result& embb::dataflow::Network::Select< Type >::GetInput ( )

Returns

Input port at Index.

6.64.4.4 template<typename Type > virtual bool embb::dataflow::Network::Select< Type >::HasOutputs ( ) const [virtual]

Returns

Always `true`.

6.64.4.5 template<typename Type > OutputsType& embb::dataflow::Network::Select< Type >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.64.4.6 template<typename Type > template<int Index> OutputsType::Types<Index>::Result& embb::dataflow::Network::Select< Type >::GetOutput ( )

Returns

Output port at Index.

6.64.4.7 template<typename Type > template<typename T > void embb::dataflow::Network::Select< Type >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.65 embb::dataflow::Network::SerialProcess< Inputs, Outputs > Class Template Reference

Generic serial process template.

```
#include <network.h>
```

Public Types

- typedef embb::base::Function< void, INPUT_TYPE_LIST, OUTPUT_TYPE_LIST > FunctionType

  Function type to use when processing tokens.
- typedef Inputs< INPUT_TYPE_LIST > InputsType

  Input port type list.
- typedef Outputs< OUTPUT_TYPE_LIST > OutputsType

  Output port type list.

Public Member Functions

- SerialProcess (Network &network, FunctionType function)

  Constructs a SerialProcess with a user specified processing function.
- SerialProcess (Network &network, embb::mtapi::Job job)

  Constructs a SerialProcess with a user specified embb::mtapi::Job.
- SerialProcess (Network &network, FunctionType function, embb::mtapi::ExecutionPolicy const &policy)

  Constructs a SerialProcess with a user specified processing function.
- SerialProcess (Network &network, embb::mtapi::Job job, embb::mtapi::ExecutionPolicy const &policy)

  Constructs a SerialProcess with a user specified embb::mtapi::Job.
- virtual bool HasInputs () const
- InputsType & GetInputs ()
- template<int Index>
  InputsType::Types< Index >::Result & GetInput ()
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)

  Connects output port 0 to input port 0 of `target`.

6.65.1 Detailed Description

template<class Inputs, class Outputs>
class embb::dataflow::Network::SerialProcess< Inputs, Outputs >

Generic serial process template.

Implements a generic serial process in the network that may have one to four input ports and one to four output ports but no more that five total ports. Tokens are processed in order.

See also

Source, ParallelProcess, Sink, Switch, Select

Template Parameters

Inputs	Inputs of the process.
Outputs	Outputs of the process.

6.65.2 Member Typedef Documentation

6.65.2.1 template<class Inputs , class Outputs > typedef embb::base::Function<void, INPUT_TYPE_LIST, OUTPUT_TYPE_LIST> embb::dataflow::Network::SerialProcess< Inputs, Outputs >::FunctionType

Function type to use when processing tokens.

6.65.2.2 template<class Inputs , class Outputs > typedef Inputs<INPUT_TYPE_LIST> embb::dataflow::Network::SerialProcess< Inputs, Outputs >::InputsType

Input port type list.

6.65.2.3 template<class Inputs , class Outputs > typedef Outputs<OUTPUT_TYPE_LIST> embb::dataflow::Network::SerialProcess< Inputs, Outputs >::OutputsType

Output port type list.

6.65.3 Constructor & Destructor Documentation

6.65.3.1 template<class Inputs , class Outputs > embb::dataflow::Network::SerialProcess< Inputs, Outputs >::SerialProcess (Network & *network*, FunctionType *function*)

Constructs a SerialProcess with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.

6.65.3.2 template<class Inputs , class Outputs > embb::dataflow::Network::SerialProcess< Inputs, Outputs >::SerialProcess (Network & *network*, embb::mtapi::Job *job*)

Constructs a SerialProcess with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.

6.65.3.3 template<class Inputs , class Outputs > embb::dataflow::Network::SerialProcess< Inputs, Outputs >::SerialProcess (Network & *network*, FunctionType *function*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a SerialProcess with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.
policy	The execution policy of the process.

6.65.3.4 template<class Inputs , class Outputs > embb::dataflow::Network::SerialProcess< Inputs, Outputs >::SerialProcess (Network & *network*, embb::mtapi::Job *job*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a SerialProcess with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.
policy	The execution policy of the process.

6.65.4 Member Function Documentation

6.65.4.1 template<class Inputs , class Outputs > virtual bool embb::dataflow::Network::SerialProcess< Inputs, Outputs >::HasInputs ( ) const [virtual]

Returns

`true` if the SerialProcess has any inputs, `false` otherwise.

6.65.4.2 template<class Inputs , class Outputs > InputsType& embb::dataflow::Network::SerialProcess< Inputs, Outputs >::GetInputs ( )

Returns

Reference to a list of all input ports.

6.65.4.3 template<class Inputs , class Outputs > template<int Index> InputsType::Types<Index>::Result& embb::dataflow::Network::SerialProcess< Inputs, Outputs >::GetInput ( )

Returns

Input port at Index.

6.65.4.4 template<class Inputs , class Outputs > virtual bool embb::dataflow::Network::SerialProcess< Inputs, Outputs >::HasOutputs ( ) const [virtual]

Returns

`true` if the SerialProcess has any outputs, `false` otherwise.

6.65.4.5 template<class Inputs , class Outputs > OutputsType& embb::dataflow::Network::SerialProcess< Inputs, Outputs >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.65.4.6 template<class Inputs , class Outputs > template<int Index> OutputsType::Types<Index>::Result& embb::dataflow::Network::SerialProcess< Inputs, Outputs >::GetOutput ( )

Returns

Output port at Index.

6.65.4.7 template<class Inputs , class Outputs > template<typename T > void embb::dataflow::Network::Serial↩Process< Inputs, Outputs >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.66 embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 > Class Template Reference

Sink process template.

```
#include <network.h>
```

Public Types

- typedef embb::base::Function< void, INPUT_TYPE_LIST > FunctionType

    Function type to use when processing tokens.
- typedef Inputs< INPUT_TYPE_LIST > InputsType

    Input port type list.

Public Member Functions

- Sink (Network &network, FunctionType function)

    Constructs a Sink with a user specified processing function.
- Sink (Network &network, embb::mtapi::Job job)

    Constructs a Sink with a user specified embb::mtapi::Job.
- Sink (Network &network, FunctionType function, embb::mtapi::ExecutionPolicy const &policy)

    Constructs a Sink with a user specified processing function.
- Sink (Network &network, embb::mtapi::Job job, embb::mtapi::ExecutionPolicy const &policy)

    Constructs a Sink with a user specified embb::mtapi::Job.
- virtual bool HasInputs () const
- InputsType & GetInputs ()
- template<int Index>
  InputsType::Types< Index >::Result & GetInput ()
- virtual bool HasOutputs () const

6.66.1 Detailed Description

template<typename I1, typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb↩ ::base::internal::Nil, typename I5 = embb::base::internal::Nil>
class embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >

Sink process template.

A sink marks the end of a particular processing chain. It can have one to five input ports and no output ports. Tokens are processed in order by the sink, regardless in which order they arrive at the input ports.

See also

Source, SerialProcess, ParallelProcess

Template Parameters

I1	Type of first input port.
I2	Optional type of second input port.
I3	Optional type of third input port.
I4	Optional type of fourth input port.
I5	Optional type of fifth input port.

6.66.2 Member Typedef Documentation

6.66.2.1 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> typedef **embb::base::Function**<void, INPUT_TYPE_LIST> **embb::dataflow::Network::Sink**< I1, I2, I3, I4, I5 >::**FunctionType**

Function type to use when processing tokens.

6.66.2.2 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> typedef **Inputs**<INPUT_TYPE_LIST> **embb::dataflow::Network::Sink**< I1, I2, I3, I4, I5 >::**InputsType**

Input port type list.

