

RTSHA

Generated by Doxygen 1.9.7

---

<b>1 Real Time Safety Heap Allocator</b>	<b>1</b>
<b>2 Module Documentation</b>	<b>2</b>
2.1 RTSHA Error Codes . . . . .	2
2.1.1 Detailed Description . . . . .	2
<b>3 Class Documentation</b>	<b>3</b>
3.1 rtsha::BigMemoryPage Class Reference . . . . .	3
3.1.1 Detailed Description . . . . .	5
3.1.2 Constructor & Destructor Documentation . . . . .	5
3.1.3 Member Function Documentation . . . . .	5
3.2 internal::FreeLinkedList Class Reference . . . . .	8
3.2.1 Detailed Description . . . . .	8
3.2.2 Constructor & Destructor Documentation . . . . .	8
3.2.3 Member Function Documentation . . . . .	9
3.3 internal::FreeList Class Reference . . . . .	10
3.3.1 Detailed Description . . . . .	10
3.3.2 Constructor & Destructor Documentation . . . . .	10
3.3.3 Member Function Documentation . . . . .	11
3.4 internal::FreeListArray Class Reference . . . . .	11
3.4.1 Detailed Description . . . . .	12
3.4.2 Constructor & Destructor Documentation . . . . .	12
3.4.3 Member Function Documentation . . . . .	12
3.5 internal::FreeMap Class Reference . . . . .	13
3.5.1 Detailed Description . . . . .	14
3.5.2 Constructor & Destructor Documentation . . . . .	14
3.5.3 Member Function Documentation . . . . .	14
3.6 rtsha::Heap Class Reference . . . . .	16
3.6.1 Detailed Description . . . . .	17
3.6.2 Member Function Documentation . . . . .	17
3.7 HeapCallbacksStruct Struct Reference . . . . .	22
3.7.1 Detailed Description . . . . .	23
3.8 internal::HeapInternal Class Reference . . . . .	23
3.8.1 Detailed Description . . . . .	24
3.8.2 Member Function Documentation . . . . .	24
3.8.3 Member Data Documentation . . . . .	27
3.9 internal::InternListAllocator< T > Class Template Reference . . . . .	28
3.9.1 Detailed Description . . . . .	29
3.9.2 Constructor & Destructor Documentation . . . . .	29
3.9.3 Member Function Documentation . . . . .	30
3.10 internal::InternMapAllocator< T > Struct Template Reference . . . . .	30
3.10.1 Detailed Description . . . . .	31
3.10.2 Constructor & Destructor Documentation . . . . .	31

---

3.10.3 Member Function Documentation . . . . .	32
3.11 rtsha::MemoryBlock Class Reference . . . . .	33
3.11.1 Detailed Description . . . . .	34
3.11.2 Constructor & Destructor Documentation . . . . .	34
3.11.3 Member Function Documentation . . . . .	35
3.12 rtsha::MemoryPage Class Reference . . . . .	38
3.12.1 Detailed Description . . . . .	40
3.12.2 Constructor & Destructor Documentation . . . . .	40
3.12.3 Member Function Documentation . . . . .	40
3.13 rtsha::PowerTwoMemoryPage Class Reference . . . . .	46
3.13.1 Detailed Description . . . . .	48
3.13.2 Constructor & Destructor Documentation . . . . .	49
3.13.3 Member Function Documentation . . . . .	49
3.14 internal::PREALLOC_MEMORY< T, n > Struct Template Reference . . . . .	50
3.14.1 Detailed Description . . . . .	50
3.14.2 Constructor & Destructor Documentation . . . . .	51
3.14.3 Member Function Documentation . . . . .	51
3.15 rtsha::rtsha_block Struct Reference . . . . .	51
3.15.1 Detailed Description . . . . .	52
3.15.2 Member Data Documentation . . . . .	52
3.16 rtsha::rtsha_page Struct Reference . . . . .	52
3.16.1 Detailed Description . . . . .	53
3.17 rtsha::RTSHAllocator< T > Struct Template Reference . . . . .	53
3.17.1 Detailed Description . . . . .	54
3.17.2 Constructor & Destructor Documentation . . . . .	54
3.17.3 Member Function Documentation . . . . .	55
3.18 rtsha::SmallFixMemoryPage Class Reference . . . . .	56
3.18.1 Detailed Description . . . . .	56
3.18.2 Constructor & Destructor Documentation . . . . .	56
3.18.3 Member Function Documentation . . . . .	57
<b>4 File Documentation</b> . . . . .	<b>57</b>
4.1 allocator.h . . . . .	57
4.2 arm_spec_functions.h . . . . .	58
4.3 BigMemoryPage.h . . . . .	59
4.4 errors.h . . . . .	59
4.5 FastPlusAllocator.h . . . . .	59
4.6 FreeLinkedList.h . . . . .	60
4.7 FreeList.h . . . . .	62
4.8 FreeListArray.h . . . . .	62
4.9 FreeMap.h . . . . .	64
4.10 Heap.h . . . . .	65

4.11 HeapCallbacks.h . . . . .	66
4.12 internal.h . . . . .	67
4.13 InternListAllocator.h . . . . .	69
4.14 InternMapAllocator.h . . . . .	70
4.15 MemoryBlock.h . . . . .	71
4.16 MemoryPage.h . . . . .	73
4.17 PowerTwoMemoryPage.h . . . . .	76
4.18 SmallFixMemoryPage.h . . . . .	77
<b>Index</b>	<b>79</b>

# 1 Real Time Safety Heap Allocator

## Author

Boris Radonic

Here is documentation of RTSHA.

## RTSHA Algorithms

There are several different algorithms that can be used for heap allocation supported by RTSHA:

1. Small Fix Memory Pages This algorithm is an approach to memory management that is often used in specific situations where objects of a certain size are frequently allocated and deallocated. By using of uses 'Fixed chunk size' algorithm greatly simplifies the memory allocation process and reduce fragmentation. The memory is divided into pages of chunks(blocks) of a fixed size (32, 64, 128, 256 and 512 bytes). When an allocation request comes in, it can simply be given one of these blocks. This means that the allocator doesn't have to search through the heap to find a block of the right size, which can improve performance. The free blocks memory is used as 'free list' storage. Deallocations are also straightforward, as the block is added back to the list of available chunks. There's no need to merge adjacent free blocks, as there is with some other allocation strategies, which can also improve performance. However, fixed chunk size allocation is not a good fit for all scenarios. It works best when the majority of allocations are of the same size, or a small number of different sizes. If allocations requests are of widely varying sizes, then this approach can lead to a lot of wasted memory, as small allocations take up an entire chunk, and large allocations require multiple chunks. Small Fix Memory Page is also used internally by "Power Two Memory Page" and "Big Memory Page" algorithms.
2. Power Two Memory Pages This is a more complex system, which only allows blocks of sizes that are powers of two. This makes merging free blocks back together easier and reduces fragmentation. A specialised search algorithm is used for fast storage and retrieval of ordered information which are stored in space of 'freed blocks'. This is a fairly efficient method of allocating memory, particularly useful for systems where memory fragmentation is an important concern. The algorithm divides memory into partitions to try to minimize fragmentation and the 'Best Fit' algorithm searches the page to find the smallest block that is large enough to satisfy the allocation. Furthermore, this system is resistant to breakdowns due to its algorithmic approach to allocating and deallocating memory. The coalescing operation helps ensure that large contiguous blocks of memory can be reformed after they are freed, reducing the likelihood of fragmentation over time. Coalescing relies on having free blocks of the same size available, which is not always the case, and so this system does not completely eliminate fragmentation but rather aims to minimize it.
3. Big Memory Pages

Note: This algorithm is primarily designed for test purposes, especially for systems with constrained memory. When compared to the "Small Fixed Memory Pages" and "Power Two Memory Pages" algorithms, this approach may exhibit relatively slower (inperformant) behaviors. The "Big Memory Pages" algorithm employs the "Best Fit" strategy, which is complemented by a "Red-Black" balanced tree. The Red-Black tree ensures worst-case guarantees for insertion, deletion, and search times, making it predictable in performance. A key distinction between this system and the "Power Two Memory Page" is in how they handle memory blocks. Unlike the latter, "Big Memory Pages" does not restrict memory to be partitioned exclusively into power-of-two sizes. Instead, variable block sizes are allowed, providing more flexibility. Additionally, once memory blocks greater than 512 bytes are released, they are promptly merged or coalesced, optimizing the memory usage. Despite its features, it's essential to understand the specific use-cases and limitations of this algorithm and to choose the most suitable one based on the system's requirements and constraints.

The use of 'Small Fixed Memory Pages' in combination with 'Power Two Memory Pages' is recommended for all real time systems.

## 2 Module Documentation

### 2.1 RTSHA Error Codes

#### Macros

- `#define RTSHA_OK (0U)`  
*Represents a successful operation or status.*
- `#define RTSHA_ErrorInit (16U)`  
*Error code indicating an initialization error.*
- `#define RTSHA_ErrorInitPageSize (32U)`  
*Error code indicating an invalid page size during initialization.*
- `#define RTSHA_ErrorInitOutOfHeap (33U)`  
*Error code indicating an out-of-heap error during initialization.*
- `#define RTSHA_OutOfMemory (64U)`  
*Error code indicating the system has run out of memory.*
- `#define RTSHA_NoPages (128U)`  
*Error code indicating no pages are available.*
- `#define RTSHA_NoPage (129U)`  
*Error code indicating a specific page is not available.*
- `#define RTSHA_NoFreePage (130U)`  
*Error code indicating there is no free page available.*
- `#define RTSHA_InvalidBlock (256U)`  
*Error code indicating the memory block is invalid.*
- `#define RTSHA_InvalidBlockDistance (257U)`  
*Error code indicating an invalid distance between blocks.*
- `#define RTSHA_InvalidNumberOfFreeBlocks (258U)`  
*Error code indicating an invalid number of free blocks.*

#### 2.1.1 Detailed Description

These are the error codes used throughout the RTSHA system.

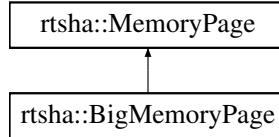
## 3 Class Documentation

### 3.1 rtsha::BigMemoryPage Class Reference

This class provides various memory handling functions that manipulate [MemoryBlock](#)'s on 'big memory page'.

```
#include <BigMemoryPage.h>
```

Inheritance diagram for rtsha::BigMemoryPage:



#### Public Member Functions

- **BigMemoryPage ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **BigMemoryPage (rtsha\_page \*page)**  
*Constructor that initializes the [BigMemoryPage](#) with a given page.*
- **virtual ~BigMemoryPage ()**  
*Virtual destructor for the [BigMemoryPage](#).*
- **virtual void \* allocate\_block (const size\_t &size) final**  
*Allocates a block of memory of the specified size.*
- **virtual void free\_block (MemoryBlock &block) final**  
*Frees the specified memory block.*
- **void createInitialFreeBlocks ()**  
*Creates the initial first two free blocks within the page.*

#### Public Member Functions inherited from rtsha::MemoryPage

- **MemoryPage ()=delete**  
*Default constructor is deleted.*
- **MemoryPage (rtsha\_page \*page) noexcept**  
*Constructor that initializes the [MemoryPage](#) with a given page.*
- **virtual ~MemoryPage ()**  
*Virtual destructor.*
- **bool checkBlock (size\_t address)**  
*Check if a block exists at the given address and if the block is valid.*
- **virtual void \* allocate\_block (const size\_t &size)=0**  
*Pure virtual function to allocate a block of memory.*
- **virtual void free\_block (MemoryBlock &block)=0**  
*Pure virtual function to free a block of memory.*
- **void \* allocate\_block\_at\_current\_pos (const size\_t &size)**  
*Allocates a block of memory at the current position.*
- **void incFreeBlocks ()**  
*Increments the count of free blocks.*

## Protected Member Functions

- void [splitBlock](#) (MemoryBlock &block, size\_t size)  
*Splits a memory block based on the specified size.*
- void [mergeLeft](#) (MemoryBlock &block)  
*Merges the specified block with its left neighbor.*
- void [mergeRight](#) (MemoryBlock &block)  
*Merges the specified block with its right neighbor.*

## Protected Member Functions inherited from [rtsha::MemoryPage](#)

- void [lock](#) ()  
*Locks the page for thread-safe operations.*
- void [unlock](#) ()  
*Unlocks the page after thread-safe operations.*
- void [reportError](#) (uint32\_t error)  
*Reports an error using the specified error callback.*
- void [setFreeBlockAllocatorsAddress](#) (const size\_t &address)  
*Sets the address of the last free block temporary. The address will be used by InternListAllocator as storage for the elements of the 'std::forward\_list'.*
- rtsha\_page\_size\_type [getPageType](#) () const  
*Retrieves the type of the page.*
- void \* [getFreeList](#) () const  
*Gets the free list pointer of the page.*
- void \* [getFreeListArray](#) () const  
*Gets the free list array pointer of the page.*
- void \* [getFreeMap](#) () const  
*Gets the free map pointer of the page.*
- size\_t [getFreeBlocks](#) () const  
*Retrieves the number of free blocks in the page.*
- size\_t [getMinBlockSize](#) () const  
*Retrieves the number of free blocks in the page.*
- address\_t [getPosition](#) () const  
*Gets the current page position.*
- void [setPosition](#) (address\_t pos)  
*Sets the current page position.*
- void [incPosition](#) (const size\_t &val)  
*Increments the page position by the given value.*
- void [decPosition](#) (const size\_t &val)  
*Decreases the page position by the given value.*
- void [decFreeBlocks](#) ()  
*Decreases the number of free blocks in the page.*
- address\_t [getEndPosition](#) () const  
*Gets the end position of the page.*
- address\_t [getStartPosition](#) () const  
*Gets the start position of the page.*
- bool [fitOnPage](#) (const size\_t &size) const  
*Checks if a block of the specified size fits on the page.*
- bool [hasLastBlock](#) () const  
*Checks if the page 'last block' has been set.*
- bool [isLastPageBlock](#) (MemoryBlock &block) const

- Determines if the provided block is the last block of the page.*
- `rtsha_block * getLastBlock () const`  
*Gets the last block of the page.*
  - `void setLastBlock (const MemoryBlock &block)`  
*Sets the last block of the page.*

## Additional Inherited Members

### Protected Attributes inherited from `rtsha::MemoryPage`

- `rtsha_page * _page`  
*Pointer to the page structure in memory.*

### 3.1.1 Detailed Description

This class provides various memory handling functions that manipulate `MemoryBlock`'s on 'big memory page'.

Similar to the 'Power Two Memory Page', this algorithm employs the 'Best Fit' algorithm, in conjunction with a 'Red-Black' balanced tree, which offers worst-case guarantees for insertion, deletion, and search times. It promptly merges or coalesces memory blocks larger than 'MIN\_BLOCK\_SIZE\_FOR\_SPLIT' bytes after they are released.

### 3.1.2 Constructor & Destructor Documentation

#### `BigMemoryPage()`

```
rtsha::BigMemoryPage::BigMemoryPage (
    rtsha_page * page ) [inline], [explicit]
```

Constructor that initializes the `BigMemoryPage` with a given page.

##### Parameters

<code>page</code>	The <code>rtsha_page</code> structure to initialize the <code>BigMemoryPage</code> with.
-------------------	--

### 3.1.3 Member Function Documentation

#### `allocate_block()`

```
rtsha::BigMemoryPage::allocate_block (
    const size_t & size ) [final], [virtual]
```

Allocates a block of memory of the specified size.

##### Parameters

<code>size</code>	The size of the memory block to allocate.
-------------------	---

**Returns**

On success, a pointer to the memory block allocated by the function.

Implements [rtsha::MemoryPage](#).

**free\_block()**

```
virtual void rtsha::BigMemoryPage::free_block (
    MemoryBlock & block ) [final], [virtual]
```

Frees the specified memory block.

**Parameters**

<i>block</i>	The memory block to be freed.
--------------	-------------------------------

Implements [rtsha::MemoryPage](#).

**mergeLeft()**

```
void rtsha::BigMemoryPage::mergeLeft (
    MemoryBlock & block ) [protected]
```

Merges the specified block with its left neighbor.

**Parameters**

<i>block</i>	The block to be merged with its left neighbor.
--------------	--

**mergeRight()**

```
void rtsha::BigMemoryPage::mergeRight (
    MemoryBlock & block ) [protected]
```

Merges the specified block with its right neighbor.

**Parameters**

<i>block</i>	The block to be merged with its right neighbor.
--------------	---

**splitBlock()**

```
void rtsha::BigMemoryPage::splitBlock (
    MemoryBlock & block,
    size_t size ) [protected]
```

Splits a memory block based on the specified size.

## Parameters

<i>block</i>	The block to be split.
<i>size</i>	The desired size of the block after splitting.

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

- BigMemoryPage.h

## 3.2 internal::FreeLinkedList Class Reference

Implements a doubly linked list for managing free blocks of memory.

```
#include <FreeLinkedList.h>
```

### Public Member Functions

- **FreeLinkedList (rtsha\_page \*page)**  
*Constructs a FreeLinkedList with the given memory page.*
- **~FreeLinkedList ()**  
*Destructor for the FreeLinkedList.*
- **rtsha\_attr\_inline void push (const size\_t &data)**  
*Adds a new node with the given memory block address to the head of the list.*
- **rtsha\_attr\_inline bool is\_empty () const**  
*Checks if the list is empty.*
- **rtsha\_attr\_inline size\_t pop ()**  
*Removes and retrieves a memory block address from the head of the list.*
- **rtsha\_attr\_inline bool delete\_address (const size\_t &address, void \*block)**  
*Deletes a node with the specified memory block address from the list.*

### 3.2.1 Detailed Description

Implements a doubly linked list for managing free blocks of memory.

This class provides basic operations like push, pop, and delete for managing free blocks of memory in a specific memory page. The nodes of the list are aligned to the size of size\_t for performance and memory layout reasons.

### 3.2.2 Constructor & Destructor Documentation

#### FreeLinkedList()

```
internal::FreeLinkedList::FreeLinkedList (   
    rtsha_page * page ) [inline], [explicit]
```

Constructs a FreeLinkedList with the given memory page.

**Parameters**

<i>page</i>	The rtsha_page that this FreeLinkedList should manage.
-------------	--

**3.2.3 Member Function Documentation****delete\_address()**

```
rtsha_attr_inline bool internal::FreeLinkedList::delete_address (
    const size_t & address,
    void * block ) [inline]
```

Deletes a node with the specified memory block address from the list.

**Parameters**

<i>address</i>	Memory block address of the node to be deleted.
<i>block</i>	Pointer to the memory block associated with the address.

**Returns**

Returns true if the node was found and removed, otherwise false.

**is\_empty()**

```
rtsha_attr_inline bool internal::FreeLinkedList::is_empty ( ) const [inline]
```

Checks if the list is empty.

**Returns**

Returns true if the list is empty, otherwise false.

**pop()**

```
rtsha_attr_inline size_t internal::FreeLinkedList::pop ( ) [inline]
```

Removes and retrieves a memory block address from the head of the list.

**Returns**

The memory block address retrieved from the list.

**push()**

```
rtsha_attr_inline void internal::FreeLinkedList::push (
    const size_t & data ) [inline]
```

Adds a new node with the given memory block address to the head of the list.