6.66.3 Constructor & Destructor Documentation

6.66.3.1 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> **embb::dataflow::Network::Sink**< I1, I2, I3, I4, I5 >::**Sink** (**Network** & *network*, **FunctionType** *function*)

Constructs a Sink with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.

6.66.3.2 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> **embb::dataflow::Network::Sink**< I1, I2, I3, I4, I5 >::**Sink** (**Network** & *network*, **embb::mtapi::Job** *job*)

Constructs a Sink with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a null pointer as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.

6.66.3.3 template< typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::Sink (Network & *network*, FunctionType *function*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a Sink with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to process a token.
policy	The execution policy of the process.

6.66.3.4 template< typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::Sink (Network & *network*, embb::mtapi::Job *job*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a Sink with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a struct containing copies of the inputs as its argument buffer and a null pointer as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.
policy	The execution policy of the process.

6.66.4 Member Function Documentation

6.66.4.1 template< typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> virtual bool embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::HasInputs ( ) const [virtual]

Returns

Always `true`.

6.66.4.2 template< typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> InputsType& embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::GetInputs ( )

Returns

Reference to a list of all input ports.

6.66.4.3 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> template<int Index> InputsType::Types<Index>::Result& embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::GetInput ( )

Returns

Input port at Index.

6.66.4.4 template<typename I1 , typename I2 = embb::base::internal::Nil, typename I3 = embb::base::internal::Nil, typename I4 = embb::base::internal::Nil, typename I5 = embb::base::internal::Nil> virtual bool embb::dataflow::Network::Sink< I1, I2, I3, I4, I5 >::HasOutputs ( ) const [virtual]

Returns

Always `false`.

6.67 embb::dataflow::Network::Source< O1, O2, O3, O4, O5 > Class Template Reference

Source process template.

`#include <network.h>`

Public Types

- typedef embb::base::Function< void, OUTPUT_TYPE_LIST > FunctionType
  Function type to use when processing tokens.
- typedef Outputs< OUTPUT_TYPE_LIST > OutputsType
  Output port type list.

Public Member Functions

- Source (Network &network, FunctionType function)
  Constructs a Source with a user specified processing function.
- Source (Network &network, embb::mtapi::Job job)
  Constructs a Source with a user specified embb::mtapi::Job.
- Source (Network &network, FunctionType function, embb::mtapi::ExecutionPolicy const &policy)
  Constructs a Source with a user specified processing function.
- Source (Network &network, embb::mtapi::Job job, embb::mtapi::ExecutionPolicy const &policy)
  Constructs a Source with a user specified embb::mtapi::Job.
- virtual bool HasInputs () const
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)
  Connects output port 0 to input port 0 of `target`.

6.67.1 Detailed Description

template< typename O1, typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil>
class embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >

Source process template.

A source marks the start of a processing chain. It can have one to five output ports and no input ports. Tokens are emitted in order by the source.

See also

SerialProcess, ParallelProcess, Sink

Template Parameters

O1	Type of first output port.
O2	Optional type of second output port.
O3	Optional type of third output port.
O4	Optional type of fourth output port.
O5	Optional type of fifth output port.

6.67.2 Member Typedef Documentation

6.67.2.1 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> typedef embb::base::Function<void, OUTPUT_TYPE_LIST> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::FunctionType

Function type to use when processing tokens.

6.67.2.2 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> typedef Outputs<OUTPUT_TYPE_LIST> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::OutputsType

Output port type list.

6.67.3 Constructor & Destructor Documentation

6.67.3.1 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::Source (Network & *network,* FunctionType *function*)

Constructs a Source with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to emit a token.

6.67.3.2 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::Source (Network & *network*, embb::mtapi::Job *job*)

Constructs a Source with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a null pointer as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.

6.67.3.3 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::Source (Network & *network*, FunctionType *function*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a Source with a user specified processing function.

Parameters

network	The network this node is going to be part of.
function	The Function to call to emit a token.
policy	The execution policy of the process.

6.67.3.4 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::Source (Network & *network*, embb::mtapi::Job *job*, embb::mtapi::ExecutionPolicy const & *policy*)

Constructs a Source with a user specified embb::mtapi::Job.

The Job must be associated with an action function accepting a null pointer as its argument buffer and a struct containing the outputs as its result buffer.

Parameters

network	The network this node is going to be part of.
job	The embb::mtapi::Job to process a token.
policy	The execution policy of the process.

6.67.4 Member Function Documentation

6.67.4.1 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> virtual bool embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::HasInputs ( ) const [virtual]

Returns

Always `false`.

6.67.4.2 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> virtual bool embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::HasOutputs ( ) const [virtual]

Returns

Always `true`.

6.67.4.3 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> OutputsType& embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.67.4.4 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> template<int Index> OutputsType::Types<Index>::Result& embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::GetOutput ( )

Returns

Output port at INDEX.

6.67.4.5 template<typename O1 , typename O2 = embb::base::internal::Nil, typename O3 = embb::base::internal::Nil, typename O4 = embb::base::internal::Nil, typename O5 = embb::base::internal::Nil> template<typename T > void embb::dataflow::Network::Source< O1, O2, O3, O4, O5 >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.68 embb::base::Spinlock Class Reference

Spinlock.

```
#include <mutex.h>
```

Public Member Functions

- Spinlock ()
  Creates a spinlock which is in unlocked state.
- ∼Spinlock ()
  Destructs a spinlock.
- void Lock ()
  Waits until the spinlock can be locked and locks it.
- bool TryLock (unsigned int number_spins=1)
  Tries to lock the spinlock for `number_spins` times and returns.
- void Unlock ()
  Unlocks the spinlock.

6.68.1 Detailed Description

Spinlock.

Implemented concepts:

Mutex Concept

6.68.2 Constructor & Destructor Documentation

6.68.2.1 embb::base::Spinlock::Spinlock ( )

Creates a spinlock which is in unlocked state.

Concurrency

Not thread-safe

6.68.2.2 embb::base::Spinlock::∼Spinlock ( )

Destructs a spinlock.

Concurrency

Not thread-safe

6.68.3 Member Function Documentation

6.68.3.1 void embb::base::Spinlock::Lock ( )

Waits until the spinlock can be locked and locks it.

Note

This method yields the current thread in regular, implementation-defined intervals.

Precondition

The spinlock is not locked by the current thread.

Postcondition

The spinlock is locked.

Concurrency

Thread-safe

See also

TryLock(), Unlock()

6.68.3.2 bool embb::base::Spinlock::TryLock (unsigned int *number_spins* = `1`)

Tries to lock the spinlock for `number_spins` times and returns.

Precondition

The spinlock is not locked by the current thread.

Postcondition

If successful, the spinlock is locked.

Returns

`true` if the spinlock could be locked, otherwise `false`.

Concurrency

Thread-safe

See also

Lock(), Unlock()

Parameters

in	*number_spins*	Maximal number of spins when trying to acquire the lock. Note that passing 0 here results in not trying to obtain the lock at all. The default parameter is 1.

6.68.3.3 void embb::base::Spinlock::Unlock ( )

Unlocks the spinlock.

Precondition

The spinlock is locked by the current thread.

Postcondition

The spinlock is unlocked.

Concurrency

Thread-safe

See also

Lock(), TryLock()

6.69 embb::mtapi::StatusException Class Reference

Represents an MTAPI error state and is thrown by almost all mtapi_cpp methods.

`#include <status_exception.h>`

Public Member Functions

- StatusException (const char ∗message)

  Constructs a StatusException.
- virtual int Code () const

  Code associated with this exception.
- virtual const char ∗ What () const throw ()

  Returns the error message.

6.69.1 Detailed Description

Represents an MTAPI error state and is thrown by almost all mtapi_cpp methods.