**Parameters**

<code>data</code>	The memory block address to be added.
-------------------	---------------------------------------

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

- `FreeLinkedList.h`

### 3.3 internal::FreeList Class Reference

A memory-efficient free list implementation aligned to the size of `size_t`.

```
#include <FreeList.h>
```

#### Public Member Functions

- **FreeList ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **FreeList (rtsha\_page \*page)**  
*Constructs a `FreeList` with the given memory page.*
- **~FreeList ()**  
*Destructor for the `FreeList`.*
- **rtsha\_attr\_inline void push (const size\_t &address)**  
*Pushes a memory address onto the free list.*
- **rtsha\_attr\_inline size\_t pop ()**  
*Pops and retrieves a memory address from the free list.*
- **rtsha\_attr\_inline bool delete\_address (const size\_t &address, void \*block)**

#### 3.3.1 Detailed Description

A memory-efficient free list implementation aligned to the size of `size_t`.

This class is designed to manage and recycle the list of free blocks in an efficient manner. Internally, it utilizes a specialized list structure and a custom allocator to manage blocks of memory. Allocator uses the space in unused space of already free blocks.

#### 3.3.2 Constructor & Destructor Documentation

##### **FreeList()**

```
internal::FreeList::FreeList (
    rtsha_page * page ) [explicit]
```

Constructs a `FreeList` with the given memory page.

**Parameters**

<code>page</code>	The rtsha_page that this <a href="#">FreeList</a> should manage.
-------------------	--

### 3.3.3 Member Function Documentation

**pop()**

```
rtsha_attr_inline size_t internal::FreeList::pop ( ) [inline]
```

Pops and retrieves a memory address from the free list.

**Returns**

The memory address retrieved from the free list.

**push()**

```
rtsha_attr_inline void internal::FreeList::push ( const size_t & address ) [inline]
```

Pushes a memory address onto the free list.

**Parameters**

<code>address</code>	The memory address to be added to the free list.
----------------------	--

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

- `FreeList.h`

## 3.4 internal::FreeListArray Class Reference

Manages memory allocation of free blocks using a free list.

```
#include <FreeListArray.h>
```

**Public Member Functions**

- **FreeListArray ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **FreeListArray (rtsha\_page \*page, size\_t min\_block\_size, size\_t page\_size)**  
*Constructs a [FreeListArray](#) with the given memory page.*
- **~FreeListArray ()**  
*Destructor for the [FreeListArray](#).*

- `rtsha_attr_inline void push (const size_t data, size_t size)`  
*Pushes a memory address onto the appropriate free list.*
- `rtsha_attr_inline size_t pop (size_t size)`  
*Pops and retrieves a memory address from the first appropriate free list.*
- `rtsha_attr_inline bool delete_address (const size_t &address, void *block, const size_t &size)`  
*Attempts to delete a memory address from the free lists.*

### 3.4.1 Detailed Description

Manages memory allocation of free blocks using a free list.

This class is used for efficient memory management by maintaining a list of free memory blocks. The memory blocks are organized in an array based on their sizes, allowing for fast allocation and deallocation operations.

### 3.4.2 Constructor & Destructor Documentation

#### `FreeListArray()`

```
internal::FreeListArray::FreeListArray (
    rtsha_page * page,
    size_t min_block_size,
    size_t page_size ) [explicit]
```

Constructs a `FreeListArray` with the given memory page.

##### Parameters

<code>page</code>	The <code>rtsha_page</code> that this <code>FreeListArray</code> should manage.
<code>min_block_size</code>	Minimum block size this free list array can manage.
<code>page_size</code>	Size of the page to be managed.

### 3.4.3 Member Function Documentation

#### `delete_address()`

```
rtsha_attr_inline bool internal::FreeListArray::delete_address (
    const size_t & address,
    void * block,
    const size_t & size ) [inline]
```

Attempts to delete a memory address from the free lists.

This method searches for a given memory address within the range of managed memory pages. If found, the address is removed from the free list.

##### Parameters

<code>address</code>	The memory address to be deleted.
<code>block</code>	Pointer to the memory block associated with the address.
<code>size</code>	Size of the memory block.

**Returns**

Returns true if the address was found and removed, otherwise false.

**pop()**

```
rtsha_attr_inline size_t internal::FreeListArray::pop (
    size_t size ) [inline]
```

Pops and retrieves a memory address from the first appropriate free list.

**Returns**

The memory address retrieved from the free list.

**push()**

```
rtsha_attr_inline void internal::FreeListArray::push (
    const size_t data,
    size_t size ) [inline]
```

Pushes a memory address onto the appropriate free list.

**Parameters**

<code>address</code>	The memory address to be added to the free list.
----------------------	--

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

- `FreeListArray.h`

## 3.5 internal::FreeMap Class Reference

A memory-efficient multimap implementation aligned to the size of `size_t`.

```
#include <FreeMap.h>
```

### Public Member Functions

- **FreeMap ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **FreeMap (rtsha\_page \*page)**  
*Constructs a `FreeMap` with the provided memory page.*
- **~FreeMap ()**  
*Destructor for the `FreeMap`.*
- **rtsha\_attr\_inline void insert (const uint64\_t key, size\_t block)**  
*Inserts a key-value pair into the multimap.*

- `rtsha_attr_inline bool del (const uint64_t key, size_t block)`  
*Deletes a key-value pair from the map based on the given key.*
- `rtsha_attr_inline size_t find (const uint64_t key)`  
*Finds the value associated with a given key.*
- `rtsha_attr_inline bool exists (const uint64_t key, size_t block)`  
*Checks if a given key-value pair exists in the map.*
- `size_t size () const`  
*Retrieves the number of key-value pairs in the map.*

### 3.5.1 Detailed Description

A memory-efficient multimap implementation aligned to the size of `size_t`.

The `FreeMap` class is designed to manage key-value pairs in memory. It offers functionalities like insertion, deletion, and lookup of key-value pairs. Internally, it employs a custom allocator and map structure to handle the memory. Custom allocator uses a small part of memory page as 'SmallFixedMemoryPage'

### 3.5.2 Constructor & Destructor Documentation

#### `FreeMap()`

```
internal::FreeMap::FreeMap (
    rtsha_page * page ) [explicit]
```

Constructs a `FreeMap` with the provided memory page.

##### Parameters

<code>page</code>	The <code>rtsha_page</code> that this <code>FreeMap</code> will manage.
-------------------	---

### 3.5.3 Member Function Documentation

#### `del()`

```
rtsha_attr_inline bool internal::FreeMap::del (
    const uint64_t key,
    size_t block ) [inline]
```

Deletes a key-value pair from the map based on the given key.

##### Parameters

<code>key</code>	The key of the key-value pair to be deleted.
<code>block</code>	The associated value for the key.

**Returns**

True if the deletion was successful, otherwise false.

**exists()**

```
rtsha_attr_inline bool internal::FreeMap::exists (
    const uint64_t key,
    size_t block ) [inline]
```

Checks if a given key-value pair exists in the map.

**Parameters**

<i>key</i>	The key to be checked.
<i>block</i>	The associated value for the key.

**Returns**

True if the key-value pair exists, otherwise false.

**find()**

```
rtsha_attr_inline size_t internal::FreeMap::find (
    const uint64_t key ) [inline]
```

Finds the value associated with a given key.

**Parameters**

<i>key</i>	The key to be looked up.
------------	--------------------------

**Returns**

The value associated with the key.

**insert()**

```
rtsha_attr_inline void internal::FreeMap::insert (
    const uint64_t key,
    size_t block ) [inline]
```

Inserts a key-value pair into the multimap.

**Parameters**

<i>key</i>	The key for the insertion. (The size of block is used as a key.)
<i>block</i>	The associated value for the key. (Address of the block in memory.)

**size()**

```
size_t internal::FreeMap::size () const [inline]
```

Retrieves the number of key-value pairs in the map.

**Returns**

The size of the map.

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

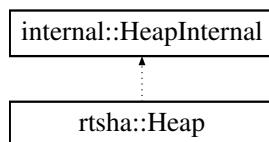
- FreeMap.h

## 3.6 rtsha::Heap Class Reference

This class encapsulates [Heap](#) in RTSHA.

```
#include <Heap.h>
```

Inheritance diagram for rtsha::Heap:



### Public Member Functions

- **Heap ()**

*Standard constructor.*

- **~Heap ()**

*Standard destructor.*

- **bool init (void \*start, size\_t size)**

*This function initializes heap.*

- **bool add\_page (HeapCallbacksStruct \*callbacks, rtsha\_page\_size\_type size\_type, size\_t size, size\_t max\_objects=0U, size\_t min\_block\_size=0U, size\_t max\_block\_size=0U)**

*This function creates memory page and adds it to the heap. RTSHA supports more than one pages per heap.*

- **size\_t get\_free\_space () const**

*This function returns free space of the heap.*

- **rtsha\_page \* get\_big\_memorypage () const**

*This function returns first Big Memory Page page on the heap.*

- **rtsha\_page \* get\_block\_page (address\_t block\_address)**

*This function returns the page of allocated block.*

- **void \* malloc (size\_t size)**

*This function allocates block of memory on the heap.*

- **void free (void \*ptr)**

*This function deallocates memory block.*

- **void \* calloc (size\_t nitems, size\_t size)**

*This function allocates the block of memory on the heap and initializes it to zero.*

- `void * realloc (void *ptr, size_t size)`

*This function reallocates the block of memory on the heap.*

- `void * memcpy (void *_Dst, void const *_Src, size_t _Size)`

*This function copies the values of num bytes from the location pointed to by source directly to the memory block pointed to by destination.*

- `void * memset (void *_Dst, int _Val, size_t _Size)`

*This function sets values of num bytes from the location pointed to by \_Dst to the specified value.*

- `rtsha_page_size_type get_ideal_page (size_t size) const`

*This function returns ideal page type based on size criteria.*

- `rtsha_page * select_page (rtsha_page_size_type ideal_page, size_t size, bool no_big=false) const`

*This function gets page from the list of heap pages.*

### 3.6.1 Detailed Description

This class encapsulates [Heap](#) in RTSHA.

### 3.6.2 Member Function Documentation

#### `add_page()`

```
bool rtsha::Heap::add_page (
    HeapCallbacksStruct * callbacks,
    rtsha_page_size_type size_type,
    size_t size,
    size_t max_objects = 0U,
    size_t min_block_size = 0U,
    size_t max_block_size = 0U )
```

This function creates memory page and adds it to the heap. RTSHA supports more than one pages per heap.

This function takes a singleton instance of the heap.

#### Parameters

<code>callbacks</code>	The <a href="#">HeapCallbacksStruct</a> with callback functions when used. NULL if 'callback' functions are not used. The callback functions for 'lock' and 'unlock' must be specified when used in multithreading environment.
<code>size</code>	The size of the page.
<code>size_type</code>	The type of the memory page.
<code>max_objects</code>	The maximum number of blocks that can exist on the page. This parameter is used exclusively for 'Big Memory Page' and 'Power of Two Memory Page'. For 'Small Fixed Memory Page', this value can be set to 0.
<code>min_block_size</code>	The minimal size of the page block. This parameter is used exclusively for 'Power of Two Memory Page'. A value of the parameter will be increased to the nearest value that is a power of 2. Please, set to 0 for 'Small Fixed Memory Page' and 'Big Memory Page'.
<code>max_block_size</code>	The maximum number of blocks that can exist on the page. This parameter is used exclusively for 'Big Memory Page' and 'Power of Two Memory Page'. When using 'Power Two Memory Page' a value of the parameter will be increased to the nearest value that is a power of 2. For 'Small Fixed Memory Page', this value can be set to 0.

**Returns**

Returns true when the page has been sucessfully created.

**calloc()**

```
void * rtsha::Heap::calloc (
    size_t nitems,
    size_t size )
```

This function allocates the block of memory on the heap and initializes it to zero.

The heap page will be automatically selected based on criteria such as size and availability.

**Parameters**

<i>nitems</i>	An unsigned integral value which represents number of elements. If the size is zero, a null pointer will be returned.
<i>size</i>	An unsigned integral value which represents the memory block in bytes. If the size is zero, a null pointer will be returned.

**Returns**

On success, a pointer to the memory block allocated by the function or null pointer if allocation fails.

**free()**

```
void rtsha::Heap::free (
    void * ptr )
```

This function deallocates memory block.

A block of memory previously allocated by a call to rtsha\_malloc, rtsha\_calloc or rtsha\_realloc is deallocated. It should not be used to release any memory block allocated any other way.

**Parameters**

<i>ptr</i>	Pointer to a previously allocated memory block. If ptr does not point to a valid block of memory allocated with rtsha_malloc, rtsha_calloc or rtsha_realloc, function does nothing.
------------	---

**get\_big\_memorypage()**

```
rtsha_page * rtsha::Heap::get_big_memorypage ( ) const
```

This function returns first Big Memory Page page on the heap.

**Returns**

Returns pointer to [rtsha\\_page](#) structure.

**get\_block\_page()**

```
rtsha_page * rtsha::Heap::get_block_page (
    address_t block_address )
```

This function returns the page of allocated block.

**Returns**

Returns pointer to [rtsha\\_page](#) structure or null pointer if fails.

**get\_free\_space()**

```
size_t rtsha::Heap::get_free_space ( ) const
```

This function returns free space of the heap.

**Returns**

Returns the number of free bytes on the heap.

**get\_ideal\_page()**

```
rtsha_page_size_type rtsha::Heap::get_ideal_page (
    size_t size ) const
```

This function returns ideal page type based on size criteria.

This function is not intended to be used by users of the library.

**Parameters**

<i>size</i>	Size of the memory block, in bytes.
-------------	-------------------------------------

**Returns**

Returns [rtsha\\_page\\_size\\_type](#).

**init()**

```
bool rtsha::Heap::init (
    void * start,
    size_t size )
```

This function initializes heap.

**Parameters**

<i>start</i>	The beginning of heap memory.
<i>size</i>	The size of heap memory.

**Returns**

Returns true when the heap has been sucessfully created.

**malloc()**

```
void * rtsha::Heap::malloc (
    size_t size )
```

This function allocates block of memory on the heap.

The heap page will be automatically selected based on criteria such as size and availability.

This method, depending on used memory page type, may allocate more than the number of bytes requested. If the block address is not so aligned, it will be rounded up to the next allocation granularity boundary.

**Parameters**

<i>size</i>	Size of the memory block, in bytes. If the size is zero, a null pointer will be returned. •
-------------	--

**Returns**

On success, a pointer to the memory block allocated by the function. The type of this pointer is always void\*, which can be cast to the desired type of data pointer in order to be dereferenceable. If the function failed to allocate the requested block of memory, a null pointer is returned.

**memcpy()**

```
void * rtsha::Heap::memcpy (
    void * _Dst,
    void const * _Src,
    size_t _Size )
```

This function copies the values of num bytes from the location pointed to by source directly to the memory block pointed to by destination.

Before copying memory from the source to the destination, the function checks if the source and destination memory addresses belong to the heap, whether the destination block is valid and not free, and if the size of the destination block is sufficiently large. If the destination does not belong to the heap memory, it will simply perform the copy function. Standard memcpy function is used.

**Parameters**

<i>_Dst</i>	Pointer to the destination.
<i>_Src</i>	Pointer to the source.
<i>_Size</i>	Number of bytes to copy.

**Returns**

On success, a pointer to the destination memory, or null pointer if the function fails.

**memset()**

```
void * rtsha::Heap::memset (
    void * _Dst,
    int _Val,
    size_t _Size )
```

This function sets values of num bytes from the location pointed to by *\_Dst* to the specified value.

Before calling standard 'memset' function, this function checks if the destination memory addresses belong to the heap, whether the destination block is valid and not free, and if the size of the destination block is sufficiently large. If the destination does not belong to the heap memory, it will simply perform the function.

**Parameters**

<i>_Dst</i>	Pointer to the destination.
<i>_Val</i>	Value to be set.
<i>_Size</i>	Number of bytes to be set to the specified value.

**Returns**

On success, a pointer to destination memory or null pointer if the function fails.

**realloc()**

```
void * rtsha::Heap::realloc (
    void * ptr,
    size_t size )
```

This function reallocates the block of memory on the heap.

The heap page will be automatically selected based on criteria such as size and availability.

The function may move the memory block to a new location. The content of the memory block is preserved up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion is indeterminate.

This method, depending on used memory page type, may allocate more than the number of bytes requested. If the block address is not so aligned, it will be rounded up to the next allocation granularity boundary.

**Parameters**

<i>ptr</i>	Pointer to the memory allocated with 'rtsha_malloc' or 'rtsha_calloc'
<i>size</i>	An unsigned integral value which represents the memory block in bytes. If the size is zero, a null pointer will be returned.

**Returns**

On success, a pointer to the memory block allocated by the function or null pointer if allocation fails.

**select\_page()**

```
rtsha_page * rtsha::Heap::select_page (
    rtsha_page_size_type ideal_page,
    size_t size,
    bool no_big = false ) const
```

This function gets page from the list of heap pages.

This function is not intended to be used by users of the library.

**Parameters**

<i>ideal_page</i>	Appropriate 'Page Type'.
<i>size</i>	Size of the memory block, in bytes.
<i>no_big</i>	Indicates to not use Big Memory Page.

**Returns**

On success, a pointer to memory page or null pointer if the function fails.

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

- Heap.h

### 3.7 HeapCallbacksStruct Struct Reference

Represents a collection of callback functions for heap operations.

```
#include <HeapCallbacks.h>
```

**Public Attributes**

- **rtshLockPagePtr ptrLockFunction**  
*Function to lock a page.*
- **rtshLockPagePtr ptrUnLockFunction**  
*Function to unlock a page.*
- **rtshErrorPagePtr ptrErrorFunction**  
*Function to handle errors.*

### 3.7.1 Detailed Description

Represents a collection of callback functions for heap operations.

This structure aggregates function pointers that the heap system can use to perform certain operations, such as locking, unlocking, or error handling.

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

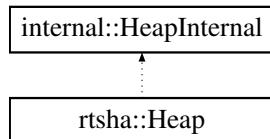
- `HeapCallbacks.h`

## 3.8 internal::HeapInternal Class Reference

This class encapsulates Heap.

```
#include <Heap.h>
```

Inheritance diagram for `internal::HeapInternal`:



### Public Member Functions

- **`HeapInternal ()`**  
*Standard constructor.*
- **`~HeapInternal ()`**  
*Standard destructor.*
- **`FreeList * createFreeList (rtsha_page *page)`**  
*This function creates a 'Free List Object' that will be used for the management of the 'free blocks' Free List object is simple 'forward\_list' or 'free linked list' used with custom memory allocator. The object is created in the predefined place on the stack using 'new placement' operator.*
- **`FreeMap * createFreeMap (rtsha_page *page)`**  
*This function creates a 'Free Map Object' that will be used for the management of the 'free blocks' Free Map is C++ Standard Library 'multimap' object that implements a sorted associative container, allowing for fast retrieval of values based on key.*
- **`FreeListArray * createFreeListArray (rtsha_page *page, size_t page_size)`**  
*This function creates a 'Free List Array Object' that will be used for the management of the Power2 Page 'free blocks'.*

### Protected Member Functions

- **`void init_small_fix_page (rtsha_page *page, size_t a_size)`**  
*Initialize a small fixed-sized page.*
- **`void init_power_two_page (rtsha_page *page, size_t a_size, size_t max_objects, size_t min_block_size, size_t max_block_size)`**  
*Initialize a page for handling power-of-two sized memory blocks.*
- **`void init_big_block_page (rtsha_page *page, size_t a_size, size_t max_objects)`**  
*Initialize a page for handling large memory blocks.*

## Protected Attributes

- `std::array<rtsha_page *, MAX_PAGES> _pages`  
*An array of pointers to manage the pages.*
- `size_t _number_pages = 0U`  
*Current number of pages managed by the heap.*
- `address_t _heap_start = 0U`  
*Starting address of the heap.*
- `size_t _heap_size = 0U`  
*Total size of the heap in bytes.*
- `address_t _heap_current_position = 0U`  
*Current position (or pointer) within the heap.*
- `address_t _heap_top = 0U`  
*The address marking the end of the heap.*
- `bool _heap_init = false`  
*Flag indicating if the heap has been initialized.*
- `uint32_t _last_heap_error = RTSHA_OK`  
*Last error code related to heap operations.*
- `bool _big_page_used`  
*Indicates that heap uses 'big memory' page.*
- `PREALLOC_MEMORY<FreeList,(MAX_SMALL_PAGES+MAX_BIG_PAGES)> _storage_free_lists = 0U`  
*Reserved storage on the stack for `FreeList` objects.*
- `PREALLOC_MEMORY<FreeListArray, MAX_POWER_TWO_PAGES > _storage_free_list_array = 0U`  
*Reserved storage on the stack for `FreeListArray` objects.*
- `PREALLOC_MEMORY<FreeMap, MAX_BIG_PAGES > _storage_free_maps = 0U`  
*Reserved storage on the stack for `FreeMap` objects.*

### 3.8.1 Detailed Description

This class encapsulates Heap.