6.69.2 Constructor & Destructor Documentation

6.69.2.1 embb::mtapi::StatusException::StatusException (const char ∗ *message*) [explicit]

Constructs a StatusException.

Concurrency

Not thread-safe

Parameters

message	The message to use.

6.69.3 Member Function Documentation

6.69.3.1 virtual int embb::mtapi::StatusException::Code ( ) const [virtual]

Code associated with this exception.

Returns

An integer representing the code of the exception

Concurrency

Thread-safe and wait-free

Implements embb::base::Exception.

6.69.3.2 virtual const char∗ embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.70 embb::dataflow::Network::Switch< Type > Class Template Reference

Switch process template.

`#include <network.h>`

Public Types

- typedef embb::base::Function< void, bool, Type, Type & > FunctionType

  Function type to use when processing tokens.
- typedef Inputs< bool, Type > InputsType

  Input port type list.
- typedef Outputs< Type > OutputsType

  Output port type list.

Public Member Functions

- Select (Network &network)

    Constructs a Switch process.
- Select (Network &network, embb::mtapi::ExecutionPolicy const &policy)

    Constructs a Switch process.
- virtual bool HasInputs () const
- InputsType & GetInputs ()
- template<int Index>
  InputsType::Types< Index >::Result & GetInput ()
- virtual bool HasOutputs () const
- OutputsType & GetOutputs ()
- template<int Index>
  OutputsType::Types< Index >::Result & GetOutput ()
- template<typename T >
  void operator>> (T &target)

    Connects output port 0 to input port 0 of `target`.

6.70.1 Detailed Description

template<typename Type>
class embb::dataflow::Network::Switch< Type >

Switch process template.

A switch has 2 inputs and 2 outputs. Input port 0 is of type boolean and selects to which output port the value of input port 1 of type `Type` is sent. If input port 0 is set to true the value goes to output port 0 and to output port 1 otherwise. Tokens are processed as soon as all inputs for that token are complete.

See also

Select

Template Parameters

Type	The type of input port 1 and output port 0 and 1.

6.70.2 Member Typedef Documentation

6.70.2.1 template<typename Type > typedef embb::base::Function<void, bool, Type, Type &> embb::dataflow::Network::Switch< Type >::FunctionType

Function type to use when processing tokens.

6.70.2.2 template<typename Type > typedef Inputs<bool, Type> embb::dataflow::Network::Switch< Type >::InputsType

Input port type list.

6.70.2.3 template<typename Type > typedef **Outputs**<Type> **embb::dataflow::Network::Switch**< Type >::**OutputsType**

Output port type list.

6.70.3 Member Function Documentation

6.70.3.1 template<typename Type > **embb::dataflow::Network::Switch**< Type >::**Select** (**Network** & *network*) [explicit]

Constructs a Switch process.

Parameters

network	The network this node is going to be part of.

6.70.3.2 template<typename Type > **embb::dataflow::Network::Switch**< Type >::**Select** (**Network** & *network*, **embb::mtapi::ExecutionPolicy** const & *policy*)

Constructs a Switch process.

Parameters

network	The network this node is going to be part of.
policy	The execution policy of the process.

6.70.3.3 template<typename Type > virtual bool **embb::dataflow::Network::Switch**< Type >::HasInputs ( ) const [virtual]

Returns

Always `true`.

6.70.3.4 template<typename Type > **InputsType& embb::dataflow::Network::Switch**< Type >::GetInputs ( )

Returns

Reference to a list of all input ports.

6.70.3.5 template<typename Type > template<int Index> **InputsType::Types**<Index>::Result& **embb::dataflow::Network::Switch**< Type >::GetInput ( )

Returns

Input port at Index.

6.70.3.6 template<typename Type > virtual bool **embb::dataflow::Network::Switch**< Type >::HasOutputs ( ) const [virtual]

Returns

Always `true`.

6.70.3.7 template<typename Type > **OutputsType& embb::dataflow::Network::Switch**< Type >::GetOutputs ( )

Returns

Reference to a list of all output ports.

6.70.3.8 template<typename Type > template<int Index> **OutputsType::Types**<Index>::Result& **embb::dataflow::Network::Switch**< Type >::GetOutput ( )

Returns

Output port at Index.

6.70.3.9 template<typename Type > template<typename T > void **embb::dataflow::Network::Switch**< Type >::operator>> (T & *target*)

Connects output port 0 to input port 0 of `target`.

Parameters

target	Process to connect to.

Template Parameters

T	Type of target process.

6.71 embb::mtapi::Task Class Reference

A Task represents a running Action of a specific Job.

`#include <task.h>`

Public Member Functions

- Task ()

  Constructs an invalid Task.
- Task (Task const &other)

Copies a Task.
- void operator= (Task const &other)

    Copies a Task.
- ∼Task ()

    Destroys a Task.
- mtapi_status_t Wait (mtapi_timeout_t timeout)

    Waits for Task to finish for `timeout` milliseconds.
- mtapi_status_t Wait ()

    Waits for Task to finish.
- void Cancel ()

    Signals the Task to cancel computation.
- mtapi_task_hndl_t GetInternal () const

    Returns the internal representation of this object.

6.71.1 Detailed Description

A Task represents a running Action of a specific Job.

6.71.2 Constructor & Destructor Documentation

6.71.2.1 embb::mtapi::Task::Task ( )

Constructs an invalid Task.

Concurrency

Thread-safe and wait-free

6.71.2.2 embb::mtapi::Task::Task (Task const & *other*)

Copies a Task.

Concurrency

Thread-safe and wait-free

Parameters

other	The task to copy.

6.71.2.3 embb::mtapi::Task::∼Task ( )

Destroys a Task.

Concurrency

Thread-safe and wait-free

6.71.3 Member Function Documentation

6.71.3.1 void embb::mtapi::Task::operator= (Task const & *other*)

Copies a Task.

Concurrency

Thread-safe and wait-free

Parameters

other	The task to copy.

6.71.3.2 mtapi_status_t embb::mtapi::Task::Wait (mtapi_timeout_t *timeout*)

Waits for Task to finish for `timeout` milliseconds.

Returns

The status of the finished Task, `MTAPI_TIMEOUT` or `MTAPI_ERR_*`

Concurrency

Thread-safe

Parameters

`in`	*timeout*	Timeout duration in milliseconds

6.71.3.3 mtapi_status_t embb::mtapi::Task::Wait ( )

Waits for Task to finish.

Returns

The status of the finished Task or `MTAPI_ERR_*`

Concurrency

Thread-safe

6.71.3.4 void embb::mtapi::Task::Cancel ( )

Signals the Task to cancel computation.

Concurrency

Thread-safe and wait-free

6.71.3.5 mtapi_task_hndl_t embb::mtapi::Task::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

The internal mtapi_task_hndl_t.

Concurrency

Thread-safe and wait-free

6.72 embb::mtapi::TaskAttributes Class Reference

Contains attributes of a Task.

```
#include <task_attributes.h>
```

Public Member Functions

- TaskAttributes ()

  Constructs a TaskAttributes object.
- TaskAttributes & SetDetached (bool state)

  Sets the detached property of a Task.
- TaskAttributes & SetPriority (mtapi_uint_t priority)

  Sets the priority of a Task.
- TaskAttributes & SetAffinity (mtapi_affinity_t affinity)

  Sets the affinity of a Task.
- TaskAttributes & SetPolicy (ExecutionPolicy const &policy)

  Sets the ExecutionPolicy of a Task.
- TaskAttributes & SetInstances (mtapi_uint_t instances)

  Sets the number of instances in a Task.
- mtapi_task_attributes_t const & GetInternal () const

  Returns the internal representation of this object.

6.72.1 Detailed Description

Contains attributes of a Task.

6.72.2 Constructor & Destructor Documentation

6.72.2.1 embb::mtapi::TaskAttributes::TaskAttributes ( )

Constructs a TaskAttributes object.

Concurrency

Not thread-safe

6.72.3 Member Function Documentation

6.72.3.1 TaskAttributes& embb::mtapi::TaskAttributes::SetDetached (bool *state*)

Sets the detached property of a Task.

If set to `true`, the started Task will have no handle and cannot be waited for.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

state	The state to set.

6.72.3.2 TaskAttributes& embb::mtapi::TaskAttributes::SetPriority (mtapi_uint_t *priority*)

Sets the priority of a Task.

The priority influences the order in which tasks are chosen for execution.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

priority	The priority to set.

6.72.3.3 TaskAttributes& embb::mtapi::TaskAttributes::SetAffinity (mtapi_affinity_t *affinity*)

Sets the affinity of a Task.

The affinity influences on which worker the Task will be executed.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

affinity	The affinity to set.

6.72.3.4 TaskAttributes& embb::mtapi::TaskAttributes::SetPolicy (ExecutionPolicy const & *policy*)

Sets the ExecutionPolicy of a Task.

The ExecutionPolicy determines the affinity and priority of a Task.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

policy	The ExecutionPolicy to set.

6.72.3.5 TaskAttributes& embb::mtapi::TaskAttributes::SetInstances (mtapi_uint_t *instances*)

Sets the number of instances in a Task.

The Task will be launched `instances` times. In the action function, the number of instances and the current instance can be queried from the TaskContext.

Returns

Reference to this object.

Concurrency

Not thread-safe

Parameters

instances	Number of instances to set.

6.72.3.6 mtapi_task_attributes_t const& embb::mtapi::TaskAttributes::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A reference to the internal mtapi_task_attributes_t structure.

Concurrency

Thread-safe and wait-free

6.73 embb::mtapi::TaskContext Class Reference

Provides information about the status of the currently running Task.

```
#include <task_context.h>
```

Public Member Functions

- TaskContext (mtapi_task_context_t ∗task_context)

  Constructs a TaskContext from the given C representation.
- bool ShouldCancel ()

  Queries whether the Task running in the TaskContext should finish.
- mtapi_task_state_t GetTaskState ()

  Queries the current Task state.
- mtapi_uint_t GetCurrentWorkerNumber ()

  Queries the index of the worker thread the Task is running on.
- mtapi_uint_t GetInstanceNumber ()

  Queries the current instance of the currently executing Task.
- mtapi_uint_t GetNumberOfInstances ()

  Queries the number of instances of the currently executing Task.
- void SetStatus (mtapi_status_t error_code)

  Sets the return status of the running Task.
- mtapi_task_context_t ∗ GetInternal () const

  Returns the internal representation of this object.

6.73.1 Detailed Description

Provides information about the status of the currently running Task.

6.73.2 Constructor & Destructor Documentation

6.73.2.1 embb::mtapi::TaskContext::TaskContext (mtapi_task_context_t * *task_context*) [explicit]

Constructs a TaskContext from the given C representation.

Concurrency

Thread-safe and wait-free

Parameters

task_context	The C task context to wrap.

6.73.3 Member Function Documentation

6.73.3.1 bool embb::mtapi::TaskContext::ShouldCancel ( )

Queries whether the Task running in the TaskContext should finish.

Returns

`true` if the Task should finish, otherwise `false`

Concurrency

Not thread-safe

6.73.3.2 mtapi_task_state_t embb::mtapi::TaskContext::GetTaskState ( )

Queries the current Task state.

Returns

The current Task state.

Concurrency

Not thread-safe

6.73.3.3 mtapi_uint_t embb::mtapi::TaskContext::GetCurrentWorkerNumber ( )

Queries the index of the worker thread the Task is running on.

Returns

The worker thread index the Task is running on

Concurrency

Not thread-safe

6.73.3.4 mtapi_uint_t embb::mtapi::TaskContext::GetInstanceNumber ( )

Queries the current instance of the currently executing Task.

Returns

The current instance number

Concurrency

Not thread-safe

6.73.3.5 mtapi_uint_t embb::mtapi::TaskContext::GetNumberOfInstances ( )

Queries the number of instances of the currently executing Task.

Returns

The number of instances

Concurrency

Not thread-safe

6.73.3.6 void embb::mtapi::TaskContext::SetStatus (mtapi_status_t *error_code*)

Sets the return status of the running Task.

This will be returned by Task::Wait() and is set to `MTAPI_SUCCESS` by default.

Concurrency

Not thread-safe

Parameters

in	*error_code*	The status to return by Task::Wait(), Group::WaitAny(), Group::WaitAll()

6.73.3.7 mtapi_task_context_t∗ embb::mtapi::TaskContext::GetInternal ( ) const

Returns the internal representation of this object.

Allows for interoperability with the C interface.

Returns

A pointer to a mtapi_task_context_t.

Concurrency

Not thread-safe

6.74 embb::base::Thread Class Reference

Represents a thread of execution.

```
#include <thread.h>
```

Classes

- class ID

    Unique ID of a thread that can be compared with other IDs.

Public Member Functions

- template<typename Function >
  Thread (Function function)

    Creates and runs a thread with zero-argument start function.
- template<typename Function >
  Thread (CoreSet &core_set, Function function)

    Creates and runs a thread with zero-argument start function.
- template<typename Function >
  Thread (CoreSet &core_set, embb_thread_priority_t priority, Function function)

    Creates and runs a thread with zero-argument start function.
- template<typename Function , typename Arg >
  Thread (Function function, Arg arg)

    Creates and runs a thread with one-argument thread start function.
- template<typename Function , typename Arg1 , typename Arg2 >
  Thread (Function function, Arg1 arg1, Arg2 arg2)

    Creates and runs a thread with two-argument thread start function.
- void Join ()

    Waits until the thread has finished execution.
- ID GetID ()

    Returns the thread ID.

Static Public Member Functions

- static unsigned int GetThreadsMaxCount ()

    Returns the maximum number of threads handled by EMB^2.
- static void SetThreadsMaxCount (unsigned int max_count)

    Sets the maximum number of threads handled by EMB^2.
- static ID CurrentGetID ()

    Returns the ID of the current thread.
- static void CurrentYield ()

    Reschedule the current thread for later execution.

6.74.1 Detailed Description

Represents a thread of execution.

Provides an abstraction from platform-specific threading implementations to create, manage, and join threads of execution. Support for thread-to-core affinities is given on thread creation by using the core set functionality.

This class is essentially a wrapper for the underlying C implementation.

6.74.2 Constructor & Destructor Documentation

6.74.2.1 template<typename Function > embb::base::Thread::Thread (Function *function*) [explicit]

Creates and runs a thread with zero-argument start function.

Note

If the function is passed as a temporary object when creating a thread, this might be interpreted as a function declaration ("most vexing parse"). C++11 resolves this by using curly braces for initialization.

Exceptions

NoMemoryException	if not enough memory is available
ErrorException	in case of another error

Dynamic memory allocation

A small constant amount of memory to store the function. This memory is freed the thread is joined.

Concurrency

Not thread-safe

Template Parameters

Function	Function object type

Parameters

in	*function*	Copyable function object, callable without arguments

6.74.2.2 template<typename Function > embb::base::Thread::Thread (CoreSet & *core_set,* Function *function*)

Creates and runs a thread with zero-argument start function.

Note

If the function is passed as a temporary object when creating a thread, this might be interpreted as a function declaration ("most vexing parse"). C++11 resolves this by using curly braces for initialization.

Exceptions

NoMemoryException	if not enough memory is available
ErrorException	in case of another error

Dynamic memory allocation

A small constant amount of memory to store the function. This memory is freed the thread is joined.