This class encapsulates `HeapInternal` in RTSHA.

### 3.8.2 Member Function Documentation

#### `createFreeList()`

```
FreeList * internal::HeapInternal::createFreeList (
    rtsha_page * page )
```

This function creates a 'Free List Object' that will be used for the management of the 'free blocks' Free List object is simple 'forward\_list' or 'free linked list' used with custom memory allocator The object is created in the predefined place on the stack using 'new placement' operator.

This function is not intended to be used by users of RTSHA library!

#### Parameters

<code>page</code>	Pointer to page object's memory.
-------------------	----------------------------------

**Returns**

On success, a pointer to 'FreeList' object. If the function fails, it returns a null pointer.

**createFreeListArray()**

```
FreeListArray * internal::HeapInternal::createFreeListArray (
    rtsha_page * page,
    size_t page_size )
```

This function creates a 'Free List Arry Object' that will be used for the management of the Power2 Page 'free blocks'.

Free List Array object is simple array of 'linked Lists' The object is created in the predefined place on the stack using 'new placement' operator.

This function is not intended to be used by users of RTSHA library!

**Parameters**

<i>page</i>	Pointer to page object's memory.
-------------	----------------------------------

**Returns**

On success, a pointer to 'FreeList' object. If the function fails, it returns a null pointer.

**createFreeMap()**

```
FreeMap * internal::HeapInternal::createFreeMap (
    rtsha_page * page )
```

This function creates a 'Free Map Object' that will be used for the management of the 'free blocks' Free Map is C++ Standard Library 'multimap' object that implements a sorted associative container, allowing for fast retrieval of values based on key.

The object is created in the predefined place on the stack using 'new placement' operator.

This function is not intended to be used by users of RTSHA library!

**Parameters**

<i>page</i>	Pointer to page object's memory.
-------------	----------------------------------

**Returns**

On success, a pointer to 'FreeList' object. If the function fails, it returns a null pointer.

**init\_big\_block\_page()**

```
void internal::HeapInternal::init_big_block_page (
    rtsha_page * page,
```

```
size_t a_size,
size_t max_objects ) [protected]
```

Initialize a page for handling large memory blocks.

This method sets up the provided page for managing memory blocks that are considered 'large' or 'big'. The specifics of what constitutes 'large' would be based on the context in which this function is used.

#### Parameters

<i>page</i>	Pointer to the <code>rtsha_page</code> structure to be initialized.
<i>a_size</i>	Total size of the memory that the page will manage.
<i>max_objects</i>	Maximum number of large memory blocks that the page can handle.

### **init\_power\_two\_page()**

```
void internal::HeapInternal::init_power_two_page (
    rtsha_page * page,
    size_t a_size,
    size_t max_objects,
    size_t min_block_size,
    size_t max_block_size ) [protected]
```

Initialize a page for handling power-of-two sized memory blocks.

This method sets up the provided page for managing memory blocks that are sized as powers of two. The page can handle a range of block sizes between a specified minimum and maximum.

#### Parameters

<i>page</i>	Pointer to the <code>rtsha_page</code> structure to be initialized.
<i>a_size</i>	Total size of the memory that the page will manage.
<i>max_objects</i>	Maximum number of memory blocks that the page can handle.
<i>min_block_size</i>	Minimum size (inclusive) for memory blocks in this page.
<i>max_block_size</i>	Maximum size (inclusive) for memory blocks in this page.

### **init\_small\_fix\_page()**

```
void internal::HeapInternal::init_small_fix_page (
    rtsha_page * page,
    size_t a_size ) [protected]
```

Initialize a small fixed-sized page.

#### Parameters

<i>page</i>	Pointer to the <code>rtsha_page</code> structure to be initialized.
<i>a_size</i>	Size of each fixed-sized memory block in the page.

### 3.8.3 Member Data Documentation

#### \_big\_page\_used

```
bool internal::HeapInternal::_big_page_used [protected]
```

Indicates that heap uses 'big memory' page.

It's set to `RTSHA_OK` by default, indicating no errors.

#### \_heap\_current\_position

```
address_t internal::HeapInternal::_heap_current_position = 0U [protected]
```

Current position (or pointer) within the heap.

Typically indicates where the next memory allocation will take place.

#### \_last\_heap\_error

```
uint32_t internal::HeapInternal::_last_heap_error = RTSHA_OK [protected]
```

Last error code related to heap operations.

It's set to `RTSHA_OK` by default, indicating no errors.

#### \_pages

```
std::array<rtsha_page*, MAX_PAGES> internal::HeapInternal::_pages [protected]
```

An array of pointers to manage the pages.

This array keeps track of all the memory pages managed by the heap.

#### \_storage\_free\_list\_array

```
PREALLOC_MEMORY<FreeListArray, MAX_POWER_TWO_PAGES> internal::HeapInternal::_storage_free_list_array = 0U [protected]
```

Reserved storage on the stack for `FreeListArray` objects.

These area is reserver for objects that will be created with placement new operator, and this storage ensures there's space for them on the stack.

### **\_storage\_free\_lists**

```
PREALLOC_MEMORY<FreeList, (MAX_SMALL_PAGES + MAX_BIG_PAGES)> internal::HeapInternal::_storage->
_free_lists = 0U [protected]
```

Reserved storage on the stack for `FreeList` objects.

These area is reserver for objects that will be created with placement new operator, and this storage ensures there's space for them on the stack.

### **\_storage\_free\_maps**

```
PREALLOC_MEMORY<FreeMap, MAX_BIG_PAGES> internal::HeapInternal::_storage_free_maps = 0U [protected]
```

Reserved storage on the stack for `FreeMap` objects.

These area is reserver for objects that will be created with placement new operator, and this storage ensures there's space for them on the stack.

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

- `Heap.h`

## **3.9 `internal::InternListAllocator< T >` Class Template Reference**

Custom allocator tailored for internal list management.

```
#include <InternListAllocator.h>
```

### **Public Types**

- `typedef T value_type`  
*Defines the type of elements managed by the allocator.*

### **Public Member Functions**

- `InternListAllocator (rtsha_page *page, size_t *_ptrSmallStorage)`  
*Constructs the allocator with a given page and a pointer to a small storage.*
- `template<class U> constexpr InternListAllocator (const InternListAllocator< U > &rhs) noexcept`  
*Copy constructor allowing for type conversion.*
- `rtsha_attr_inline T * allocate (std::size_t n) noexcept`  
*Allocates a memory for an array of `n` objects of type `T`. It is called from 'forward\_list' every time when 'push' method is called Memory of the free block specified in `lastFreeBlockAddress` is used as a storage.*
- `rtsha_attr_inline void deallocate (T *, std::size_t) noexcept`  
*Deallocates memory. In this custom allocator, the deallocation is a no-op.*

## Public Attributes

- `rtsha_page * _page`  
*The memory page managed by the allocator.*
- `size_t _allocated_intern = 0U`  
*Counter to track a first few internal allocations.*
- `size_t * _ptrInternalSmallStorage = NULL`  
*Pointer to the small reserved storage.*

### 3.9.1 Detailed Description

`template<class T>`  
`class internal::InternListAllocator< T >`

Custom allocator tailored for internal list management.

This allocator is designed to efficiently manage memory for lists, leveraging specific characteristics of the `rtsha←_page` structure and a reserved small storage.

#### Template Parameters

<code>T</code>	The data type the allocator is responsible for.
----------------	---

### 3.9.2 Constructor & Destructor Documentation

#### `InternListAllocator()` [1/2]

```
template<class T >
internal::InternListAllocator< T >::InternListAllocator (
    rtsha_page * page,
    size_t * _ptrSmallStorage ) [inline], [explicit]
```

Constructs the allocator with a given page and a pointer to a small storage.

#### Parameters

<code>page</code>	Pointer to the <code>rtsha_page</code> the allocator should use to allocate the memory for 'free list'.
<code>_ptrSmallStorage</code>	Pointer to a small reserved storage.

#### `InternListAllocator()` [2/2]

```
template<class T >
template<class U >
constexpr internal::InternListAllocator< T >::InternListAllocator (
    const InternListAllocator< U > & rhs ) [inline], [constexpr], [noexcept]
```

Copy constructor allowing for type conversion.

called from 'std::forward\_list'

**Template Parameters**

<i>U</i>	The data type of the other allocator.
----------	---------------------------------------

**Parameters**

<i>rhs</i>	The other allocator to be copied from.
------------	--

**3.9.3 Member Function Documentation****allocate()**

```
template<class T >
rtsha_attr_inline T * internal::InternListAllocator< T >::allocate (
    std::size_t n ) [inline], [noexcept]
```

Allocates a memory for an array of *n* objects of type *T*. It is called from 'forward\_list' every time when 'push' method is called. Memory of the free block specified in lastFreeBlockAddress is used as a storage.

**Parameters**

<i>n</i>	Number of objects to allocate memory for.
----------	---

**Returns**

Pointer to the allocated block of memory.

**deallocate()**

```
template<class T >
rtsha_attr_inline void internal::InternListAllocator< T >::deallocate (
    T * ,
    std::size_t ) [inline], [noexcept]
```

Deallocates memory. In this custom allocator, the deallocation is a no-op.

It is called from 'forward\_list' every time when 'pop' method is called.

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

- InternListAllocator.h

**3.10 internal::InternMapAllocator< T > Struct Template Reference**

Allocator designed to handle internal memory allocations used with [FreeMap](#) class that is used to manage key-value pairs in memory.

```
#include <InternMapAllocator.h>
```

**Public Types**

- **typedef T value\_type**  
*Defines the type of elements managed by the allocator.*

**Public Member Functions**

- **InternMapAllocator (rtsha\_page \*page)**  
*Construct a new Intern Map Allocator object.*
- **template<class U >**  
**constexpr InternMapAllocator (const InternMapAllocator< U > &rhs) noexcept**  
*Copy constructor.*
- **rtsha\_attr\_inline T \* allocate (std::size\_t n) noexcept**  
*Allocate memory.*
- **rtsha\_attr\_inline void deallocate (T \*p, std::size\_t) noexcept**  
*Deallocate memory.*

**Public Attributes**

- **rtsha\_page \* \_page**  
*Memory page context.*

**3.10.1 Detailed Description**

**template<class T>**  
**struct internal::InternMapAllocator< T >**

Allocator designed to handle internal memory allocations used with `FreeMap` class that is used to manage key-value pairs in memory.

This custom allocator is optimized for managing memory in a specific context where memory is provided by an instance of `rtsha_page`.

**Template Parameters**

<b>T</b>	The type of elements being allocated.
----------	---------------------------------------

**3.10.2 Constructor & Destructor Documentation****InternMapAllocator() [1/2]**

```
template<class T >
internal::InternMapAllocator< T >::InternMapAllocator (
    rtsha_page * page ) [inline]
```

Construct a new Intern Map Allocator object.

**Parameters**

<i>page</i>	The memory page context in which the allocator will operate.
-------------	--

**InternMapAllocator() [2/2]**

```
template<class T >
template<class U >
constexpr internal::InternMapAllocator< T >::InternMapAllocator (
    const InternMapAllocator< U > & rhs ) [inline], [constexpr], [noexcept]
```

Copy constructor.

This constructor allows for the creation of an allocator of one type from another type, provided they have the same base template.

- called from 'std::multimap'

**Template Parameters**

<i>U</i>	The source type for the allocator.
----------	------------------------------------

**Parameters**

<i>rhs</i>	The source allocator.
------------	-----------------------

### 3.10.3 Member Function Documentation

**allocate()**

```
template<class T >
rtsha_attr_inline T * internal::InternMapAllocator< T >::allocate (
    std::size_t n ) [inline], [noexcept]
```

Allocate memory.

Attempt to allocate memory for *n* items of type *T*.

Memory in predefined 'map\_page' which is page of memory page will be used as storage. Allocator uses SmallFixMemoryPage together with 64 bytes blocks.

- It is called from 'std::multimap' every time when 'insert' method is called.

**Parameters**

<i>n</i>	Number of items of type <i>T</i> to allocate memory for.
----------	--

**Returns**

T\* Pointer to the allocated memory, or nullptr if allocation failed.

**deallocate()**

```
template<class T >
rtsha_attr_inline void internal::InternMapAllocator< T >::deallocate (
    T * p,
    std::size_t   ) [inline], [noexcept]
```

Deallocate memory.

Release previously allocated memory back to the 'map\_page' pool.

**Parameters**

<i>p</i>	Pointer to the memory to be deallocated.
----------	--

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

- [InternMapAllocator.h](#)

## 3.11 rtsha::MemoryBlock Class Reference

Provides an abstraction for handling blocks of memory.

```
#include <MemoryBlock.h>
```

**Public Member Functions**

- **MemoryBlock ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **MemoryBlock ([rtsha\\_block](#) \*block)**  
*Constructor that initializes the [MemoryBlock](#) with a given block.*
- **~MemoryBlock ()**  
*Destructor for the [MemoryBlock](#).*
- **MemoryBlock & operator= (const [MemoryBlock](#) &rhs)**  
*Overloaded assignment operator for the [MemoryBlock](#) class.*
- **void [split](#) (const size\_t &new\_size, bool last)**  
*Splits the current block into two blocks.*
- **void [split\\_22](#) ()**  
*Splits the current block into two blocks of the same size such that the old block is on the right side.*
- **void [merge\\_left](#) ()**  
*Merges the current block with the one to its left.*
- **void [merge\\_right](#) ()**  
*Merges the current block with the one to its right.*
- **rtsha\_attr\_inline void [setAllocated](#) ()**

- `rtsha_attr_inline void setFree ()`  
*Marks the current block as allocated.*
- `rtsha_attr_inline void setLast ()`  
*Marks the current block as the last block in the chain.*
- `rtsha_attr_inline void clearIsLast ()`  
*Clears the 'is last' status of the current block.*
- `rtsha_attr_inline rtsha_block * getBlock () const`  
*Retrieves the underlying `rtsha_block` pointer.*
- `rtsha_attr_inline void * getAllocAddress () const`  
*Retrieves the memory address allocated for the block.*
- `rtsha_attr_inline size_t getSize () const`  
*Gets the size of the current block.*
- `rtsha_attr_inline bool isValid () const`  
*Checks if the current `MemoryBlock` instance is valid.*
- `rtsha_attr_inline void setSize (size_t size)`  
*Sets the size of the current block.*
- `rtsha_attr_inline size_t getFreeBlockAddress () const`  
*Retrieves the address of the free block.*
- `rtsha_attr_inline void setPrev (const MemoryBlock &prev)`  
*Sets the previous block for the current block.*
- `rtsha_attr_inline void setAsFirst ()`  
*Sets the current block as the first block in a chain.*
- `rtsha_attr_inline bool isFree ()`  
*Checks if the current block is free.*
- `rtsha_attr_inline bool isLast ()`  
*Checks if the current block is the last block in the chain.*
- `rtsha_attr_inline bool hasPrev ()`  
*Checks if the current block has a predecessor.*
- `rtsha_attr_inline rtsha_block * getNextBlock () const`  
*Retrieves the next block in memory relative to the current block.*
- `rtsha_attr_inline rtsha_block * getPrev () const`  
*Retrieves the previous block relative to the current block.*
- `rtsha_attr_inline void prepare ()`  
*Prepares the current block for use by resetting its attributes in memory.*

### 3.11.1 Detailed Description

Provides an abstraction for handling blocks of memory.

This class offers various utility functions for manipulating a block of memory, including splitting, merging, and various getters and setters.

### 3.11.2 Constructor & Destructor Documentation

#### `MemoryBlock()`

```
rtsha::MemoryBlock::MemoryBlock (
    rtsha_block * block ) [inline], [explicit]
```

Constructor that initializes the `MemoryBlock` with a given block.

**Parameters**

<i>block</i>	The <a href="#">rtsha_block</a> to initialize the <a href="#">MemoryBlock</a> with.
--------------	---

### 3.11.3 Member Function Documentation

**getAllocAddress()**

```
rtsha_attr_inline void * rtsha::MemoryBlock::getAllocAddress () const [inline]
```

Retrieves the memory address allocated for the block.

**Returns**

The starting address of the allocated memory (block data).

**getBlock()**

```
rtsha_attr_inline rtsha_block * rtsha::MemoryBlock::getBlock () const [inline]
```

Retrieves the underlying [rtsha\\_block](#) pointer.

**Returns**

A pointer to the associated [rtsha\\_block](#).

**getFreeBlockAddress()**

```
rtsha_attr_inline size_t rtsha::MemoryBlock::getFreeBlockAddress () const [inline]
```

Retrieves the address of the free block.

**Returns**

The address of the free block.

**getNextBlock()**

```
rtsha_attr_inline rtsha_block * rtsha::MemoryBlock::getNextBlock () const [inline]
```

Retrieves the next block in memory relative to the current block.

**Returns**

A pointer to the next [rtsha\\_block](#) in the chain.

**getPrev()**

```
rtsha_attr_inline rtsha_block * rtsha::MemoryBlock::getPrev ( ) const [inline]
```

Retrieves the previous block relative to the current block.

**Returns**

A pointer to the previous [rtsha\\_block](#).

**getSize()**

```
rtsha_attr_inline size_t rtsha::MemoryBlock::getSize ( ) const [inline]
```

Gets the size of the current block.

**Returns**

The size of the block.

**hasPrev()**

```
rtsha_attr_inline bool rtsha::MemoryBlock::hasPrev ( ) [inline]
```

Checks if the current block has a predecessor.

**Returns**

True if the block has a previous block, otherwise false.

**isFree()**

```
rtsha_attr_inline bool rtsha::MemoryBlock::isFree ( ) [inline]
```

Checks if the current block is free.

The method examines the 0th bit of the size attribute to determine the block's status.

**Returns**

True if the block is free, otherwise false.

**isLast()**

```
rtsha_attr_inline bool rtsha::MemoryBlock::isLast ( ) [inline]
```

Checks if the current block is the last block in the chain.

The method examines the 1st bit of the size attribute to determine the block's position.

**Returns**

True if the block is the last block, otherwise false.

**isValid()**

```
rtsha_attr_inline bool rtsha::MemoryBlock::isValid ( ) const [inline]
```

Checks if the current [MemoryBlock](#) instance is valid.

**Returns**

True if the block is valid, otherwise false.

**operator=()**

```
MemoryBlock & rtsha::MemoryBlock::operator= (
    const MemoryBlock & rhs ) [inline]
```

Overloaded assignment operator for the [MemoryBlock](#) class.