Concurrency

Not thread-safe

Template Parameters

Function	Function object type

Parameters

in	*core_set*	Set of cores on which the thread shall be executed.
in	*function*	Copyable function object, callable without arguments

6.74.2.3 template<typename Function > embb::base::Thread::Thread (CoreSet & *core_set*, embb_thread_priority_t *priority*, Function *function*)

Creates and runs a thread with zero-argument start function.

Note

If the function is passed as a temporary object when creating a thread, this might be interpreted as a function declaration ("most vexing parse"). C++11 resolves this by using curly braces for initialization.

Exceptions

NoMemoryException	if not enough memory is available
ErrorException	in case of another error

Dynamic memory allocation

A small constant amount of memory to store the function. This memory is freed the thread is joined.

Concurrency

Not thread-safe

Template Parameters

Function	Function object type

Parameters

in	*core_set*	Set of cores on which the thread shall be executed.
in	*priority*	Priority of the new thread.

Parameters

in	*function*	Copyable function object, callable without arguments

6.74.2.4 template<typename Function , typename Arg > embb::base::Thread::Thread (Function *function,* Arg *arg*)

Creates and runs a thread with one-argument thread start function.

Note

If the function is passed as a temporary object when creating a thread, this might be interpreted as a function declaration ("most vexing parse"). C++11 resolves this by using curly braces for initialization.

Exceptions

NoMemoryException	if not enough memory is available
ErrorException	in case of another error

Dynamic memory allocation

A small constant amount of memory to store the function. This memory is freed the thread is joined.

Concurrency

Not thread-safe

Template Parameters

Function	Function object type
Argument	Type of argument

Parameters

in	*function*	Copyable function object, callable with one argument
in	*arg*	Argument for function (must be copyable)

6.74.2.5 template<typename Function , typename Arg1 , typename Arg2 > embb::base::Thread::Thread (Function *function,* Arg1 *arg1,* Arg2 *arg2*)

Creates and runs a thread with two-argument thread start function.

Note

If the function is passed as a temporary object when creating a thread, this might be interpreted as a function declaration ("most vexing parse"). C++11 resolves this by using curly braces for initialization.

Exceptions

NoMemoryException	if not enough memory is available
ErrorException	in case of another error

Dynamic memory allocation

A small constant amount of memory to store the function. This memory is freed the thread is joined.

Concurrency

Not thread-safe

Template Parameters

Function	Function object type
Arg1	Type of first argument
Arg2	Type of second argument

Parameters

`in`	*function*	Copyable function object, callable with two arguments
`in`	*arg1*	First argument for function (must be copyable)
`in`	*arg2*	Second argument for function (must be copyable)

6.74.3 Member Function Documentation

6.74.3.1 static unsigned int embb::base::Thread::GetThreadsMaxCount ( ) `[static]`

Returns the maximum number of threads handled by EMB^2.

See embb_thread_get_max_count() for a description of the semantics.

Returns

Maximum number of threads

Concurrency

Thread-safe and lock-free

See also

SetThreadsMaxCount()

6.74.3.2 static void embb::base::Thread::SetThreadsMaxCount (unsigned int *max_count*) [static]

Sets the maximum number of threads handled by EMB^2.

Concurrency

Not thread-safe

See also

GetThreadsMaxCount()

Parameters

in	*max_count*	Maximum number of threads

6.74.3.3 static ID embb::base::Thread::CurrentGetID ( ) [static]

Returns the ID of the current thread.

The ID is only valid within the calling thread.

Returns

ID of the calling thread

Concurrency

Thread-safe

6.74.3.4 static void embb::base::Thread::CurrentYield ( ) [static]

Reschedule the current thread for later execution.

This is only a request, the realization depends on the implementation and the scheduler employed by the operating system.

Concurrency

Thread-safe

6.74.3.5 void embb::base::Thread::Join ( )

Waits until the thread has finished execution.

Precondition

The thread has been created but not yet been joined.

Postcondition

The thread has finished execution and dynamic memory allocated during creation has been freed.

Concurrency

Not thread-safe

6.74.3.6 ID embb::base::Thread::GetID ( )

Returns the thread ID.

Returns

ID of the thread

Concurrency

Thread-safe

6.75 embb::base::ThreadSpecificStorage< Type > Class Template Reference

Represents thread-specific storage (TSS).

```
#include <thread_specific_storage.h>
```

Public Member Functions

- ThreadSpecificStorage ()

  Creates the TSS and default initializes all slots.
- template<typename Initializer1 , ... >
  ThreadSpecificStorage (Initializer1 initializer1,...)

  Creates the TSS and initializes all slots with up to four constructor arguments.
- ∼ThreadSpecificStorage ()

  Destroys the TSS and frees allocated memory for the TSS slots.
- Type & Get ()

  Returns a reference to the TSS slot of the current thread.
- const Type & Get () const

  Returns a const reference to the TSS slot of the current thread.

6.75.1 Detailed Description

template<typename Type>
class embb::base::ThreadSpecificStorage< Type >

Represents thread-specific storage (TSS).

Provides for each thread a separate slot storing an object of the given type.

Template Parameters

Type	Type of the objects

6.75.2 Constructor & Destructor Documentation

6.75.2.1 template<typename Type > embb::base::ThreadSpecificStorage< Type >::ThreadSpecificStorage ( )

Creates the TSS and default initializes all slots.

Exceptions

NoMemoryException	if not enough memory is available to allocate the TSS slots

Dynamic memory allocation

Dynamically allocates embb::base::Thread::GetThreadsMaxCount() pointers and slots of the TSS type

Concurrency

Not thread-safe

6.75.2.2 template<typename Type > template<typename Initializer1 , ... > embb::base::ThreadSpecificStorage< Type >::ThreadSpecificStorage (Initializer1 *initializer1, ...*) [explicit]

Creates the TSS and initializes all slots with up to four constructor arguments.

The TSS objects are created by calling their constructor as follows: Type(initializer1, ...).

Exceptions

NoMemoryException	if not enough memory is available to allocate the TSS slots

Dynamic memory allocation

Dynamically allocates embb::base::Thread::GetThreadsMaxCount() pointers and slots of the TSS type

Concurrency

Not thread-safe

Parameters

`in`	*initializer1*	First argument for constructor ...

6.75.2.3 template<typename Type > embb::base::ThreadSpecificStorage< Type >::∼ThreadSpecificStorage ( )

Destroys the TSS and frees allocated memory for the TSS slots.

6.75.3 Member Function Documentation

6.75.3.1 template< typename Type > Type& embb::base::ThreadSpecificStorage< Type >::Get ( )

Returns a reference to the TSS slot of the current thread.

Returns

Reference to TSS slot

Exceptions

embb::base::ErrorException	if the maximum number of threads has been exceeded

Concurrency

Thread-safe and lock-free

See also

Get() const

6.75.3.2 template< typename Type > const Type& embb::base::ThreadSpecificStorage< Type >::Get ( ) const

Returns a const reference to the TSS slot of the current thread.

Returns

Constant reference to TSS slot

Exceptions

embb::base::ErrorException	if the maximum number of threads has been exceeded

Concurrency

Thread-safe and lock-free

See also

Get()

6.76 embb::base::Time Class Reference

Represents an absolute time point.

```
#include <time.h>
```

Public Member Functions

- Time ()

    Constructs an instance representing the current point of time.
- template< typename Tick >
  Time (const Duration< Tick > &duration)

    Constructs an instance representing the current point of time plus `duration`.

6.76.1 Detailed Description

Represents an absolute time point.

6.76.2 Constructor & Destructor Documentation

6.76.2.1 embb::base::Time::Time ( )

Constructs an instance representing the current point of time.

Concurrency

Not thread-safe

6.76.2.2 template< typename Tick > embb::base::Time::Time (const Duration< Tick > & *duration*) [explicit]

Constructs an instance representing the current point of time plus `duration`.

Concurrency

Not thread-safe

See also

Duration

Template Parameters

Tick	Type of tick of the Duration

Parameters

`in`	*duration*	Duration added to the current point of time.