This allows one [MemoryBlock](#) to be assigned to another, copying its underlying block reference.

**Parameters**

<i>rhs</i>	The right-hand side <a href="#">MemoryBlock</a> instance to assign from.
------------	--

**Returns**

A reference to the updated [MemoryBlock](#).

**setPrev()**

```
rtsha_attr_inline void rtsha::MemoryBlock::setPrev (
    const MemoryBlock & prev ) [inline]
```

Sets the previous block for the current block.

**Parameters**

<i>prev</i>	The previous <a href="#">MemoryBlock</a> to set.
-------------	--

**setSize()**

```
rtsha_attr_inline void rtsha::MemoryBlock::setSize (
    size_t size ) [inline]
```

Sets the size of the current block.

**Parameters**

<i>size</i>	The new size to set for the block.
-------------	------------------------------------

**splitt()**

```
void rtsha::MemoryBlock::splitt (
    const size_t & new_size,
    bool last )
```

Splits the current block into two blocks.

After splitting, the original block is resized and located on the left side.

**Parameters**

<i>new_size</i>	The size of the original block after the split.
<i>last</i>	Indicates if the new block should be the last in the chain.

**splitt\_22()**

```
void rtsha::MemoryBlock::splitt_22 ( )
```

Splits the current block into two blocks of the same size such that the old block is on the right side.

This is used when the old block is the last block in a chain.

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

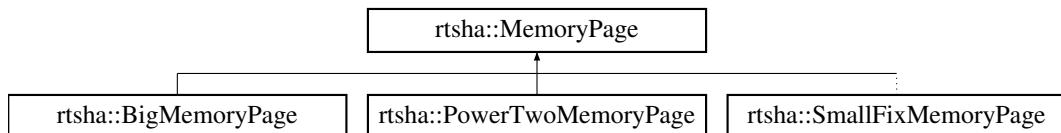
- MemoryBlock.h

## 3.12 rtsha::MemoryPage Class Reference

This is a base class representing a page in memory. It provides various memory handling functions that manipulate [MemoryBlock](#)'s.

```
#include <MemoryPage.h>
```

Inheritance diagram for rtsha::MemoryPage:



## Public Member Functions

- **MemoryPage ()=delete**  
*Default constructor is deleted.*
- **MemoryPage (rtsha\_page \*page) noexcept**  
*Constructor that initializes the [MemoryPage](#) with a given page.*
- **virtual ~MemoryPage ()**  
*Virtual destructor.*
- **bool checkBlock (size\_t address)**  
*Check if a block exists at the given address and if the block is valid.*
- **virtual void \* allocate\_block (const size\_t &size)=0**  
*Pure virtual function to allocate a block of memory.*
- **virtual void free\_block (MemoryBlock &block)=0**  
*Pure virtual function to free a block of memory.*
- **void \* allocate\_block\_at\_current\_pos (const size\_t &size)**  
*Allocates a block of memory at the current position.*
- **void incFreeBlocks ()**  
*Increments the count of free blocks.*

## Protected Member Functions

- **void lock ()**  
*Locks the page for thread-safe operations.*
- **void unlock ()**  
*Unlocks the page after thread-safe operations.*
- **void reportError (uint32\_t error)**  
*Reports an error using the specified error callback.*
- **void setFreeBlockAllocatorsAddress (const size\_t &address)**  
*Sets the address of the last free block temporary. The address will be used by InternListAllocator as storage for the elements of the 'std::forward\_list'.*
- **rtsha\_page\_size\_type getPageType () const**  
*Retrieves the type of the page.*
- **void \* getFreeList () const**  
*Gets the free list pointer of the page.*
- **void \* getFreeListArray () const**  
*Gets the free list array pointer of the page.*
- **void \* getFreeMap () const**  
*Gets the free map pointer of the page.*
- **size\_t getFreeBlocks () const**  
*Retrieves the number of free blocks in the page.*
- **size\_t getMinBlockSize () const**  
*Retrieves the number of free blocks in the page.*
- **address\_t getPosition () const**  
*Gets the current page position.*
- **void setPosition (address\_t pos)**  
*Sets the current page position.*
- **void incPosition (const size\_t &val)**  
*Increments the page position by the given value.*
- **void decPosition (const size\_t &val)**  
*Decreases the page position by the given value.*
- **void decFreeBlocks ()**

- Decreases the number of free blocks in the page.*
- `address_t getEndPosition () const`  
*Gets the end position of the page.*
  - `address_t getStartPosition () const`  
*Gets the start position of the page.*
  - `bool fitOnPage (const size_t &size) const`  
*Checks if a block of the specified size fits on the page.*
  - `bool hasLastBlock () const`  
*Checks if the page 'last block' has been set.*
  - `bool isLastPageBlock (MemoryBlock &block) const`  
*Determines if the provided block is the last block of the page.*
  - `rtsha_block * getLastBlock () const`  
*Gets the last block of the page.*
  - `void setLastBlock (const MemoryBlock &block)`  
*Sets the last block of the page.*

## Protected Attributes

- `rtsha_page * _page`  
*Pointer to the page structure in memory.*

### 3.12.1 Detailed Description

This is a base class representing a page in memory. It provides various memory handling functions that manipulate `MemoryBlock`'s.

### 3.12.2 Constructor & Destructor Documentation

#### `MemoryPage()`

```
rtsha::MemoryPage::MemoryPage (
    rtsha_page * page ) [inline], [explicit], [noexcept]
```

Constructor that initializes the `MemoryPage` with a given page.

##### Parameters

<code>page</code>	The pointer to page in memory
-------------------	-------------------------------

### 3.12.3 Member Function Documentation

#### `allocate_block()`

```
virtual void * rtsha::MemoryPage::allocate_block (
    const size_t & size ) [pure virtual]
```

Pure virtual function to allocate a block of memory.

**Parameters**

<code>size</code>	Size of the block to allocate.
-------------------	--------------------------------

**Returns**

A pointer to the allocated block.

Implemented in [rtsha::BigMemoryPage](#), [rtsha::PowerTwoMemoryPage](#), and [rtsha::SmallFixMemoryPage](#).

**allocate\_block\_at\_current\_pos()**

```
void * rtsha::MemoryPage::allocate_block_at_current_pos (
    const size_t & size )
```

Allocates a block of memory at the current position.

**Parameters**

<code>size</code>	Size of the block to allocate.
-------------------	--------------------------------

**Returns**

A pointer to the allocated block.

**checkBlock()**

```
bool rtsha::MemoryPage::checkBlock (
    size_t address )
```

Check if a block exists at the given address and if the block is valid.

**Parameters**

<code>address</code>	The address to check.
----------------------	-----------------------

**Returns**

True if the block exists, otherwise false.

**decPosition()**

```
void rtsha::MemoryPage::decPosition (
    const size_t & val ) [inline], [protected]
```

Decreases the oage position by the given value.

**Parameters**

<i>val</i>	The value to decrement the position by.
------------	---

**fitOnPage()**

```
bool rtsha::MemoryPage::fitOnPage (
    const size_t & size ) const [inline], [protected]
```

Checks if a block of the specified size fits on the page.

**Parameters**

<i>size</i>	The size of the block to check.
-------------	---------------------------------

**Returns**

True if the block fits, false otherwise.

**free\_block()**

```
virtual void rtsha::MemoryPage::free_block (
    MemoryBlock & block ) [pure virtual]
```

Pure virtual function to free a block of memory.

**Parameters**

<i>block</i>	The block of memory to be freed.
--------------	----------------------------------

Implemented in [rtsha::BigMemoryPage](#), [rtsha::PowerTwoMemoryPage](#), and [rtsha::SmallFixMemoryPage](#).

**getEndPosition()**

```
address_t rtsha::MemoryPage::getEndPosition () const [inline], [protected]
```

Gets the end position of the page.

**Returns**

The end position.

**getFreeBlocks()**

```
size_t rtsha::MemoryPage::getFreeBlocks () const [inline], [protected]
```

Retrieves the number of free blocks in the page.

**Returns**

The number of free blocks.

**getFreeList()**

```
void * rtsha::MemoryPage::getFreeList ( ) const [inline], [protected]
```

Gets the free list pointer of the page.

**Returns**

A pointer to the free list.

**getFreeListArray()**

```
void * rtsha::MemoryPage::getFreeListArray ( ) const [inline], [protected]
```

Gets the free list array pointer of the page.

**Returns**

A pointer to the free list array.

**getFreeMap()**

```
void * rtsha::MemoryPage::getFreeMap ( ) const [inline], [protected]
```

Gets the free map pointer of the page.

**Returns**

A pointer to the free map.

**getLastBlock()**

```
rtsha_block * rtsha::MemoryPage::getLastBlock ( ) const [inline], [protected]
```

Gets the last block of the page.

**Returns**

A pointer to the last block.

**getMinBlockSize()**

```
size_t rtsha::MemoryPage::getMinBlockSize ( ) const [inline], [protected]
```

Retrieves the number of free blocks in the page.

**Returns**

The number of free blocks.

**getPageType()**

```
rtsha_page_size_type rtsha::MemoryPage::getPageType () const [inline], [protected]
```

Retrieves the type of the page.

**Returns**

The type of the page.

**getPosition()**

```
address_t rtsha::MemoryPage::getPosition () const [inline], [protected]
```

Gets the current page position.

**Returns**

The current position.

**getStartPosition()**

```
address_t rtsha::MemoryPage::getStartPosition () const [inline], [protected]
```

Gets the start position of the page.

**Returns**

The start position.

**hasLastBlock()**

```
bool rtsha::MemoryPage::hasLastBlock () const [inline], [protected]
```

Checks if the page 'last block' has been set.

**Returns**

True if the page has a last block, false otherwise.

**incFreeBlocks()**

```
void rtsha::MemoryPage::incFreeBlocks () [inline]
```

Increments the count of free blocks.

If the `_page` is not null, this function increments the `free_blocks` count associated with the `_page`. Typically called when a block is freed.

**incPosition()**

```
void rtsha::MemoryPage::incPosition (
    const size_t & val ) [inline], [protected]
```

Increments the page position by the given value.

**Parameters**

<i>val</i>	The value to increment the position by.
------------	---

**isLastPageBlock()**

```
bool rtsha::MemoryPage::isLastPageBlock (
    MemoryBlock & block ) const [inline], [protected]
```

Determines if the provided block is the last block of the page.

**Parameters**

<i>block</i>	The block to check.
--------------	---------------------

**Returns**

True if it's the last block, false otherwise.

**lock()**

```
void rtsha::MemoryPage::lock () [inline], [protected]
```

Locks the page for thread-safe operations.

This method is used in conjunction with multithreading support to ensure that modifications to the page are synchronized.

**reportError()**

```
void rtsha::MemoryPage::reportError (
    uint32_t error ) [inline], [protected]
```

Reports an error using the specified error callback.

**Parameters**

<i>error</i>	The error code to report.
--------------	---------------------------

**setFreeBlockAllocatorsAddress()**

```
void rtsha::MemoryPage::setFreeBlockAllocatorsAddress (
    const size_t & address ) [inline], [protected]
```

Sets the address of the last free block temporary. The address will be used by InternListAllocator as storage for the elements of the 'std::forward\_list'.

**Parameters**

<code>address</code>	The address to set.
----------------------	---------------------

**setLastBlock()**

```
void rtsha::MemoryPage::setLastBlock (
    const MemoryBlock & block ) [inline], [protected]
```

Sets the last block of the page.

**Parameters**

<code>block</code>	The block to set as the last block.
--------------------	-------------------------------------

**setPosition()**

```
void rtsha::MemoryPage::setPosition (
    address_t pos ) [inline], [protected]
```

Sets the current page position.

**Parameters**

<code>pos</code>	The position to set.
------------------	----------------------

**unlock()**

```
void rtsha::MemoryPage::unlock ( ) [inline], [protected]
```

Unlocks the page after thread-safe operations.

This method complements the `lock` method by releasing the lock on the page.

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

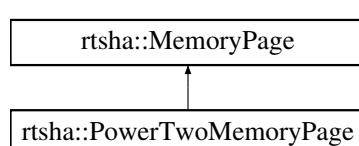
- `MemoryPage.h`

### 3.13 rtsha::PowerTwoMemoryPage Class Reference

This class provides various memory handling functions that manipulate `MemoryBlock`'s on 'Power two memory page'.

```
#include <PowerTwoMemoryPage.h>
```

Inheritance diagram for `rtsha::PowerTwoMemoryPage`:



## Public Member Functions

- **PowerTwoMemoryPage ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **PowerTwoMemoryPage (rtsha\_page \*page)**  
*Constructor that initializes the [PowerTwoMemoryPage](#) with a given page.*
- **virtual ~PowerTwoMemoryPage ()**  
*Virtual destructor for the [PowerTwoMemoryPage](#).*
- **virtual void \* allocate\_block (const size\_t &size) final**  
*Allocates a memory block of power 2 size.*
- **virtual void free\_block (MemoryBlock &block) final**  
*This function deallocates memory block.*
- **void createInitialFreeBlocks ()**  
*This function creates initial free blocks on empty memory page.*

## Public Member Functions inherited from [rtsha::MemoryPage](#)

- **MemoryPage ()=delete**  
*Default constructor is deleted.*
- **MemoryPage (rtsha\_page \*page) noexcept**  
*Constructor that initializes the [MemoryPage](#) with a given page.*
- **virtual ~MemoryPage ()**  
*Virtual destructor.*
- **bool checkBlock (size\_t address)**  
*Check if a block exists at the given address and if the block is valid.*
- **virtual void \* allocate\_block (const size\_t &size)=0**  
*Pure virtual function to allocate a block of memory.*
- **virtual void free\_block (MemoryBlock &block)=0**  
*Pure virtual function to free a block of memory.*
- **void \* allocate\_block\_at\_current\_pos (const size\_t &size)**  
*Allocates a block of memory at the current position.*
- **void incFreeBlocks ()**  
*Increments the count of free blocks.*

## Additional Inherited Members

### Protected Member Functions inherited from [rtsha::MemoryPage](#)

- **void lock ()**  
*Locks the page for thread-safe operations.*
- **void unlock ()**  
*Unlocks the page after thread-safe operations.*
- **void reportError (uint32\_t error)**  
*Reports an error using the specified error callback.*
- **void setFreeBlockAllocatorsAddress (const size\_t &address)**  
*Sets the address of the last free block temporary. The address will be used by InternListAllocator as storage for the elements of the 'std::forward\_list'.*
- **rtsha\_page\_size\_type getPageType () const**  
*Retrieves the type of the page.*
- **void \* getFreeList () const**

- `void * getFreeListArray () const`  
*Gets the free list pointer of the page.*
- `void * getFreeMap () const`  
*Gets the free map pointer of the page.*
- `size_t getFreeBlocks () const`  
*Retrieves the number of free blocks in the page.*
- `size_t getMinBlockSize () const`  
*Retrieves the number of free blocks in the page.*
- `address_t getPosition () const`  
*Gets the current page position.*
- `void setPosition (address_t pos)`  
*Sets the current page position.*
- `void incPosition (const size_t &val)`  
*Increments the page position by the given value.*
- `void decPosition (const size_t &val)`  
*Decreases the page position by the given value.*
- `void decFreeBlocks ()`  
*Decreases the number of free blocks in the page.*
- `address_t getEndPosition () const`  
*Gets the end position of the page.*
- `address_t getStartPosition () const`  
*Gets the start position of the page.*
- `bool fitOnPage (const size_t &size) const`  
*Checks if a block of the specified size fits on the page.*
- `bool hasLastBlock () const`  
*Checks if the page 'last block' has been set.*
- `bool isLastPageBlock (MemoryBlock &block) const`  
*Determines if the provided block is the last block of the page.*
- `rtsha_block * getLastBlock () const`  
*Gets the last block of the page.*
- `void setLastBlock (const MemoryBlock &block)`  
*Sets the last block of the page.*

#### Protected Attributes inherited from `rtsha::MemoryPage`

- `rtsha_page * _page`  
*Pointer to the page structure in memory.*

#### 3.13.1 Detailed Description

This class provides various memory handling functions that manipulate `MemoryBlock`'s on 'Power two memory page'.

This is a complex system, which only allows blocks of sizes that are powers of two. This makes merging free blocks back together easier and reduces fragmentation. A specialised binary search tree data structures (red-black tree) for fast storage and retrieval of ordered information are stored at the end of the page using fixed size Small Fix Memory Page.

This is a fairly efficient method of allocating memory, particularly useful for systems where memory fragmentation is an important concern. The algorithm divides memory into partitions to try to minimize fragmentation and the 'Best Fit' algorithm searches the page to find the smallest block that is large enough to satisfy the allocation.

Furthermore, this system is resistant to breakdowns due to its algorithmic approach to allocating and deallocating memory. The coalescing operation helps ensure that large contiguous blocks of memory can be reformed after they are freed, reducing the likelihood of fragmentation over time.

Coalescing relies on having free blocks of the same size available, which is not always the case, and so this system does not completely eliminate fragmentation but rather aims to minimize it.

### 3.13.2 Constructor & Destructor Documentation

#### **PowerTwoMemoryPage()**

```
rtsha::PowerTwoMemoryPage::PowerTwoMemoryPage (
    rtsha_page * page ) [inline]
```

Constructor that initializes the [PowerTwoMemoryPage](#) with a given page.

##### Parameters

<i>page</i>	The <a href="#">rtsha_page</a> structure to initialize the <a href="#">PowerTwoMemoryPage</a> with.
-------------	---

### 3.13.3 Member Function Documentation

#### **allocate\_block()**

```
rtsha::PowerTwoMemoryPage::allocate_block (
    const size_t & size ) [final], [virtual]
```

Allocates a memory block of power 2 size.

##### Parameters

<i>size</i>	The size of the memory block, in bytes.
-------------	---

##### Returns

On success, a pointer to the memory block allocated by the function.

Implements [rtsha::MemoryPage](#).

#### **createInitialFreeBlocks()**

```
rtsha::PowerTwoMemoryPage::createInitialFreeBlocks ( )
```

This function creates initial free blocks on empty memory page.

## free\_block()

```
rtsha::PowerTwoMemoryPage::free_block (
    MemoryBlock & block ) [final], [virtual]
```

This function deallocates memory block.

A block of previously allocated memory.

### Parameters

<code>block</code>	Previously allocated memory block.
--------------------	------------------------------------

Implements [rtsha::MemoryPage](#).

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

- [PowerTwoMemoryPage.h](#)

## 3.14 internal::PREALLOC\_MEMORY< T, n > Struct Template Reference

Memory storage template for pre-allocation.

```
#include <internal.h>
```

### Public Member Functions

- **PREALLOC\_MEMORY ()**  
*Default constructor.*
- **PREALLOC\_MEMORY (uint8\_t init)**  
*Constructor that initializes memory with a given value.*
- **void \* get\_ptr ()**  
*Retrieves the pointer to the beginning of the memory block.*
- **void \* get\_next\_ptr ()**  
*Retrieves the pointer to the next available memory block.*

### 3.14.1 Detailed Description

```
template<typename T, size_t n = 1U>
struct internal::PREALLOC_MEMORY< T, n >
```

Memory storage template for pre-allocation.

This template is designed to pre-allocate memory for objects on the stack.

### Template Parameters

<code>T</code>	Type of the elements the storage will manage.
<code>n</code>	Number of elements of type <code>T</code> the storage will manage. Default is 1.

### 3.14.2 Constructor & Destructor Documentation

#### PREALLOC\_MEMORY()

```
template<typename T , size_t n = 1U>
internal::PREALLOC_MEMORY< T, n >::PREALLOC_MEMORY (
    uint8_t init ) [inline]
```

Constructor that initializes memory with a given value.