6.77 embb::base::TryLockTag Struct Reference

Tag type for try-lock UniqueLock construction.

```
#include <mutex.h>
```

6.77.1 Detailed Description

Tag type for try-lock UniqueLock construction.

Use the try_lock variable in constructor calls.

6.78 embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 >::Types< Index > Struct Template Reference

Type list used to derive input port types from Index.

```
#include <network.h>
```

Public Types

- typedef In< T_Index > Result

  Result of an input port type query.

6.78.1 Detailed Description

template<typename T1, typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil>
template<int Index>
struct embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 >::Types< Index >

Type list used to derive input port types from Index.

Template Parameters

Index	The index of the input port type to query.

6.78.2 Member Typedef Documentation

6.78.2.1 template<typename T1 , typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil> template<int Index> typedef In<T_Index> embb::dataflow::Network::Inputs< T1, T2, T3, T4, T5 >::Types< Index >::Result

Result of an input port type query.

T_Index is T1 if Index is 0, T2 if Index is 1 and so on.

6.79 embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 >::Types< Index > Struct Template Reference

Type list used to derive output port types from Index.

```
#include <network.h>
```

Public Types

- typedef Out< T_Index > Result

  Result of an output port type query.

6.79.1 Detailed Description

template<typename T1, typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil>
template<int Index>
struct embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 >::Types< Index >

Type list used to derive output port types from Index.

Template Parameters

Index	The index of the output port type to query.

6.79.2 Member Typedef Documentation

6.79.2.1 template<typename T1 , typename T2 = embb::base::internal::Nil, typename T3 = embb::base::internal::Nil, typename T4 = embb::base::internal::Nil, typename T5 = embb::base::internal::Nil> template<int Index> typedef Out<T_Index> embb::dataflow::Network::Outputs< T1, T2, T3, T4, T5 >::Types< Index >::Result

Result of an output port type query.

T_Index is T1 if Index is 0, T2 if Index is 1 and so on.

6.80 embb::base::UnderflowException Class Reference

Indicates a numeric underflow.

```
#include <exceptions.h>
```

Public Member Functions

- UnderflowException (const char *message)

  Constructs an exception with the specified message.
- virtual int Code () const

  Returns an integer code representing the exception.
- virtual const char * What () const throw ()

  Returns the error message.

6.80.1 Detailed Description

Indicates a numeric underflow.

6.80.2 Constructor & Destructor Documentation

6.80.2.1 embb::base::UnderflowException::UnderflowException (const char * *message*) [explicit]

Constructs an exception with the specified message.

Parameters

in	*message*	Error message

6.80.3 Member Function Documentation

6.80.3.1 virtual int embb::base::UnderflowException::Code ( ) const [virtual]

Returns an integer code representing the exception.

Returns

Exception code

Implements embb::base::Exception.

6.80.3.2 virtual const char* embb::base::Exception::What ( ) const throw) [virtual], [inherited]

Returns the error message.

Returns

Pointer to error message

6.81 embb::base::UniqueLock< Mutex > Class Template Reference

Flexible ownership wrapper for a mutex.

`#include <mutex.h>`

Public Member Functions

- UniqueLock ()
  Creates a lock without assigned mutex.
- UniqueLock (Mutex &mutex)
  Creates a lock from an unlocked mutex and locks it.
- UniqueLock (Mutex &mutex, DeferLockTag)
  Creates a lock from an unlocked mutex without locking it.
- UniqueLock (Mutex &mutex, TryLockTag)
  Creates a lock from an unlocked mutex and tries to lock it.
- UniqueLock (Mutex &mutex, AdoptLockTag)
  Creates a lock from an already locked mutex.
- ∼UniqueLock ()
  Unlocks the mutex if owned.
- void Lock ()
  Waits until the mutex is unlocked and locks it.
- bool TryLock ()
  Tries to lock the mutex and returns immediately.
- void Unlock ()
  Unlocks the mutex.
- void Swap (UniqueLock< Mutex > &other)
  Exchanges ownership of the wrapped mutex with another lock.
- Mutex ∗ Release ()
  Gives up ownership of the mutex and returns a pointer to it.
- bool OwnsLock () const
  Checks whether the mutex is owned and locked.

6.81.1 Detailed Description

template<typename Mutex = embb::base::Mutex>
class embb::base::UniqueLock< Mutex >

Flexible ownership wrapper for a mutex.

Provides exception controlled locking of a mutex with non-recursive semantics, that gives more flexibility than Lock↩Guard but also has slightly increased memory and processing overhead. Each instance of a UniqueLock can be used by one thread only!

Concurrency

Not thread-safe

See also

Mutex, LockGuard

Template Parameters

Mutex	Used mutex type. Has to fulfil the Mutex Concept.

6.81.2 Constructor & Destructor Documentation

6.81.2.1 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::UniqueLock ( )

Creates a lock without assigned mutex.

A mutex can be assigned to the lock using the method Swap().

6.81.2.2 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::UniqueLock (Mutex & *mutex*) [explicit]

Creates a lock from an unlocked mutex and locks it.

Precondition

`mutex` is unlocked

Parameters

in	*mutex*	Mutex to be managed.

6.81.2.3 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::UniqueLock (Mutex & *mutex*, DeferLockTag)

Creates a lock from an unlocked mutex without locking it.

Precondition

`mutex` is unlocked

Parameters

in	*mutex*	Mutex to be managed

6.81.2.4 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::UniqueLock (Mutex & *mutex*, TryLockTag)

Creates a lock from an unlocked mutex and tries to lock it.

Precondition

`mutex` is unlocked

Parameters

`in`	*mutex*	Mutex to be managed

6.81.2.5 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::UniqueLock (Mutex & *mutex*, AdoptLockTag)

Creates a lock from an already locked mutex.

Precondition

`mutex` is locked

Parameters

`in`	*mutex*	Mutex to be managed

6.81.2.6 template<typename Mutex = embb::base::Mutex> embb::base::UniqueLock< Mutex >::~UniqueLock ( )

Unlocks the mutex if owned.

6.81.3 Member Function Documentation

6.81.3.1 template<typename Mutex = embb::base::Mutex> void embb::base::UniqueLock< Mutex >::Lock ( )

Waits until the mutex is unlocked and locks it.

Exceptions

ErrorException,if	no mutex is set or it is locked

6.81.3.2 template<typename Mutex = embb::base::Mutex> bool embb::base::UniqueLock< Mutex >::TryLock ( )

Tries to lock the mutex and returns immediately.

Returns

`true` if the mutex could be locked, otherwise `false`.

Exceptions

ErrorException,if	no mutex is set

6.81.3.3 template<typename Mutex = embb::base::Mutex> void embb::base::UniqueLock< Mutex >::Unlock ( )

Unlocks the mutex.

Exceptions

ErrorException,if	no mutex is set or it is not locked

6.81.3.4 template<typename Mutex = embb::base::Mutex> void embb::base::UniqueLock< Mutex >::Swap (UniqueLock< Mutex > & *other*)

Exchanges ownership of the wrapped mutex with another lock.

Parameters

`in,out`	*other*	The lock to exchange ownership with

6.81.3.5 template<typename Mutex = embb::base::Mutex> Mutex* embb::base::UniqueLock< Mutex >::Release ( )

Gives up ownership of the mutex and returns a pointer to it.

Returns

A pointer to the owned mutex or NULL, if no mutex was owned

6.81.3.6 template<typename Mutex = embb::base::Mutex> bool embb::base::UniqueLock< Mutex >::OwnsLock ( ) const

Checks whether the mutex is owned and locked.

Returns

`true` if mutex is locked, otherwise `false`.

6.82 embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator > Class Template Reference

Wait-free value pool using array construction.

`#include <wait_free_array_value_pool.h>`

Classes

- class Iterator

  Forward iterator to iterate over the allocated elements of the pool.

Public Member Functions

- Iterator Begin ()

  Gets a forward iterator to the first allocated element in the pool.
- Iterator End ()

  Gets a forward iterator pointing after the last allocated element in the pool.
- template< typename ForwardIterator >
  WaitFreeArrayValuePool (ForwardIterator first, ForwardIterator last)

  Constructs a pool and fills it with the elements in the specified range.
- ∼WaitFreeArrayValuePool ()

  Destructs the pool.
- int Allocate (Type &element)

  Allocates an element from the pool.
- void Free (Type element, int index)

  Returns an element to the pool.

Static Public Member Functions

- static size_t GetMinimumElementCountForGuaranteedCapacity (size_t capacity)

  Due to concurrency effects, a pool might provide less elements than managed by it.

6.82.1 Detailed Description

template< typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic< Type > > >
class embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >

Wait-free value pool using array construction.

Implemented concepts:

Value Pool Concept

See also

LockFreeTreeValuePool

Template Parameters

Type	Element type (must support atomic operations such as `int`).
Undefined	Bottom element (cannot be stored in the pool)
Allocator	Allocator used to allocate the pool array

6.82.2 Constructor & Destructor Documentation

6.82.2.1 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> template<typename ForwardIterator > embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::WaitFreeArrayValuePool (ForwardIterator *first*, ForwardIterator *last*)

Constructs a pool and fills it with the elements in the specified range.

Dynamic memory allocation

Dynamically allocates `n*sizeof(embb::base::Atomic<Type>)` bytes, where `n = std::distance(first, last)` is the number of pool elements.

Concurrency

Not thread-safe

See also

Value Pool Concept

Parameters

in	*first*	Iterator pointing to the first element of the range the pool is filled with
in	*last*	Iterator pointing to the last plus one element of the range the pool is filled with

6.82.2.2 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::~WaitFreeArrayValuePool ( )

Destructs the pool.

Concurrency

Not thread-safe

6.82.3 Member Function Documentation

6.82.3.1 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> Iterator embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Begin ( )

Gets a forward iterator to the first allocated element in the pool.

Returns

a forward iterator pointing to the first allocated element.

Concurrency

Thread-safe and wait-free

6.82.3.2 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> Iterator embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::End ( )

Gets a forward iterator pointing after the last allocated element in the pool.

Returns

a forward iterator pointing after the last allocated element.

Concurrency

Thread-safe and wait-free

6.82.3.3 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> static size_t embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::GetMinimumElementCountForGuaranteedCapacity (size_t *capacity*) [static]

Due to concurrency effects, a pool might provide less elements than managed by it.

However, usually one wants to guarantee a minimal capacity. The count of elements that must be given to the pool when to guarantee `capacity` elements is computed using this function.

Returns

count of indices the pool has to be initialized with

Parameters

in	*capacity*	count of indices that shall be guaranteed

6.82.3.4 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> int embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Allocate (Type & *element*)

Allocates an element from the pool.

Returns

Index of the element if the pool is not empty, otherwise `-1`.

Concurrency

Thread-safe and wait-free

See also

Value Pool Concept

Parameters

`in,out`	*element*	Reference to the allocated element. Unchanged, if the operation was not successful.

6.82.3.5 template<typename Type, Type Undefined, class Allocator = embb::base::Allocator< embb::base::Atomic<Type> >> void embb::containers::WaitFreeArrayValuePool< Type, Undefined, Allocator >::Free (Type *element*, int *index*)

Returns an element to the pool.

Note

The element must have been allocated with Allocate().

Concurrency

Thread-safe and wait-free

See also

Value Pool Concept

Parameters

`in`	*element*	Element to be returned to the pool
`in`	*index*	Index of the element as obtained by Allocate()

6.83 embb::containers::WaitFreeSPSCQueue< Type, Allocator > Class Template Reference

Wait-free queue for a single producer and a single consumer.

```
#include <wait_free_spsc_queue.h>
```

Public Member Functions

- WaitFreeSPSCQueue (size_t capacity)

  Creates a queue with at least the specified capacity.
- ∼WaitFreeSPSCQueue ()

  Destroys the queue.
- size_t GetCapacity ()

  Returns the capacity of the queue.
- bool TryEnqueue (Type const &element)

  Tries to enqueue an element into the queue.
- bool TryDequeue (Type &element)

  Tries to dequeue an element from the queue.

6.83.1 Detailed Description

**template< typename Type, class Allocator = embb::base::Allocator< Type >>
class embb::containers::WaitFreeSPSCQueue< Type, Allocator >**

Wait-free queue for a single producer and a single consumer.

Implemented concepts:

Queue Concept

See also

LockFreeMPMCQueue

Template Parameters

Type	Type of the queue elements
Allocator	Allocator type for allocating queue elements.

6.83.2 Constructor & Destructor Documentation

6.83.2.1 template< typename Type , class Allocator = embb::base::Allocator< Type >> embb::containers::WaitFreeSPSCQueue< Type, Allocator >::WaitFreeSPSCQueue (size_t *capacity*)

Creates a queue with at least the specified capacity.

Dynamic memory allocation

Allocates `2^k` elements of type `Type`, where `k` is the smallest number such that `capacity <= 2^k` holds.

Concurrency

Not thread-safe

See also

Queue Concept

Parameters

in	*capacity*	Capacity of the queue

6.83.2.2 template<typename Type , class Allocator = embb::base::Allocator< Type >> embb::containers::WaitFreeSPSCQueue< Type, Allocator >::~WaitFreeSPSCQueue ( )

Destroys the queue.

Concurrency

Not thread-safe

6.83.3 Member Function Documentation

6.83.3.1 template<typename Type , class Allocator = embb::base::Allocator< Type >> size_t embb::containers::WaitFreeSPSCQueue< Type, Allocator >::GetCapacity ( )

Returns the capacity of the queue.

Returns

Number of elements the queue can hold.

Concurrency

Thread-safe and wait-free

6.83.3.2 template<typename Type , class Allocator = embb::base::Allocator< Type >> bool embb::containers::WaitFreeSPSCQueue< Type, Allocator >::TryEnqueue (Type const & *element*)

Tries to enqueue an element into the queue.

Returns

`true` if the element could be enqueued, `false` if the queue is full.

Concurrency

Thread-safe and wait-free

Note

Concurrently enqueueing elements by multiple producers leads to undefined behavior.

See also

Queue Concept

Parameters

in	*element*	Const reference to the element that shall be enqueued

6.83.3.3 template<typename Type , class Allocator = embb::base::Allocator< Type >> bool embb::containers::WaitFreeSPSCQueue< Type, Allocator >::TryDequeue (Type & *element*)

Tries to dequeue an element from the queue.

Returns

`true` if an element could be dequeued, `false` if the queue is empty.

Concurrency

Thread-safe and wait-free

Note

Concurrently dequeueing elements by multiple consumers leads to undefined behavior.

See also

Queue Concept

Parameters

`in,out`	*element*	Reference to the dequeued element. Unchanged, if the operation was not successful.

6.84 embb::algorithms::ZipIterator< IteratorA, IteratorB > Class Template Reference

Zip container for two iterators.

```
#include <zip_iterator.h>
```

Public Types

Iterator Typedefs

Necessary typedefs for iterators (std compatibility).