##### Parameters

<i>init</i>	Value used to initialize the memory.
-------------	--------------------------------------

### 3.14.3 Member Function Documentation

#### get\_next\_ptr()

```
template<typename T , size_t n = 1U>
void * internal::PREALLOC_MEMORY< T, n >::get_next_ptr () [inline]
```

Retrieves the pointer to the next available memory block.

It increments the internal count to keep track of utilized memory blocks.

##### Returns

Void pointer to the next available memory block, or nullptr if no block is available.

#### get\_ptr()

```
template<typename T , size_t n = 1U>
void * internal::PREALLOC_MEMORY< T, n >::get_ptr () [inline]
```

Retrieves the pointer to the beginning of the memory block.

##### Returns

Void pointer to the beginning of the memory block.

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

- internal.h

## 3.15 rtsha::rtsha\_block Struct Reference

Represents a block of memory within a memory page.

```
#include <MemoryBlock.h>
```

## Public Member Functions

- **rtsha\_block ()**

*Default constructor for the block, initializing it to default values.*

## Public Attributes

- size\_t **size**
- **rtsha\_block \* prev**

*Pointer to the previous block.*

### 3.15.1 Detailed Description

Represents a block of memory within a memory page.

This structure provides the necessary attributes to manage a memory block, including its size and reference to a previous block.

### 3.15.2 Member Data Documentation

#### size

```
size_t rtsha::rtsha_block::size
```

Size of the block. Aligned size with the last two bits reserved for special flags. Bit 0 indicates free status, and bit 1 indicates if it's the last block.

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

- MemoryBlock.h

## 3.16 rtsha::rtsha\_page Struct Reference

Represents a page within the memory system.

```
#include <MemoryPage.h>
```

## Public Member Functions

- **rtsha\_page ()**

*Default constructor for initialization.*

**Public Attributes**

- **address\_t ptr\_list\_map** = 0U  
*Pointer or address to the list/map associated with the page.*
- **uint32\_t flags** = 0U  
*Flags associated with the page.*
- **address\_t start\_position** = 0U  
*Start address of page data.*
- **address\_t end\_position** = 0U  
*End address of the page.*
- **address\_t position** = 0U  
*Current position or address within the page.*
- **size\_t free\_blocks** = 0U  
*Number of free blocks within the page.*
- **rtsha\_block \* last\_block** = NULL  
*Pointer to the last block within the page.*
- **address\_t lastFreeBlockAddress** = 0U  
*Address of the last free block within the page.*
- **address\_t start\_map\_data** = 0U  
*Start address or position of map data for the page.*
- **rtsha\_page \* map\_page** = 0U  
*Associated map page if any. Used with together with 'Big Memory Page' and 'Power Two Page'.*
- **size\_t max\_blocks** = 0U  
*Maximum number of blocks supported by the page.*
- **size\_t min\_block\_size** = 0U  
*Minimum block size for the page (used with Power Two pages).*
- **size\_t max\_block\_size** = 0U  
*Maximum block size for the page (used with PowerTwo pages).*
- **HeapCallbacksStruct \* callbacks** = NULL  
*Callback functions associated with the page.*
- **rtsha\_page \* next** = NULL  
*Pointer to the next page similar structure.*

**3.16.1 Detailed Description**

Represents a page within the memory system.

This structure provides details about a page's layout, size, position, and associated blocks, as well as callback and linking mechanisms for managing the page in larger memory structures.

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

- MemoryPage.h

**3.17 rtsha::RTSHAllocator< T > Struct Template Reference**

Custom allocator leveraging the `rtsha_malloc` function for memory management.

```
#include <FastPlusAllocator.h>
```

## Public Types

- **typedef T value\_type**  
*Type of the elements being managed by the allocator.*

## Public Member Functions

- **RTSHAllocator ()=default**  
*Default constructor.*
- **template<class U >**  
**constexpr RTSHAllocator (const RTSHAllocator< U > &) noexcept**  
*Copy constructor.*
- **T \* allocate (std::size\_t n)**  
*Allocate memory.*
- **void deallocate (T \*p, std::size\_t n) noexcept**  
*Deallocate memory.*

### 3.17.1 Detailed Description

```
template<class T>
struct rtsha::RTSHAllocator< T >
```

Custom allocator leveraging the `rtsha_malloc` function for memory management.

This allocator provides a mechanism to utilize a specific memory management method (`rtsha_malloc` and `rtsha_free`) for allocation and deallocation.

It can be used as custom allocator for std containers.

Singleton RTSH Heap instance must be created.

#### Template Parameters

<i>T</i>	The type of elements being allocated.
----------	---------------------------------------

### 3.17.2 Constructor & Destructor Documentation

#### RTSHAllocator()

```
template<class T >
template<class U >
constexpr rtsha::RTSHAllocator< T >::RTSHAllocator (
    const RTSHAllocator< U > & ) [inline], [constexpr], [noexcept]
```

Copy constructor.

This constructor allows for the creation of an allocator of one type from another type, provided they have the same base template.

**Template Parameters**

<i>U</i>	The source type for the allocator.
----------	------------------------------------

**3.17.3 Member Function Documentation****allocate()**

```
template<class T >
T * rtsha::RTSHAllocator< T >::allocate (
    std::size_t n ) [inline]
```

Allocate memory.

Attempt to allocate memory for *n* items of type *T*.

**Parameters**

<i>n</i>	Number of items of type <i>T</i> to allocate memory for.
----------	--

**Returns**

*T\** Pointer to the allocated memory.

**Exceptions**

<i>std::bad_alloc</i>	If memory allocation fails.
<i>std::bad_array_new_length</i>	If the allocation size exceeds system limits.

**deallocate()**

```
template<class T >
void rtsha::RTSHAllocator< T >::deallocate (
    T * p,
    std::size_t n ) [inline], [noexcept]
```

Deallocate memory.

Release previously allocated memory.

**Parameters**

<i>p</i>	Pointer to the memory to be deallocated.
<i>n</i>	Number of items originally requested during allocation.

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

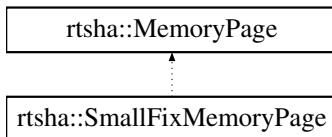
- FastPlusAllocator.h

## 3.18 rtsha::SmallFixMemoryPage Class Reference

This class provides various memory handling functions that manipulate [MemoryBlock](#)'s on memory page with fixed blocked size. This algorithm is an approach to memory management that is often used in specific situations where objects of a certain size are frequently allocated and deallocated. By using of uses 'Fixed chunk size' algorithm greatly simplifies the memory allocation process and reduce fragmentation.

```
#include <SmallFixMemoryPage.h>
```

Inheritance diagram for rtsha::SmallFixMemoryPage:



### Public Member Functions

- **SmallFixMemoryPage ()=delete**  
*Default constructor is deleted to prevent default instantiation.*
- **SmallFixMemoryPage (rtsha\_page \*page)**  
*Constructor that initializes the [SmallFixMemoryPage](#) with a given page.*
- virtual ~**SmallFixMemoryPage ()**  
*Virtual destructor for the [SmallFixMemoryPage](#).*
- virtual void \* **allocate\_block** (const size\_t &size) final  
*Allocates a memory block of fixed size.*
- virtual void **free\_block** ([MemoryBlock](#) &block) final  
*This function deallocates memory block.*

### 3.18.1 Detailed Description

This class provides various memory handling functions that manipulate [MemoryBlock](#)'s on memory page with fixed blocked size. This algorithm is an approach to memory management that is often used in specific situations where objects of a certain size are frequently allocated and deallocated. By using of uses 'Fixed chunk size' algorithm greatly simplifies the memory allocation process and reduce fragmentation.

The memory is divided into pages of chunks(blocks) of a fixed size (32, 64, 128, 256 and 512 bytes). When an allocation request comes in, it can simply be given one of these blocks. This means that the allocator doesn't have to search through the heap to find a block of the right size, which can improve performance. The free blocks memory is used as 'free list' storage.

Deallocations are also straightforward, as the block is added back to the list of available chunks. There's no need to merge adjacent free blocks, as there is with some other allocation strategies, which can also improve performance.

However, fixed chunk size allocation is not a good fit for all scenarios. It works best when the majority of allocations are of the same size, or a small number of different sizes. If allocations requests are of widely varying sizes, then this approach can lead to a lot of wasted memory, as small allocations take up an entire chunk, and large allocations require multiple chunks.

Small Fix Memory Page is also used internally by "Power Two Memory Page" and "Big Memory Page" algorithms.

### 3.18.2 Constructor & Destructor Documentation

#### **SmallFixMemoryPage()**

```
rtsha::SmallFixMemoryPage::SmallFixMemoryPage (
    rtsha_page * page ) [inline], [explicit]
```

Constructor that initializes the [SmallFixMemoryPage](#) with a given page.

**Parameters**

<i>page</i>	The <code>rtsha_page</code> structure to initialize the <code>SmallFixMemoryPage</code> with.
-------------	---

### 3.18.3 Member Function Documentation

**allocate\_block()**

```
rtsha::SmallFixMemoryPage::allocate_block (
    const size_t & size ) [final], [virtual]
```

Allocates a memory block of fixed size.

**Parameters**

<i>size</i>	Size of the memory block, in bytes.
-------------	-------------------------------------

**Returns**

On success, a pointer to the memory block allocated by the function.

Implements `rtsha::MemoryPage`.

**free\_block()**

```
rtsha::SmallFixMemoryPage::free_block (
    MemoryBlock & block ) [final], [virtual]
```

This function deallocates memory block.

A block of previously allocated memory.

**Parameters**

<i>block</i>	Previously allocated memory block.
--------------	------------------------------------

Implements `rtsha::MemoryPage`.

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

- `SmallFixMemoryPage.h`

## 4 File Documentation

### 4.1 allocator.h

```
00001 #pragma once
```

```

00002
00005 // This entire file will not be documented by Doxygen.
00006
00007
00008 #include "internal.h"
00009 #include "HeapCallbacks.h"
00010
00011
00012 #if defined(_MSC_VER) || defined(__MINGW32__)
00013
00014 #if !defined(RTSHA_SHARED_LIB)
00015 #define rtsha_decl_export
00016 #elif defined(RTSHA_SHARED_LIB_EXPORT)
00017 #define rtsha_decl_export           __declspec(dllexport)
00018 #else
00019 #define rtsha_decl_export           __declspec(dllimport)
00020 #endif
00021
00022 #define rtsha_cdecl                __cdecl
00023 #elif defined(__GNUC__)
00024 #if defined(rtsha_SHARED_LIB) && defined(rtsha_SHARED_LIB_EXPORT)
00025 #define rtsha_decl_export          __attribute__((visibility("default")))
00026 #else
00027 #define rtsha_decl_export
00028 #endif
00029 #define rtsha_cdecl
00030 #else
00031 #define rtsha_cdecl
00032 #define rtsha_decl_export
00033 #endif
00034
00035
00036
00037 #define RTSHA_PAGE_TYPE_32          32U
00038 #define RTSHA_PAGE_TYPE_64          64U
00039 #define RTSHA_PAGE_TYPE_128         128U
00040 #define RTSHA_PAGE_TYPE_256         256U
00041 #define RTSHA_PAGE_TYPE_512         512U
00042 #define RTSHA_PAGE_TYPE_BIG        613U
00043 #define RTSHA_PAGE_TYPE_POWER_TWO   713U
00044
00045
00046
00047 #ifdef __cplusplus
00048 extern "C"
00049 {
00050 #endif
00051 /*rtsha_decl_nodiscard rtsha_decl_export*/ bool rtsha_create_heap(void* start, size_t size);
00052
00053 rtsha_decl_export           bool rtsha_add_page(HeapCallbacksStruct* callbacks,
00054     uint16_t page_type, size_t size, size_t max_objects = 0U, size_t min_block_size = 0U, size_t
00055     max_block_size = 0U);
00056
00057 rtsha_decl_nodiscard rtsha_decl_export void* rtsha_malloc(size_t size);
00058
00059 rtsha_decl_nodiscard rtsha_decl_export void* rtsha_realloc(void* ptr, size_t size);
00060
00061 rtsha_decl_nodiscard rtsha_decl_export void* rtsha_calloc(size_t nitems, size_t size);
00062
00063 rtsha_decl_export           void* rtsha_memcpy(void* _Dst, void const* _Src, size_t
00064     _Size);
00065 rtsha_decl_export           void* rtsha_memset(void* _Dst, int _Val, size_t _Size);
00066
00067 #ifdef __cplusplus
00068 }
00069 #endif
00070

```

## 4.2 arm\_spec\_functions.h

```

00001 #pragma once
00002
00003 #ifdef __arm__ //ARM architecture
00004 #include <cstdint>
00005 #include <stddef.h>
00006
00007 void* arm_wide64_memcpy(void* dst, const void* src, size_t n);
00008 #endif

```

### 4.3 BigMemoryPage.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryPage.h"
00004
00005
00006 namespace rtsha
00007 {
00008     using namespace std;
00009
0018     class BigMemoryPage : public MemoryPage
0019     {
0020         public:
0021             BigMemoryPage() = delete;
0024
0029             explicit BigMemoryPage(rtsha_page* page) : MemoryPage(page)
0030             {
0031             }
0032
0034             virtual ~BigMemoryPage()
0035             {
0036             }
0037
0045             virtual void* allocate_block(const size_t& size) final;
0046
0051             virtual void free_block(MemoryBlock& block) final;
0052
0056             void createInitialFreeBlocks();
0057
0058     protected:
0059
0065             void splitBlock(MemoryBlock& block, size_t size);
0066
0071             void mergeLeft(MemoryBlock& block);
0072
0077             void mergeRight(MemoryBlock& block);
0078     };
0079 }
```

### 4.4 errors.h

```

00001 #include "internal.h"
00002
00003 #pragma once
00004
0011 #define RTSHA_OK                      (0U)
0012
0014 #define RTSHA_ErrorInit                (16U)
0015
0017 #define RTSHA_ErrorInitPageSize        (32U)
0018
0020 #define RTSHA_ErrorInitOutOfHeap       (33U)
0021
0023 #define RTSHA_OutOfMemory              (64U)
0024
0026 #define RTSHA_NoPages                 (128U)
0027
0029 #define RTSHA_NoPage                  (129U)
0030
0032 #define RTSHA_NoFreePage               (130U)
0033
0035 #define RTSHA_InvalidBlock             (256U)
0036
0038 #define RTSHA_InvalidBlockDistance    (257U)
0039
0041 #define RTSHA_InvalidNumberOfFreeBlocks (258U)
0042 // end of RTSHA_ERRORS group
0044
```

### 4.5 FastPlusAllocator.h

```

00001 #pragma once
00002 #include <cstdlib>
00003 #include <new>
00004 #include <limits>
00005 #include <iostream>
00006 #include "allocator.h"
00007
00008 namespace rtsha
```

```

00009 {
00010
00023     template<class T>
00024     struct RTSHAllocator
00025     {
00026         typedef T value_type;
00027
00028         RTSHAllocator() = default;
00029
00030
00033     template<class U>
00034     constexpr RTSHAllocator(const RTSHAllocator <U>&) noexcept {}
00035
00036     [[nodiscard]] T* allocate(std::size_t n)
00037     {
00038         if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
00039             throw std::bad_array_new_length();
00040
00041         if (auto p = static_cast<T*>(rtsha_malloc(n * sizeof(T))))
00042         {
00043             //report(p, n);
00044             return p;
00045         }
00046
00047         throw std::bad_alloc();
00048     }
00049
00050     void deallocate(T* p, std::size_t n) noexcept
00051     {
00052         //report(p, n, 0);
00053         rtsha_free(p);
00054     }
00055
00056     private:
00057
00058         void report(T* p, std::size_t n, bool alloc = true) const
00059         {
00060             std::cout << (alloc ? "Alloc: " : "Dealloc: ") << sizeof(T) * n
00061             << " bytes at " << std::hex << std::showbase
00062             << reinterpret_cast<void*>(p) << std::dec << '\n';
00063         }
00064
00065     };
00066
00067     template<class T, class U>
00068     bool operator==(const RTSHAllocator <T>&, const RTSHAllocator <U>&) { return true; }
00069
00070     template<class T, class U>
00071     bool operator!=(const RTSHAllocator <T>&, const RTSHAllocator <U>&) { return false; }
00072 }

```

## 4.6 FreeLinkedList.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "internal.h"
00004 #include "MemoryPage.h"
00005 #include <bitset>
00006
00007 namespace internal
00008 {
00009     using namespace rtsha;
00010
00011     class alignas(sizeof(size_t)) FreeLinkedList
00012     {
00013         struct alignas(sizeof(size_t)) Node
00014         {
00015             size_t data;
00016             Node* prev;
00017             Node* next;
00018
00019             explicit Node(const size_t& data, Node* head, Node* prev = nullptr) : data(data),
00020             next(head), prev(prev)
00021             {
00022             };
00023         };
00024
00025         public:
00026
00027             explicit FreeLinkedList(rtsha_page* page) : head(NULL), _page(page), count(0U)
00028             {
00029             }
00030
00031             ~FreeLinkedList()
00032             {
00033             }
00034
00035     };
00036
00037     template<class T>
00038     class alignas(sizeof(size_t)) FreeLinkedList<T>
00039     {
00040         Node<T> head;
00041         Node<T> tail;
00042         size_t count;
00043
00044         Node<T> alloc(size_t data)
00045         {
00046             Node<T> node = Node<T>(data, head.next, head);
00047             head.next = node;
00048             count++;
00049
00050             return node;
00051         }
00052
00053         Node<T> deallocate(Node<T> node)
00054         {
00055             Node<T> next = node.next;
00056             Node<T> prev = node.prev;
00057
00058             if (next)
00059                 next.prev = prev;
00060             if (prev)
00061                 prev.next = next;
00062
00063             count--;
00064
00065             return node;
00066         }
00067
00068         public:
00069
00070             FreeLinkedList() : head(nullptr), tail(nullptr), count(0U)
00071             {
00072             }
00073
00074             ~FreeLinkedList()
00075             {
00076                 while (head)
00077                     deallocate(head);
00078             }
00079
00080             void push_back(const T& data)
00081             {
00082                 tail.next = alloc(data);
00083                 tail = tail.next;
00084             }
00085
00086             void pop_back()
00087             {
00088                 Node<T> node = tail;
00089                 tail = tail.prev;
00090
00091                 deallocate(node);
00092             }
00093
00094             void clear()
00095             {
00096                 while (head)
00097                     deallocate(head);
00098             }
00099
00100             T front() const
00101             {
00102                 return head.data;
00103             }
00104
00105             T back() const
00106             {
00107                 return tail.data;
00108             }
00109
00110             size_t size() const
00111             {
00112                 return count;
00113             }
00114
00115             T at(size_t index) const
00116             {
00117                 Node<T> node = head;
00118                 for (size_t i = 0; i < index; i++)
00119                     node = node.next;
00120
00121                 return node.data;
00122             }
00123
00124             void swap(FreeLinkedList<T> &other)
00125             {
00126                 std::swap(head, other.head);
00127                 std::swap(tail, other.tail);
00128                 std::swap(count, other.count);
00129             }
00130
00131             void reverse()
00132             {
00133                 Node<T> current = head;
00134                 Node<T> previous = tail;
00135
00136                 while (current)
00137                 {
00138                     Node<T> next = current.next;
00139                     current.next = previous;
00140                     previous = current;
00141                     current = next;
00142                 }
00143
00144                 head = previous;
00145                 tail = current;
00146             }
00147
00148             void sort()
00149             {
00150                 Node<T> current = head;
00151
00152                 while (current)
00153                 {
00154                     Node<T> min_node = current;
00155                     Node<T> min_data = current.data;
00156
00157                     Node<T> next = current.next;
00158
00159                     while (next)
00160                     {
00161                         if (next.data < min_data)
00162                             min_node = next;
00163                             min_data = next.data;
00164
00165                         next = next.next;
00166                     }
00167
00168                     if (min_node != current)
00169                     {
00170                         std::swap(min_node.data, current.data);
00171                         std::swap(min_node.next, current.next);
00172                         std::swap(min_node.prev, current.prev);
00173                     }
00174
00175                     current = current.next;
00176                 }
00177             }
00178
00179             void merge(FreeLinkedList<T> &other)
00180             {
00181                 if (head == nullptr)
00182                     head = other.head;
00183                 else
00184                     tail.next = other.head;
00185
00186                 tail = other.tail;
00187
00188                 other.head = nullptr;
00189                 other.tail = nullptr;
00190
00191                 other.count = 0;
00192             }
00193
00194             void merge_inplace(FreeLinkedList<T> &other)
00195             {
00196                 if (head == nullptr)
00197                     head = other.head;
00198                 else
00199                     tail.next = other.head;
00200
00201                 tail = other.tail;
00202
00203                 other.head = nullptr;
00204                 other.tail = nullptr;
00205
00206                 other.count = 0;
00207             }
00208
00209             void merge_if_bigger(FreeLinkedList<T> &other)
00210             {
00211                 if (count > other.count)
00212                     merge_inplace(other);
00213                 else
00214                     other.merge_inplace(*this);
00215             }
00216
00217             void merge_if_smaller(FreeLinkedList<T> &other)
00218             {
00219                 if (other.count > count)
00220                     merge_inplace(other);
00221                 else
00222                     other.merge_inplace(*this);
00223             }
00224
00225             void merge_if_equal(FreeLinkedList<T> &other)
00226             {
00227                 if (count == other.count)
00228                     merge_inplace(other);
00229                 else
00230                     other.merge_inplace(*this);
00231             }
00232
00233             void merge_if_bigger_or_equal(FreeLinkedList<T> &other)
00234             {
00235                 if (count >= other.count)
00236                     merge_inplace(other);
00237                 else
00238                     other.merge_inplace(*this);
00239             }
00240
00241             void merge_if_smaller_or_equal(FreeLinkedList<T> &other)
00242             {
00243                 if (other.count >= count)
00244                     merge_inplace(other);
00245                 else
00246                     other.merge_inplace(*this);
00247             }
00248
00249             void merge_if_smaller_and_bigger(FreeLinkedList<T> &other)
00250             {
00251                 if (other.count < count)
00252                     merge_inplace(other);
00253                 else
00254                     other.merge_inplace(*this);
00255             }
00256
00257             void merge_if_bigger_and_smaller(FreeLinkedList<T> &other)
00258             {
00259                 if (count < other.count)
00260                     merge_inplace(other);
00261                 else
00262                     other.merge_inplace(*this);
00263             }
00264
00265             void merge_if_smaller_and_equal(FreeLinkedList<T> &other)
00266             {
00267                 if (other.count == count)
00268                     merge_inplace(other);
00269                 else
00270                     other.merge_inplace(*this);
00271             }
00272
00273             void merge_if_bigger_and_equal(FreeLinkedList<T> &other)
00274             {
00275                 if (count == other.count)
00276                     merge_inplace(other);
00277                 else
00278                     other.merge_inplace(*this);
00279             }
00280
00281             void merge_if_smaller_and_bigger_or_equal(FreeLinkedList<T> &other)
00282             {
00283                 if (other.count <= count)
00284                     merge_inplace(other);
00285                 else
00286                     other.merge_inplace(*this);
00287             }
00288
00289             void merge_if_bigger_and_smaller_or_equal(FreeLinkedList<T> &other)
00290             {
00291                 if (count <= other.count)
00292                     merge_inplace(other);
00293                 else
00294                     other.merge_inplace(*this);
00295             }
00296
00297             void merge_if_smaller_and_bigger_and_equal(FreeLinkedList<T> &other)
00298             {
00299                 if (other.count == count)
00300                     merge_inplace(other);
00301                 else
00302                     other.merge_inplace(*this);
00303             }
00304
00305             void merge_if_bigger_and_smaller_and_equal(FreeLinkedList<T> &other)
00306             {
00307                 if (count == other.count)
00308                     merge_inplace(other);
00309                 else
00310                     other.merge_inplace(*this);
00311             }
00312
00313             void merge_if_smaller_and_bigger_and_smaller(FreeLinkedList<T> &other)
00314             {
00315                 if (other.count < count)
00316                     merge_inplace(other);
00317                 else
00318                     other.merge_inplace(*this);
00319             }
00320
00321             void merge_if_bigger_and_smaller_and_bigger(FreeLinkedList<T> &other)
00322             {
00323                 if (count < other.count)
00324                     merge_inplace(other);
00325                 else
00326                     other.merge_inplace(*this);
00327             }
00328
00329             void merge_if_smaller_and_bigger_and_bigger(FreeLinkedList<T> &other)
00330             {
00331                 if (other.count < count)
00332                     merge_inplace(other);
00333                 else
00334                     other.merge_inplace(*this);
00335             }
00336
00337             void merge_if_bigger_and_smaller_and_bigger_or_equal(FreeLinkedList<T> &other)
00338             {
00339                 if (other.count <= count)
00340                     merge_inplace(other);
00341                 else
00342                     other.merge_inplace(*this);
00343             }
00344
00345             void merge_if_bigger_and_smaller_and_bigger_and_equal(FreeLinkedList<T> &other)
00346             {
00347                 if (other.count == count)
00348                     merge_inplace(other);
00349                 else
00350                     other.merge_inplace(*this);
00351             }
00352
00353             void merge_if_smaller_and_bigger_and_bigger_and_equal(FreeLinkedList<T> &other)
00354             {
00355                 if (other.count == count)
00356                     merge_inplace(other);
00357                 else
00358                     other.merge_inplace(*this);
00359             }
00360
00361             void merge_if_smaller_and_bigger_and_bigger_and_smaller(FreeLinkedList<T> &other)
00362             {
00363                 if (other.count < count)
00364                     merge_inplace(other);
00365                 else
00366                     other.merge_inplace(*this);
00367             }
00368
00369             void merge_if_bigger_and_smaller_and_bigger_and_smaller(FreeLinkedList<T> &other)
00370             {
00371                 if (other.count < count)
00372                     merge_inplace(other);
00373                 else
00374                     other.merge_inplace(*this);
00375             }
00376
00377             void merge_if_smaller_and_bigger_and_bigger_and_smaller_or_equal(FreeLinkedList<T> &other)
00378             {
00379                 if (other.count <= count)
00380                     merge_inplace(other);
00381                 else
00382                     other.merge_inplace(*this);
00383             }
00384
00385             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_equal(FreeLinkedList<T> &other)
00386             {
00387                 if (other.count == count)
00388                     merge_inplace(other);
00389                 else
00390                     other.merge_inplace(*this);
00391             }
00392
00393             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_equal(FreeLinkedList<T> &other)
00394             {
00395                 if (other.count == count)
00396                     merge_inplace(other);
00397                 else
00398                     other.merge_inplace(*this);
00399             }
00400
00401             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller(FreeLinkedList<T> &other)
00402             {
00403                 if (other.count < count)
00404                     merge_inplace(other);
00405                 else
00406                     other.merge_inplace(*this);
00407             }
00408
00409             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller(FreeLinkedList<T> &other)
00410             {
00411                 if (other.count < count)
00412                     merge_inplace(other);
00413                 else
00414                     other.merge_inplace(*this);
00415             }
00416
00417             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_or_equal(FreeLinkedList<T> &other)
00418             {
00419                 if (other.count <= count)
00420                     merge_inplace(other);
00421                 else
00422                     other.merge_inplace(*this);
00423             }
00424
00425             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_equal(FreeLinkedList<T> &other)
00426             {
00427                 if (other.count == count)
00428                     merge_inplace(other);
00429                 else
00430                     other.merge_inplace(*this);
00431             }
00432
00433             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_equal(FreeLinkedList<T> &other)
00434             {
00435                 if (other.count == count)
00436                     merge_inplace(other);
00437                 else
00438                     other.merge_inplace(*this);
00439             }
00440
00441             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00442             {
00443                 if (other.count < count)
00444                     merge_inplace(other);
00445                 else
00446                     other.merge_inplace(*this);
00447             }
00448
00449             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00450             {
00451                 if (other.count < count)
00452                     merge_inplace(other);
00453                 else
00454                     other.merge_inplace(*this);
00455             }
00456
00457             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_or_equal(FreeLinkedList<T> &other)
00458             {
00459                 if (other.count <= count)
00460                     merge_inplace(other);
00461                 else
00462                     other.merge_inplace(*this);
00463             }
00464
00465             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_equal(FreeLinkedList<T> &other)
00466             {
00467                 if (other.count == count)
00468                     merge_inplace(other);
00469                 else
00470                     other.merge_inplace(*this);
00471             }
00472
00473             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller_and_equal(FreeLinkedList<T> &other)
00474             {
00475                 if (other.count == count)
00476                     merge_inplace(other);
00477                 else
00478                     other.merge_inplace(*this);
00479             }
00480
00481             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00482             {
00483                 if (other.count < count)
00484                     merge_inplace(other);
00485                 else
00486                     other.merge_inplace(*this);
00487             }
00488
00489             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00490             {
00491                 if (other.count < count)
00492                     merge_inplace(other);
00493                 else
00494                     other.merge_inplace(*this);
00495             }
00496
00497             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00498             {
00499                 if (other.count < count)
00500                     merge_inplace(other);
00501                 else
00502                     other.merge_inplace(*this);
00503             }
00504
00505             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00506             {
00507                 if (other.count < count)
00508                     merge_inplace(other);
00509                 else
00510                     other.merge_inplace(*this);
00511             }
00512
00513             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00514             {
00515                 if (other.count < count)
00516                     merge_inplace(other);
00517                 else
00518                     other.merge_inplace(*this);
00519             }
00520
00521             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00522             {
00523                 if (other.count < count)
00524                     merge_inplace(other);
00525                 else
00526                     other.merge_inplace(*this);
00527             }
00528
00529             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00530             {
00531                 if (other.count < count)
00532                     merge_inplace(other);
00533                 else
00534                     other.merge_inplace(*this);
00535             }
00536
00537             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00538             {
00539                 if (other.count < count)
00540                     merge_inplace(other);
00541                 else
00542                     other.merge_inplace(*this);
00543             }
00544
00545             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00546             {
00547                 if (other.count < count)
00548                     merge_inplace(other);
00549                 else
00550                     other.merge_inplace(*this);
00551             }
00552
00553             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00554             {
00555                 if (other.count < count)
00556                     merge_inplace(other);
00557                 else
00558                     other.merge_inplace(*this);
00559             }
00560
00561             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00562             {
00563                 if (other.count < count)
00564                     merge_inplace(other);
00565                 else
00566                     other.merge_inplace(*this);
00567             }
00568
00569             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00570             {
00571                 if (other.count < count)
00572                     merge_inplace(other);
00573                 else
00574                     other.merge_inplace(*this);
00575             }
00576
00577             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00578             {
00579                 if (other.count < count)
00580                     merge_inplace(other);
00581                 else
00582                     other.merge_inplace(*this);
00583             }
00584
00585             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00586             {
00587                 if (other.count < count)
00588                     merge_inplace(other);
00589                 else
00590                     other.merge_inplace(*this);
00591             }
00592
00593             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00594             {
00595                 if (other.count < count)
00596                     merge_inplace(other);
00597                 else
00598                     other.merge_inplace(*this);
00599             }
00600
00601             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00602             {
00603                 if (other.count < count)
00604                     merge_inplace(other);
00605                 else
00606                     other.merge_inplace(*this);
00607             }
00608
00609             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00610             {
00611                 if (other.count < count)
00612                     merge_inplace(other);
00613                 else
00614                     other.merge_inplace(*this);
00615             }
00616
00617             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00618             {
00619                 if (other.count < count)
00620                     merge_inplace(other);
00621                 else
00622                     other.merge_inplace(*this);
00623             }
00624
00625             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00626             {
00627                 if (other.count < count)
00628                     merge_inplace(other);
00629                 else
00630                     other.merge_inplace(*this);
00631             }
00632
00633             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00634             {
00635                 if (other.count < count)
00636                     merge_inplace(other);
00637                 else
00638                     other.merge_inplace(*this);
00639             }
00640
00641             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00642             {
00643                 if (other.count < count)
00644                     merge_inplace(other);
00645                 else
00646                     other.merge_inplace(*this);
00647             }
00648
00649             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00650             {
00651                 if (other.count < count)
00652                     merge_inplace(other);
00653                 else
00654                     other.merge_inplace(*this);
00655             }
00656
00657             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00658             {
00659                 if (other.count < count)
00660                     merge_inplace(other);
00661                 else
00662                     other.merge_inplace(*this);
00663             }
00664
00665             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00666             {
00667                 if (other.count < count)
00668                     merge_inplace(other);
00669                 else
00670                     other.merge_inplace(*this);
00671             }
00672
00673             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00674             {
00675                 if (other.count < count)
00676                     merge_inplace(other);
00677                 else
00678                     other.merge_inplace(*this);
00679             }
00680
00681             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00682             {
00683                 if (other.count < count)
00684                     merge_inplace(other);
00685                 else
00686                     other.merge_inplace(*this);
00687             }
00688
00689             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00690             {
00691                 if (other.count < count)
00692                     merge_inplace(other);
00693                 else
00694                     other.merge_inplace(*this);
00695             }
00696
00697             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00698             {
00699                 if (other.count < count)
00700                     merge_inplace(other);
00701                 else
00702                     other.merge_inplace(*this);
00703             }
00704
00705             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00706             {
00707                 if (other.count < count)
00708                     merge_inplace(other);
00709                 else
00710                     other.merge_inplace(*this);
00711             }
00712
00713             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00714             {
00715                 if (other.count < count)
00716                     merge_inplace(other);
00717                 else
00718                     other.merge_inplace(*this);
00719             }
00720
00721             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00722             {
00723                 if (other.count < count)
00724                     merge_inplace(other);
00725                 else
00726                     other.merge_inplace(*this);
00727             }
00728
00729             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00730             {
00731                 if (other.count < count)
00732                     merge_inplace(other);
00733                 else
00734                     other.merge_inplace(*this);
00735             }
00736
00737             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00738             {
00739                 if (other.count < count)
00740                     merge_inplace(other);
00741                 else
00742                     other.merge_inplace(*this);
00743             }
00744
00745             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00746             {
00747                 if (other.count < count)
00748                     merge_inplace(other);
00749                 else
00750                     other.merge_inplace(*this);
00751             }
00752
00753             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00754             {
00755                 if (other.count < count)
00756                     merge_inplace(other);
00757                 else
00758                     other.merge_inplace(*this);
00759             }
00760
00761             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00762             {
00763                 if (other.count < count)
00764                     merge_inplace(other);
00765                 else
00766                     other.merge_inplace(*this);
00767             }
00768
00769             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00770             {
00771                 if (other.count < count)
00772                     merge_inplace(other);
00773                 else
00774                     other.merge_inplace(*this);
00775             }
00776
00777             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00778             {
00779                 if (other.count < count)
00780                     merge_inplace(other);
00781                 else
00782                     other.merge_inplace(*this);
00783             }
00784
00785             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00786             {
00787                 if (other.count < count)
00788                     merge_inplace(other);
00789                 else
00790                     other.merge_inplace(*this);
00791             }
00792
00793             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00794             {
00795                 if (other.count < count)
00796                     merge_inplace(other);
00797                 else
00798                     other.merge_inplace(*this);
00799             }
00800
00801             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00801             {
00802                 if (other.count < count)
00803                     merge_inplace(other);
00804                 else
00805                     other.merge_inplace(*this);
00806             }
00807
00808             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00809             {
00810                 if (other.count < count)
00811                     merge_inplace(other);
00812                 else
00813                     other.merge_inplace(*this);
00814             }
00815
00816             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00817             {
00818                 if (other.count < count)
00819                     merge_inplace(other);
00820                 else
00821                     other.merge_inplace(*this);
00822             }
00823
00824             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00825             {
00826                 if (other.count < count)
00827                     merge_inplace(other);
00828                 else
00829                     other.merge_inplace(*this);
00830             }
00831
00832             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00833             {
00834                 if (other.count < count)
00835                     merge_inplace(other);
00836                 else
00837                     other.merge_inplace(*this);
00838             }
00839
00840             void merge_if_bigger_and_smaller_and_bigger_and_smaller_and_smaller_and_smaller(FreeLinkedList<T> &other)
00841             {
00842                 if (other.count < count)
00843                     merge_inplace(other);
00844                 else
00845                     other.merge_inplace(*this);
00846             }
00847
00848             void merge_if_smaller_and_bigger_and_bigger_and_smaller_and_smaller_and_smaller_and_smaller(FreeLinkedList&lt
```

```
00059
00060     rtsha_attr_inline void push(const size_t& data)
00061     {
00062         Node* newNode = new (reinterpret_cast<void*>(_page->lastFreeBlockAddress)) Node(data,
00063         head, nullptr);
00064         if (head)
00065         {
00066             head->prev = newNode;
00067         }
00068         head = newNode;
00069         count++;
00070     }
00071
00072     rtsha_attr_inline bool is_empty() const
00073     {
00074         if (head == nullptr)
00075         {
00076             assert(count == 0);
00077             return true;
00078         }
00079         if (count == 0)
00080         {
00081             return true;
00082         }
00083         return false;
00084     }
00085
00086     rtsha_attr_inline size_t pop()
00087     {
00088         size_t ret(0U);
00089         if (!is_empty() && head != NULL)
00090         {
00091             Node* temp = head;
00092             ret = temp->data;
00093             head = head->next;
00094
00095             if( count > 0U )
00096                 count--;
00097
00098             if (count == 0U)
00099             {
00100                 head = NULL;
00101             }
00102             else
00103             {
00104                 count = 0U;
00105             }
00106             return ret;
00107         }
00108     }
00109
00110     rtsha_attr_inline bool delete_address(const size_t& address, void* block)
00111     {
00112         if (head == nullptr)
00113         {
00114             // the list is empty
00115             count = 0U;
00116             return false;
00117         }
00118         Node* temp = reinterpret_cast<Node*>(address);
00119
00120         if (head->data == temp->data)
00121         {
00122             //the node to delete is the head
00123             if (head->next)
00124                 head->next->prev = nullptr;
00125             head = head->next;
00126
00127             temp->data = 0U;
00128
00129             if (count)
00130                 count--;
00131             return true;
00132         }
00133         size_t blockAddress = reinterpret_cast<size_t>(block);
00134         if (temp->data == blockAddress)
00135         {
00136             temp->data = 0U;
00137             if (temp->prev)
00138             {
00139                 temp->prev->next = temp->next;
00140             }
00141             if (temp->next)
00142             {
00143                 temp->next->prev = temp->prev;
00144             }
00145         }
00146     }
```

---

```

00163         }
00164         if (count)
00165             count--;
00166         return true;
00167     }
00168     return false;
00169 }
00170
00171 private:
00172     rtsha_page* _page;
00173     Node* head;
00174     size_t count = 0U;
00175 };
00176 }
```