- typedef std::random_access_iterator_tag **iterator_category**
- typedef std::iterator_traits< IteratorA >::difference_type **difference_type**
- typedef std::iterator_traits< IteratorA >::reference **RefA**
- typedef std::iterator_traits< IteratorB >::reference **RefB**
- typedef std::iterator_traits< IteratorA >::value_type **ValueA**
- typedef std::iterator_traits< IteratorB >::value_type **ValueB**
- typedef ZipPair< ValueA, ValueB > **value_type**
- typedef ZipPair< RefA, RefB > **reference**
- typedef ZipPair< RefA, RefB > **pointer**

Public Member Functions

- ZipIterator (IteratorA iter_a, IteratorB iter_b)

  Constructs a zip iterator from two iterators of any type.
- bool operator== (const ZipIterator &other) const

  Compares two zip iterators for equality.
- bool operator!= (const ZipIterator &other) const

  Compares two zip iterators for inequality.
- void operator++ ()

  Applies prefix increment on both iterators.
- void operator-- ()

  Applies prefix decrement on both iterators.
- ZipIterator< IteratorA, IteratorB > operator+ (difference_type distance) const

  Returns an instance of a zip iterator where both iterators have been advanced by the specified distance.
- ZipIterator< IteratorA, IteratorB > operator- (difference_type distance) const

  Returns an instance of a zip iterator where both iterators have been regressed by the specified distance.
- ZipIterator< IteratorA, IteratorB > & operator+= (difference_type distance)

  Advances both iterators by the specified distance.
- ZipIterator< IteratorA, IteratorB > & operator-= (difference_type distance)

  Regresses both iterators by the specified distance.
- difference_type operator- (const ZipIterator< IteratorA, IteratorB > &other) const

  Computes the distance between two zip iterators.
- reference operator∗ () const

  Dereferences the zip iterator.

6.84.1 Detailed Description

template<typename IteratorA, typename IteratorB>
class embb::algorithms::ZipIterator< IteratorA, IteratorB >

Zip container for two iterators.

This container allows zipping two iterators together, thus enabling them to be used in situations where only one iterator can be used. Any operation applied to the zip iterator will subsequently be applied to the contained iterators. Dereferencing the iterator will return a ZipPair containing the values referenced by the iterators.

Concurrency

Not thread-safe

Note

It is required that the two iterators have the same `difference_type` or that at least the first iterator's `difference_type` can be implicitly converted to the second iterator's `difference_type`. Moreover, when calculating the distance between two ZipIterators, the distances between both pairs of iterators are equal.

See also

ZipPair

Template Parameters

IteratorA	First iterator
IteratorB	Second iterator

6.84.2 Constructor & Destructor Documentation

6.84.2.1 template<typename IteratorA, typename IteratorB> embb::algorithms::ZipIterator< IteratorA, IteratorB >::ZipIterator (IteratorA *iter_a*, IteratorB *iter_b*)

Constructs a zip iterator from two iterators of any type.

Parameters

`in`	*iter_a*	First iterator
`in`	*iter_b*	Second iterator

6.84.3 Member Function Documentation

6.84.3.1 template<typename IteratorA, typename IteratorB> bool embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator== (const ZipIterator< IteratorA, IteratorB > & *other*) const

Compares two zip iterators for equality.

Returns

`true` if zip iterators are equal, otherwise `false`

Parameters

`in`	*other*	Reference to right-hand side of equality operator

6.84.3.2 template<typename IteratorA, typename IteratorB> bool embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator!= (const ZipIterator< IteratorA, IteratorB > & *other*) const

Compares two zip iterators for inequality.

Returns

`true` if any iterator doesn't equal the other, otherwise `false`

Parameters

in	*other*	Reference to right-hand side of inequality operator

6.84.3.3 template<typename IteratorA, typename IteratorB> void embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator++ ( )

Applies prefix increment on both iterators.

6.84.3.4 template<typename IteratorA, typename IteratorB> void embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator-- ( )

Applies prefix decrement on both iterators.

6.84.3.5 template<typename IteratorA, typename IteratorB> ZipIterator<IteratorA, IteratorB> embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator+ (difference_type *distance*) const

Returns an instance of a zip iterator where both iterators have been advanced by the specified distance.

Returns

New zip iterator containing the advanced iterators

Parameters

in	*distance*	Number of elements to advance the iterators

6.84.3.6 template<typename IteratorA, typename IteratorB> ZipIterator<IteratorA, IteratorB> embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator- (difference_type *distance*) const

Returns an instance of a zip iterator where both iterators have been regressed by the specified distance.

Returns

New zip iterator containing the regressed iterators

Parameters

in	*distance*	Number of elements to regress the iterators

6.84.3.7 template<typename IteratorA, typename IteratorB> ZipIterator<IteratorA, IteratorB>& embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator+= (difference_type *distance*)

Advances both iterators by the specified distance.

Returns

Reference to *this

Parameters

in	*distance*	Number of elements to advance the iterators

6.84.3.8 template<typename IteratorA, typename IteratorB> ZipIterator<IteratorA, IteratorB>& embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator-= (difference_type *distance*)

Regresses both iterators by the specified distance.

Returns

Reference to *this

Parameters

in	*distance*	Number of elements to regress the iterators

6.84.3.9 template<typename IteratorA, typename IteratorB> difference_type embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator- (const ZipIterator< IteratorA, IteratorB > & *other*) const

Computes the distance between two zip iterators.

It is assumed that both iterator pairs have the same distance.

Returns

The distance between the zip iterators

Parameters

in	*other*	Reference to right-hand side of subtraction operator

6.84.3.10 template<typename IteratorA, typename IteratorB> reference embb::algorithms::ZipIterator< IteratorA, IteratorB >::operator* ( ) const

Dereferences the zip iterator.

Returns

ZipPair containing the dereferenced values.

6.85 embb::algorithms::ZipPair< TypeA, TypeB > Class Template Reference

Container for the values of two dereferenced iterators.

`#include <zip_iterator.h>`

Public Member Functions

- ZipPair (TypeA first, TypeB second)
  Constructs a pair from two values.
- ZipPair (const ZipPair &other)
  Copies a pair.
- TypeA First ()
  Returns the first value of the pair.
- TypeB Second ()
  Returns the second value of the pair.
- const TypeA First () const
  Returns the first value of the pair.
- const TypeB Second () const
  Returns the second value of the pair.

6.85.1 Detailed Description

template<typename TypeA, typename TypeB>
class embb::algorithms::ZipPair< TypeA, TypeB >

Container for the values of two dereferenced iterators.

The values contained are of type `std::iterator_traits<Iterator>::reference`.

Concurrency

Not thread-safe

Template Parameters

TypeA	Type of the first value
TypeB	Type of the first value

6.85.2 Constructor & Destructor Documentation

6.85.2.1 template<typename TypeA, typename TypeB> embb::algorithms::ZipPair< TypeA, TypeB >::ZipPair (TypeA *first*, TypeB *second*)

Constructs a pair from two values.

Parameters

in	*first*	First value
in	*second*	Second value

6.85.2.2 template< typename TypeA, typename TypeB > embb::algorithms::ZipPair< TypeA, TypeB >::ZipPair (const ZipPair< TypeA, TypeB > & *other*)

Copies a pair.

Parameters

in	*other*	pair to copy

6.85.3 Member Function Documentation

6.85.3.1 template< typename TypeA, typename TypeB > TypeA embb::algorithms::ZipPair< TypeA, TypeB >::First ( )

Returns the first value of the pair.

Returns

The first value of the pair.

6.85.3.2 template< typename TypeA, typename TypeB > TypeB embb::algorithms::ZipPair< TypeA, TypeB >::Second ( )

Returns the second value of the pair.

Returns

The second value of the pair

6.85.3.3 template< typename TypeA, typename TypeB > const TypeA embb::algorithms::ZipPair< TypeA, TypeB >::First ( ) const

Returns the first value of the pair.

Returns

The first value of the pair.

6.85.3.4 template< typename TypeA, typename TypeB > const TypeB embb::algorithms::ZipPair< TypeA, TypeB >::Second ( ) const

Returns the second value of the pair.

Returns

The second value of the pair