## 4.7 FreeList.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryBlock.h"
00004 #include "InternListAllocator.h"
00005 #include "FreeLinkedList.h"
00006
00007 namespace internal
00008 {
00009     using namespace std;
00010
00011     using flist = FreeLinkedList;
00012
00021     class alignas(sizeof(size_t)) FreeList
00022     {
00023     public:
00024
00026         FreeList() = delete;
00027
00032         explicit FreeList(rtsha_page* page);
00033
00035         ~FreeList()
00036         {
00037         }
00038
00044         rtsha_attr_inline void push(const size_t& address)
00045         {
00046             ptrLlist->push(address);
00047         }
00048
00054         rtsha_attr_inline size_t pop()
00055         {
00056             return ptrLlist->pop();
00057         }
00058
00059         rtsha_attr_inline bool delete_address(const size_t& address, void* block)
00060         {
00061             return (ptrLlist->delete_address(address, block));
00062         }
00063
00064     private:
00065
00066         rtsha_page* _page;
00067
00068         flist* ptrLlist;
00070
00072         PREALLOC_MEMORY<flist> _storage_list = 0U;
00073     };
00074 }
```

## 4.8 FreeListArray.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryBlock.h"
00004 #include "FreeLinkedList.h"
00005 #include "MemoryPage.h"
00006 #include <bit>
00007
00008 namespace internal
00009 {
00010     #define MIN_BLOCK_SIZE 32U
00011     #define MAX_BLOCK_SIZE 0x4000000U
```

```
00012     class alignas(sizeof(size_t)) FreeListArray
00020     {
00021
00022     public:
00024
00026         FreeListArray() = delete;
00027
00028         explicit FreeListArray(rtsha_page* page, size_t min_block_size, size_t page_size);
00029
00030         ~FreeListArray()
00031         {
00032         }
00033
00034         rtsha_attr_inline void push(const size_t data, size_t size)
00035         {
00036             if ((data > _page->start_position) && (data < _page->end_position))
00037             {
00038                 int32_t index = std::bit_width(size) - min_bin - 1;
00039                 assert(index >= 0);
00040                 if (index >= 0)
00041                 {
00042                     arrPtrLists[index]->push(data);
00043                 }
00044             }
00045
00046         rtsha_attr_inline size_t pop(size_t size)
00047         {
00048             size_t ret(0U);
00049             size_t log2_size = std::bit_width(size);
00050             assert(log2_size > min_bin);
00051             if (log2_size > min_bin)
00052             {
00053                 for (size_t n = log2_size; n <= max_bin; n++)
00054                 {
00055                     int32_t index = n - min_bin - 1;
00056                     assert(index >= 0);
00057                     if (index >= 0)
00058                     {
00059                         if (false == arrPtrLists[index]->is_empty())
00060                         {
00061                             ret = arrPtrLists[index]->pop();
00062                             if (ret > 0U)
00063                             {
00064                                 if ((ret > _page->start_position) && (ret < _page->end_position))
00065                                 {
00066                                     return ret;
00067                                 }
00068                             }
00069                         }
00070                     }
00071                 }
00072             }
00073             return 0U;
00074         }
00075
00076         rtsha_attr_inline bool delete_address(const size_t& address, void* block, const size_t& size)
00077         {
00078             if ((address > _page->start_position) && (address < _page->end_position))
00079             {
00080                 size_t log2_size = std::bit_width(size);
00081                 if (log2_size >= min_bin)
00082                 {
00083                     for (size_t n = log2_size; n <= max_bin; n++)
00084                     {
00085                         int32_t index = n - min_bin - 1;
00086                         assert(index >= 0);
00087                         if (index >= 0)
00088                         {
00089                             if (arrPtrLists[n - min_bin - 1]->delete_address(address, block))
00090                             {
00091                                 return true;
00092                             }
00093                         }
00094                     }
00095                 }
00096             }
00097             return false;
00098         }
00099
00100     private:
00101         size_t min_bin;
00102         size_t max_bin;
00103
00104
00105
00106
00107
00108
00109
00110
00111
00112
00113
00114
00115
00116
00117
00118
00119
00120
00121
00122
00123
00124
00125
00126
00127
00128
00129
00130
00131
00132
00133
00134
```

---

```

00135     rtsha_page* _page;
00136
00137     FreeLinkedList* arrPtrLists[MAX_BINS];
00138
00139     PREALLOC_MEMORY<FreeLinkedList, MAX_BINS> _storage_list_array = 0U;
00140
00141 };
00142 }

```

## 4.9 FreeMap.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryBlock.h"
00004 #include "InternMapAllocator.h"
00005 #include "internal.h"
00006 #include "map"
00007
00008 namespace internal
00009 {
00010     using namespace std;
00011     using namespace rtsha;
00012
00013     using mmap_allocator = InternMapAllocator<std::pair<const uint64_t, size_t>>;
00014
00015     using mmap = std::multimap<const uint64_t, size_t, std::less<const uint64_t>,
00016     internal::InternMapAllocator<std::pair<const uint64_t, size_t>>;
00017
00018     class alignas(sizeof(size_t)) FreeMap
00019     {
00020     public:
00021
00022         FreeMap() = delete;
00023
00024         explicit FreeMap(rtsha_page* page);
00025
00026         ~FreeMap()
00027         {
00028
00029             rtsha_attr_inline void insert(const uint64_t key, size_t block)
00030         {
00031             if (_ptrMap != nullptr)
00032             {
00033                 _ptrMap->insert(std::pair<const uint64_t, size_t>(key, block));
00034             }
00035         }
00036
00037         rtsha_attr_inline bool del(const uint64_t key, size_t block)
00038         {
00039             if (_ptrMap != nullptr)
00040             {
00041                 mmap::iterator it = _ptrMap->find(key);
00042                 while(it != _ptrMap->end())
00043                 {
00044                     if ((it->first == key) && (it->second == block))
00045                     {
00046                         it = _ptrMap->erase(it);
00047                         return true;
00048                     }
00049                     it++;
00050                 }
00051             }
00052             return false;
00053         }
00054
00055         rtsha_attr_inline size_t find(const uint64_t key)
00056         {
00057             if (_ptrMap != nullptr)
00058             {
00059                 mmap::iterator it = _ptrMap->lower_bound(key);
00060                 if (it != _ptrMap->end())
00061                 {
00062                     return it->second;
00063                 }
00064             }
00065             return 0U;
00066         }
00067
00068         rtsha_attr_inline bool exists(const uint64_t key, size_t block)
00069         {
00070             if (_ptrMap != nullptr)
00071             {
00072                 mmap::iterator it = _ptrMap->find(key);
00073

```

```

00116         if (it != _ptrMap->end())
00117             {
00118                 if ((it->first == key) && (it->second == block))
00119                 {
00120                     return true;
00121                 }
00122             }
00123         }
00124     }
00125 }
00126
00127     size_t size() const
00128     {
00129         if (_ptrMap != nullptr)
00130         {
00131             return _ptrMap->size();
00132         }
00133         return 0U;
00134     }
00135
00136
00137
00138
00139
00140
00141
00142
00143     private:
00144         rtsha_page*           _page;
00145         mmap_allocator*       _mallocator;
00146         mmap*                  _ptrMap;
00147
00148
00149     private:
00150
00151         PREALLOC_MEMORY <mmap_allocator>           _storage_allocator = 0U;
00152         PREALLOC_MEMORY<mmap>                      _storage_map = 0U;
00153     };
00154
00155 }
```

## 4.10 Heap.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryPage.h"
00004 #include "errors.h"
00005 #include "FreeList.h"
00006 #include "FreeListArray.h"
00007 #include "FreeMap.h"
00008 #include <array>
00009
00010 namespace internal
00011 {
00012     using namespace std;
00013
00025     class HeapInternal
00026     {
00027     public:
00031         HeapInternal():_big_page_used(false)
00032         {
00033             for (size_t i = 0; i < _pages.size(); i++)
00034             {
00035                 _pages[i] = nullptr;
00036             }
00037         }
00038
00042         ~HeapInternal()
00043         {
00044
00045         }
00046
00047
00060         FreeList* createFreeList(rtsha_page* page);
00061
00076         FreeMap* createFreeMap(rtsha_page* page);
00077
00091         FreeListArray* createFreeListArray(rtsha_page* page, size_t page_size);
00092
00093
00094     protected:
00095
00102         void init_small_fix_page(rtsha_page* page, size_t a_size);
00103
00117         void init_power_two_page(rtsha_page* page, size_t a_size, size_t max_objects, size_t
00118             min_block_size, size_t max_block_size);
00119
00130         void init_big_block_page(rtsha_page* page, size_t a_size, size_t max_objects);
00131
00132     protected:
```

```

00133     std::array<rtsha_page*, MAX_PAGES> _pages;
00140
00144     size_t      _number_pages = 0U;
00145
00149     address_t   _heap_start = 0U;
00150
00154     size_t      _heap_size = 0U;
00155
00159     address_t   _heap_current_position = 0U;
00162
00166     address_t   _heap_top = 0U;
00167
00171     bool        _heap_init = false;
00172
00178     uint32_t    _last_heap_error = RTSHA_OK;
00179
00185     bool _big_page_used;
00186
00187
00194     PREALLOC_MEMORY<FreeList, (MAX_SMALL_PAGES + MAX_BIG_PAGES)> _storage_free_lists = 0U;
00195
00202     PREALLOC_MEMORY<FreeListArray, MAX_POWER_TWO_PAGES> _storage_free_list_array = 0U;
00203
00210     PREALLOC_MEMORY<FreeMap, MAX_BIG_PAGES>      _storage_free_maps = 0U;
00211 };
00212 }
00213
00214 namespace rtsha
00215 {
00216     using namespace std;
00217     using namespace internal;
00218
00224     class Heap : HeapInternal
00225     {
00226         public:
00227
00231         Heap();
00232
00233
00237         ~Heap();
00238
00239
00248         bool init(void* start, size_t size);
00249
00250
00273         bool add_page(HeapCallbacksStruct* callbacks, rtsha_page_size_type size_type, size_t size,
00274             size_t max_objects = 0U, size_t min_block_size = 0U, size_t max_block_size = 0U);
00275
00280         size_t get_free_space() const;
00281
00287         rtsha_page* get_big_memorypage() const;
00288
00294         rtsha_page* get_block_page(address_t block_address);
00295
00296
00311         void* malloc(size_t size);
00312
00322         void free(void* ptr);
00323
00336         void* calloc(size_t nitems, size_t size);
00337
00357         void* realloc(void* ptr, size_t size);
00358
00376         void* memcpy(void* _Dst, void const* _Src, size_t _Size);
00377
00394         void* memset(void* _Dst, int _Val, size_t _Size);
00395
00396
00406         rtsha_page_size_type get_ideal_page(size_t size) const;
00407
00421         rtsha_page* select_page(rtsha_page_size_type ideal_page, size_t size, bool no_big = false)
00422             const;
00423     };
00424 }
00425

```

## 4.11 HeapCallbacks.h

```

00001 #pragma once
00002
00006 typedef void (*rtshLockPagePtr)      (void);
00007

```

```

00011 typedef void (*rtshUnLockPagePtr) (void);
00012
00018 typedef void (*rtshErrorPagePtr) (uint32_t);
00019
00027 typedef struct HeapCallbacksStruct
00028 {
00029     rtshLockPagePtr    ptrLockFunction;
00030     rtshLockPagePtr    ptrUnLockFunction;
00031     rtshErrorPagePtr   ptrErrorFunction;
00032 } HeapCallbacks;
00033

```

## 4.12 internal.h

```

00001 #pragma once
00002
00003 #include <assert.h>
00004 #include <cstring>
00005 #include "stdint.h"
00006
00007 #ifdef _MSC_VER
00008 # include <immintrin.h>
00009 # include <intrin.h>
00010 #else
00011 #endif
00012
00013
00014
00051 #define MULTITHREADING_SUPPORT
00052
00053
00054 #define MAX_BLOCKS_PER_PAGE UINT32_MAX
00055
00056 #define MAX_SMALL_PAGES 32U
00057 #define MAX_BIG_PAGES 2U
00058 #define MAX_POWER_TWO_PAGES 2U
00059
00060 #define MAX_PAGES (MAX_SMALL_PAGES+MAX_BIG_PAGES+MAX_POWER_TWO_PAGES)
00061
00062 #define MAX_BINS 27U
00063
00064 #if (_MSC_VER >= 1930 )
00065 #define rtsha_attr_inline inline __forceinline
00066 #else
00067 #define rtsha_attr_inline inline
00068 #endif
00069
00070
00071
00072 #if defined _WIN64 || defined _ARM64
00073 #define RTSHA_BLOCK_HEADER_SIZE (2 * sizeof(size_t))
00074 #define MIN_BLOCK_SIZE_FOR_SPLIT 56U /*todo*/
00075 #else
00076 #define RTSHA_LIST_ITEM_SIZE (2 * sizeof(size_t))
00077 #define MIN_BLOCK_SIZE_FOR_SPLIT 512U /*todo*/
00078 #endif
00079
00080 #ifdef __cplusplus
00081 #if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
00082 #define rtsha_attr_noexcept noexcept
00083 #else
00084 #define rtsha_attr_noexcept throw()
00085 #endif
00086 #else
00087 #define rtsha_attr_noexcept
00088 #endif
00089
00090
00091 #if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
00092 #define rtsha_decl_nodiscard [[nodiscard]]
00093 #elif (defined(__GNUC__) && (__GNUC__ >= 4)) || defined(__clang__)
00094 #define rtsha_decl_nodiscard __attribute__((warn_unused_result))
00095 #elif defined(_HAS_NODISCARD)
00096 #else
00097 #define rtsha_decl_nodiscard
00098 #endif
00099
00100
00101
00102
00103
00104 #define RTSHA_ALIGNMENT sizeof(size_t) /*4U or 8U*/
00105
00106 #define is_bit(val,n) ( (val >> n) & 0x01U )

```

```

00107
00108 #ifndef rtsha_assert
00109     #define rtsha_assert(x) assert(x)
00110 #endif
00111
00112
00113 #if __WIN32 || __WIN64
00114     #if defined(WIN64) || defined(__amd64__)
00115         #define ENV64BIT
00116     #else
00117         #define ENV32BIT
00118     #endif
00119 #else
00120     #if __GNUC__
00121         #if __x86_64__ || __ppc64__
00122             #define ENV64BIT
00123         #else
00124             #define ENV32BIT
00125         #endif
00126     #endif
00127 #endif
00128
00129
00130 #ifdef __arm__ //ARM architecture
00131 #define ARCH_ARM
00132 #endif
00133
00134 #ifdef __aarch64__ //ARM 64-bit
00135 #define ARCH_ARM
00136 #define ARCH_ARM_64
00137 #define ARCH_64BIT
00138 #endif
00139
00140
00141 namespace internal
00142 {
00143     using address_t = uintptr_t;
00144
00145 #if defined(WIN64) // The _BitScanReverse64 intrinsic is only available for 64 bit builds because it
depends on x64
00146     inline uint64_t ExpandToPowerOf2(uint64_t Value)
00147     {
00148         unsigned long Index;
00149         _BitScanReverse64(&Index, Value - 1);
00150         return (1ULL << (Index + 1));
00151     }
00152 }
00153 #else
00154
00155
00156 #ifdef __arm__ //ARM architecture
00157
00158     rtsha_attr_inline uint32_t ExpandToPowerOf2(uint32_t Value)
00159     {
00160         unsigned long leading_zeros = __builtin_clz(Value);
00161         return (1U << (32U - leading_zeros));
00162     }
00163
00164 #else
00165     inline uint32_t ExpandToPowerOf2(uint32_t Value)
00166     {
00167         unsigned long Index;
00168         _BitScanReverse(&Index, Value - 1);
00169         return (1U << (Index + 1));
00170     }
00171 #endif
00172 #endif
00173
00174
00183     template<typename T, size_t n = 1U>
00184     struct alignas(sizeof(size_t)) PREALLOC_MEMORY
00185     {
00186         public:
00187
00191         PREALLOC_MEMORY()
00192         {
00193         }
00194
00200         PREALLOC_MEMORY(uint8_t init)
00201         {
00202             std::memset(_memory, init, sizeof(_memory));
00203         }
00204
00205         public:
00206
00212         inline void* get_ptr()
00213         {

```

```

00214         return (_memory);
00215     }
00216
00224     inline void* get_next_ptr()
00225     {
00226         if (_count < n)
00227         {
00228             void* ret = (void*) (_memory + _count * sizeof(T));
00229             _count++;
00230             return ret;
00231         }
00232         return nullptr;
00233     }
00234 private:
00235     uint8_t _memory[n * sizeof(T)];
00236     size_t _count = 0U;
00237 };
00238
00247 static inline uintptr_t rtsha_align(uintptr_t ptr, size_t alignment)
00248 {
00249     uintptr_t mask = alignment - 1U;
00250
00251     if ((alignment & mask) == 0U)
00252     {
00253         return ((ptr + mask) & ~mask);
00254     }
00255     return (((ptr + mask) / alignment) * alignment);
00256 }
00257
00258 }
```

## 4.13 InternListAllocator.h

```

00001 #pragma once
00002 #include "MemoryPage.h"
00003 #include <cstdlib>
00004
00005 #ifdef __RTSHA_DIAGNOSTIK
00006 #include <iostream>
00007 #endif
00008
00009 namespace internal
00010 {
00011     using namespace rtsha;
00012
00022     template<class T>
00023     struct InternListAllocator
00024     {
00028         typedef T value_type;
00029
00036         explicit InternListAllocator(rtsha_page* page, size_t* _ptrSmallStorage)
00037             : _page(page),
00038             _allocated_intern(0U),
00039             _ptrInternalSmallStorage(_ptrSmallStorage)
00040         {
00041         }
00042
00043
00052         template<class U>
00053         constexpr InternListAllocator(const InternListAllocator <U>& rhs) noexcept
00054         {
00055             this->_page = rhs._page;
00056             this->_allocated_intern = rhs._allocated_intern;
00057             this->_ptrInternalSmallStorage = rhs._ptrInternalSmallStorage;
00058         }
00059
00068         [[nodiscard]] rtsha_attr_inline T* allocate(std::size_t n) noexcept
00069     {
00070         /*max. 1 block*/
00071         if (n != 1U)
00072         {
00073             return nullptr;
00074         }
00075         if (_allocated_intern == 0U) && (_page->lastFreeBlockAddress == 0U)
00076         {
00077             if (n > 1U)
00078             {
00079                 return nullptr;
00080             }
00081             _allocated_intern++;
00082             auto p = static_cast<T*>((void*) _ptrInternalSmallStorage);
00083             return p;
00084         }
00085         auto p = static_cast<T*>( (void*) _page->lastFreeBlockAddress);
00086     }
00087 }
```

```

00086         return p;
00087     }
00088
00095     rtsha_attr_inline void deallocate(T* /*p*/, std::size_t /*n*/) noexcept
00096     {
00097     }
00098
00099     rtsha_page* _page;
00100     size_t      _allocated_intern          = 0U;
00101     size_t*     _ptrInternalSmallStorage   = NULL;
00102
00103     private:
00104
00105 #ifdef _RTSHA_DIAGNOSTIK
00113     void report(T* p, std::size_t n, bool alloc = true) const
00114     {
00115         std::cout << (alloc ? "LAlloc: " : "LDealloc: ") << sizeof(T) * n
00116         << " bytes at " << std::hex << std::showbase
00117         << reinterpret_cast<void*>(p) << std::dec << '\n';
00118     }
00119 #endif
00120 };
00121
00129     template<class T, class U>
00130     bool operator==(const InternListAllocator <T>&, const InternListAllocator <U>&) { return true; }
00131
00139     template<class T, class U>
00140     bool operator!=(const InternListAllocator <T>&, const InternListAllocator <U>&) { return false; }
00141 }
00142

```

## 4.14 InternMapAllocator.h

```

00001 #pragma once
00002 #include "SmallFixMemoryPage.h"
00003 #include <cstdlib>
00004 #ifdef _RTSHA_DIAGNOSTIK
00005 #include <iostream>
00006 #endif
00007
00008 namespace internal
00009 {
00010     #if defined _WIN64 || defined _ARM64
00011         #define INTERNAL_MAP_STORAGE_SIZE 64U
00012     #else
00013         #define INTERNAL_MAP_STORAGE_SIZE 32U
00014     #endif
00015
00016     using namespace rtsha;
00017
00026     template<class T>
00027     struct InternMapAllocator
00028     {
00032         typedef T value_type;
00033
00039         InternMapAllocator(rtsha_page* page)
00040             :_page(page)
00041         {
00042         }
00043
00055         template<class U>
00056         constexpr InternMapAllocator(const InternMapAllocator <U>& rhs) noexcept
00057         {
00058             this->_page           = rhs._page;
00059         }
00060
00074         [[nodiscard]] rtsha_attr_inline T* allocate(std::size_t n)    noexcept
00075         {
00076             if (_page->map_page != nullptr)
00077             {
00078                 SmallFixMemoryPage map_page(_page->map_page);
00079                 if ((size_t)_page->map_page->flags >= (n * sizeof(T)))
00080                 {
00081                     auto p =
00082                         reinterpret_cast<T*>(map_page.allocate_block((size_t)_page->map_page->flags));
00083                     if (p != nullptr)
00084                     {
00085                         //report(p, n, 1);
00086                         return p;
00087                     }
00088                 }
00089             }
00090         }

```

```

00091     rtsha_attr_inline void deallocate(T*p, std::size_t /*n*/) noexcept
00092     {
00093         if (_page->map_page != nullptr)
00094         {
00095             SmallFixMemoryPage map_page(_page->map_page);
00096
00097             size_t address = reinterpret_cast<size_t>(p);
00098             address -= (2U * sizeof(size_t)); /*skip size and pointer to prev*/
00099
00100             MemoryBlock block((rtsha_block*)(void*)address);
00101
00102             if (block.isValid())
00103             {
00104                 map_page.free_block(block);
00105                 //report(p, n, 0);
00106             }
00107         }
00108     }
00109
00110     rtsha_page* _page;
00111
00112     private:
00113 #ifdef _RTSHA_DIAGNOSTIK
00114     void report(T* p, std::size_t n, bool alloc = true) const
00115     {
00116         std::cout << (alloc ? "MAlloc: " : "MDealloc: ") << sizeof(T) * n
00117         << " bytes at " << std::hex << std::showbase
00118         << reinterpret_cast<void*>(p) << std::dec << '\n';
00119     }
00120 #endif
00121 };
00122
00123 template<class T, class U>
00124 bool operator==(const InternMapAllocator <T> &, const InternMapAllocator <U> &) { return true; }
00125
00126 template<class T, class U>
00127 bool operator!=(const InternMapAllocator <T> &, const InternMapAllocator <U> &) { return false; }
00128
00129 }
```

## 4.15 MemoryBlock.h

```

00001 #pragma once
00002 #pragma once
00003 #include <stdint.h>
00004 #include "internal.h"
00005
00006 namespace rtsha
00007 {
00008     using namespace std;
00009
00010     struct rtsha_block
00011     {
00012         rtsha_block()
00013             : size(0U)
00014             , prev(NULL)
00015         {
00016         }
00017
00018         size_t           size;
00019         rtsha_block* prev;
00020     };
00021
00022     class MemoryBlock
00023     {
00024     public:
00025
00026         MemoryBlock() = delete;
00027
00028         explicit MemoryBlock(rtsha_block* block) : _block(block)
00029         {
00030         }
00031
00032         ~MemoryBlock()
00033         {
00034         }
00035
00036         MemoryBlock& operator = (const MemoryBlock& rhs)
00037         {
00038             this->_block = rhs._block;
00039             return *this;
00040         }
00041     };
00042 }
```

```

00072
00080     void splitt(const size_t& new_size, bool last);
00081
00087     void splitt_22();
00088
00092     void merge_left();
00093
00097     void merge_right();
00098
00099
00103     rtsha_attr_inline void setAllocated()
00104     {
00105         _block->size = (_block->size >> 1U) << 1U;
00106     }
00107
00111     rtsha_attr_inline void setFree()
00112     {
00113         _block->size = (_block->size | 1U);
00114     }
00115
00119     rtsha_attr_inline void setLast()
00120     {
00121         _block->size = (_block->size | 2U);
00122     }
00123
00127     rtsha_attr_inline void clearIsLast()
00128     {
00129         _block->size &= ~(1UL << 1U);
00130     }
00131
00136     rtsha_attr_inline rtsha_block* getBlock() const
00137     {
00138         return _block;
00139     }
00140
00145     rtsha_attr_inline void* getAllocAddress() const
00146     {
00147         return reinterpret_cast<void*>((size_t)_block + 2U * sizeof(size_t));
00148     }
00149
00154     rtsha_attr_inline size_t getSize() const
00155     {
00156         return (_block->size >> 2U) << 2U;
00157     }
00158
00163     rtsha_attr_inline bool isValid() const
00164     {
00165         if (_block != nullptr)
00166         {
00167             size_t size = getSize();
00168             if (_block != _block->prev) && (size > sizeof(size_t))
00169             {
00170                 size_t* ptrSize2 = reinterpret_cast<size_t*>((size_t)_block + size -
00171                     sizeof(size_t));
00172                 return (*ptrSize2 == size);
00173             }
00174             return false;
00175         }
00176
00181     rtsha_attr_inline void setSize( size_t size )
00182     {
00183         if (size > sizeof(size_t))
00184         {
00185             bool free = isFree();
00186             bool last = isLast();
00187             _block->size = size;
00188             if (free)
00189             {
00190                 setFree();
00191             }
00192             if (last)
00193             {
00194                 setLast();
00195             }
00196             size_t* ptrSize2 = reinterpret_cast<size_t*>((size_t)_block + size - sizeof(size_t));
00197             *ptrSize2 = size;
00198         }
00199         else
00200         {
00201             _block->size = 0U;
00202         }
00203     }
00204
00209     rtsha_attr_inline size_t getFreeBlockAddress() const
00210     {
00211         return ((size_t)_block + 2U * sizeof(size_t));

```

```

00212         }
00213
00218     rtsha_attr_inline void setPrev(const MemoryBlock& prev)
00219     {
00220         if (prev.isValid())
00221         {
00222             _block->prev = prev.getBlock();
00223         }
00224         else
00225         {
00226             _block->prev = NULL;
00227         }
00228     }
00229
00233     rtsha_attr_inline void setAsFirst()
00234     {
00235         _block->prev = NULL;
00236     }
00237
00244     rtsha_attr_inline bool isFree()
00245     {
00246         return is_bit(_block->size, 0U);
00247     }
00248
00255     rtsha_attr_inline bool isLast()
00256     {
00257         return is_bit(_block->size, 1U);
00258     }
00259
00265     rtsha_attr_inline bool hasPrev()
00266     {
00267         return (_block->prev != NULL);
00268     }
00269
00275     rtsha_attr_inline rtsha_block* getNextBlock() const
00276     {
00277         return reinterpret_cast<rtsha_block*>((size_t)_block + this->getSize());
00278     }
00279
00285     rtsha_attr_inline rtsha_block* getPrev() const
00286     {
00287         return _block->prev;
00288     }
00289
00293     rtsha_attr_inline void prepare()
00294     {
00295         _block->prev = NULL;
00296         _block->size = 0;
00297     }
00298
00299 private:
00300     rtsha_block* _block;
00301 };
00302 }
```

## 4.16 MemoryPage.h

```

00001 #pragma once
00002 #include "internal.h"
00003 #include <stdint.h>
00004 #include "MemoryBlock.h"
00005 #include "HeapCallbacks.h"
00006
00007
00008 namespace rtsha
00009 {
00010     using namespace std;
00011     using namespace internal;
00012
00020     enum struct rtsha_page_size_type : uint16_t
00021     {
00022         PageTypeNotDefined = 0U,
00023
00024         PageType16 = 16U,
00025         PageType32 = 32U,
00026         PageType64 = 64U,
00027         PageType128 = 128U,
00028         PageType256 = 256U,
00029         PageType512 = 512U,
00030
00031         PageTypeBig = 613U,
00032         PageTypePowerTwo = 713U
00033     };
00034 }
```

```

00043     struct rtsha_page
00044     {
00046         rtsha_page()
00047     {
00048     }
00049
00050     address_t          ptr_list_map      = 0U;
00051
00052     uint32_t            flags           = 0U;
00053
00054     address_t          start_position   = 0U;
00055     address_t          end_position    = 0U;
00056
00057     address_t          position        = 0U;
00058     size_t             free_blocks    = 0U;
00059
00060     rtsha_block*       last_block      = NULL;
00061
00062     address_t          lastFreeBlockAddress = 0U;
00063
00064     address_t          start_map_data = 0U;
00065
00066     rtsha_page*        map_page        = 0U;
00067
00068     size_t             max_blocks     = 0U;
00069
00070     size_t             min_block_size = 0U;
00071     size_t             max_block_size = 0U;
00072
00073     HeapCallbacksStruct* callbacks      = NULL;
00074
00075     rtsha_page*        next           = NULL;
00076 };
00077
00082 class MemoryPage
00083 {
00084 public:
00085
00087     MemoryPage() = delete;
00088
00093     explicit MemoryPage(rtsha_page* page) noexcept
00094     : _page(page)
00095     {
00096     }
00097
00099     virtual ~MemoryPage()
00100    {
00101    }
00102
00108     bool checkBlock(size_t address);
00109
00115     virtual void* allocate_block(const size_t& size) = 0;
00116
00121     virtual void free_block(MemoryBlock& block) = 0;
00122
00128     void* allocate_block_at_current_pos(const size_t& size);
00129
00136     inline void incFreeBlocks()
00137     {
00138         if (_page != nullptr)
00139         {
00140             _page->free_blocks++;
00141         }
00142     }
00143
00144 protected:
00145
00152     inline void lock()
00153     {
00154         #ifndef MULTITHREADING_SUPPORT
00155         if ((nullptr != this->_page->callbacks) && (nullptr !=
00156             this->_page->callbacks->ptrLockFunction))
00157         {
00158             this->_page->callbacks->ptrLockFunction();
00159         }
00160     }
00161
00167     inline void unlock()
00168     {
00169         #ifndef MULTITHREADING_SUPPORT
00170         if ((nullptr != this->_page->callbacks) && (nullptr !=
00171             this->_page->callbacks->ptrUnLockFunction))
00172         {
00173             this->_page->callbacks->ptrUnLockFunction();
00174         }
00175     }

```

```
00175         }
00176
00177     inline void reportError(uint32_t error)
00178     {
00179         if ((nullptr != this->_page->callbacks) && (nullptr !=
00180             this->_page->callbacks->ptrErrorFunction))
00181         {
00182             this->_page->callbacks->ptrErrorFunction(error);
00183         }
00184     }
00185
00186     inline void setFreeBlockAllocatorsAddress(const size_t& address)
00187     {
00188         _page->lastFreeBlockAddress = address;
00189     }
00200
00201     inline rtsha_page_size_type getPageType() const
00202     {
00203         return (rtsha_page_size_type) _page->flags;
00204     }
00210
00211     inline void* getFreeList() const
00212     {
00213         return reinterpret_cast<void*>(_page->ptr_list_map);
00214     }
00220
00221     inline void* getFreeListArray() const
00222     {
00223         return reinterpret_cast<void*>(_page->ptr_list_map);
00224     }
00230
00231     inline void* getFreeMap() const
00232     {
00233         return reinterpret_cast<void*>(_page->ptr_list_map);
00234     }
00240
00241     inline size_t getFreeBlocks() const
00242     {
00243         if (_page != nullptr)
00244         {
00245             return _page->free_blocks;
00246         }
00247         return 0U;
00248     }
00254
00255     inline size_t getMinBlockSize() const
00256     {
00257         if (_page != nullptr)
00258         {
00259             return _page->min_block_size;
00260         }
00261         return 0U;
00262     }
00268
00269     inline address_t getPosition() const
00270     {
00271         if (_page != nullptr)
00272         {
00273             return _page->position;
00274         }
00275         return 0U;
00276     }
00282
00283     inline void setPosition(address_t pos)
00284     {
00285         if (_page != nullptr)
00286         {
00287             _page->position = pos;
00288         }
00289     }
00295
00300     inline void incPosition(const size_t& val)
00301     {
00302         if (_page != nullptr)
00303         {
00304             _page->position += val;
00305         }
00306     }
00307
00308
00314     inline void decPosition(const size_t& val)
00315     {
00316         if (_page != nullptr)
00317         {
00318             if (_page->position >= val)
00319             {
00320                 _page->position -= val;
00321             }
00322         }
00323     }
00324 }
```

```

00322         }
00323     }
00324
00328     inline void decFreeBlocks()
00329     {
00330         if ( (_page != nullptr) && (_page->free_blocks > 0U) )
00331         {
00332             _page->free_blocks--;
00333         }
00334     }
00335
00341     inline address_t getPosition() const
00342     {
00343         return _page->start_position;
00344     }
00345
00351     inline address_t getStartPosition() const
00352     {
00353         return _page->start_position;
00354     }
00355
00362     inline bool fitOnPage(const size_t& size) const
00363     {
00364         if ((_page->position + size) < (_page->end_position))
00365         {
00366             if (_page->start_map_data == 0U)
00367             {
00368                 return true;
00369             }
00370             else
00371             {
00372                 if ((_page->position + size) < _page->start_map_data)
00373                 {
00374                     return true;
00375                 }
00376             }
00377         }
00378         return false;
00379     }
00380
00386     inline bool hasLastBlock() const
00387     {
00388         return (_page->last_block != nullptr);
00389     }
00390
00397     inline bool isLastPageBlock(MemoryBlock& block) const
00398     {
00399         if (this->getPosition() == ((size_t)block.getBlock() + block.getSize()))
00400         {
00401             return (block.getBlock() == _page->last_block);
00402         }
00403         return false;
00404     }
00405
00411     inline rtsha_block* getLastBlock() const
00412     {
00413         return _page->last_block;
00414     }
00415
00421     inline void setLastBlock(const MemoryBlock& block)
00422     {
00423         _page->last_block = block.getBlock();
00424     }
00425
00429     rtsha_page* _page;
00430
00431 };
00432 }
00433

```

## 4.17 PowerTwoMemoryPage.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "internal.h"
00004 #include "MemoryPage.h"
00005
00006 namespace rtsha
00007 {
00008     using namespace std;
00009
00028     class PowerTwoMemoryPage : public MemoryPage
00029     {
00030         public:

```

```
00031     PowerTwoMemoryPage() = delete;
00033
00034     PowerTwoMemoryPage(rtsha_page* page) : MemoryPage(page)
00039     {
00040     }
00042
00044     virtual ~PowerTwoMemoryPage()
00045     {
00046     }
00047
00048     virtual void* allocate_block(const size_t& size) final;
00049
00050     virtual void free_block(MemoryBlock& block) final;
00051
00052     void createInitialFreeBlocks();
00053 }
```

## 4.18 SmallFixMemoryPage.h

```
00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryPage.h"
00004
00005 namespace rtsha
00006 {
00007     using namespace std;
00008
00009
00010     class SmallFixMemoryPage : MemoryPage
00011     {
00012     public:
00013         SmallFixMemoryPage() = delete;
00014
00015         explicit SmallFixMemoryPage(rtsha_page* page) : MemoryPage(page)
00016         {
00017         }
00018
00019         virtual ~SmallFixMemoryPage()
00020         {
00021         }
00022
00023         virtual void* allocate_block(const size_t& size) final;
00024
00025         virtual void free_block(MemoryBlock& block) final;
00026     };
00027 }
```



# Index

\_big\_page\_used  
    internal::HeapInternal, 27

\_heap\_current\_position  
    internal::HeapInternal, 27

\_last\_heap\_error  
    internal::HeapInternal, 27

\_pages  
    internal::HeapInternal, 27

\_storage\_free\_list\_array  
    internal::HeapInternal, 27

\_storage\_free\_lists  
    internal::HeapInternal, 27

\_storage\_free\_maps  
    internal::HeapInternal, 28

add\_page  
    rtsha::Heap, 17

allocate  
    internal::InternListAllocator< T >, 30  
    internal::InternMapAllocator< T >, 32  
    rtsha::RTSHMemoryAllocator< T >, 55

allocate\_block  
    rtsha::BigMemoryPage, 5  
    rtsha::MemoryPage, 40  
    rtsha::PowerTwoMemoryPage, 49  
    rtsha::SmallFixMemoryPage, 57

allocate\_block\_at\_current\_pos  
    rtsha::MemoryPage, 41

allocator.h, 57

arm\_spec\_functions.h, 58

BigMemoryPage  
    rtsha::BigMemoryPage, 5

BigMemoryPage.h, 59

calloc  
    rtsha::Heap, 18

checkBlock  
    rtsha::MemoryPage, 41

createFreeList  
    internal::HeapInternal, 24

createFreeListArray  
    internal::HeapInternal, 25

createFreeMap  
    internal::HeapInternal, 25

createInitialFreeBlocks  
    rtsha::PowerTwoMemoryPage, 49

deallocate  
    internal::InternListAllocator< T >, 30  
    internal::InternMapAllocator< T >, 33  
    rtsha::RTSHMemoryAllocator< T >, 55

decPosition  
    rtsha::MemoryPage, 41

del  
    internal::FreeMap, 14

delete\_address  
    internal::FreeLinkedList, 9  
    internal::FreeListArray, 12

errors.h, 59

exists  
    internal::FreeMap, 15

FastPlusAllocator.h, 59

find  
    internal::FreeMap, 15

fitOnPage  
    rtsha::MemoryPage, 42

free  
    rtsha::Heap, 18

free\_block  
    rtsha::BigMemoryPage, 6  
    rtsha::MemoryPage, 42  
    rtsha::PowerTwoMemoryPage, 49  
    rtsha::SmallFixMemoryPage, 57

FreeLinkedList  
    internal::FreeLinkedList, 8

FreeLinkedList.h, 60

FreeList  
    internal::FreeList, 10

FreeList.h, 62

FreeListArray  
    internal::FreeListArray, 12

FreeListArray.h, 62

FreeMap  
    internal::FreeMap, 14

FreeMap.h, 64

get\_big\_memorypage  
    rtsha::Heap, 18

get\_block\_page  
    rtsha::Heap, 18

get\_free\_space  
    rtsha::Heap, 19

get\_ideal\_page  
    rtsha::Heap, 19

get\_next\_ptr  
    internal::PREALLOC\_MEMORY< T, n >, 51

get\_ptr  
    internal::PREALLOC\_MEMORY< T, n >, 51

getAllocAddress  
    rtsha::MemoryBlock, 35

getBlock  
    rtsha::MemoryBlock, 35

getEndPosition  
    rtsha::MemoryPage, 42

getFreeBlockAddress  
    rtsha::MemoryBlock, 35

getFreeBlocks  
    rtsha::MemoryPage, 42

getFreeList

rtsha::MemoryPage, 42  
 getFreeListArray  
     rtsha::MemoryPage, 43  
 getFreeMap  
     rtsha::MemoryPage, 43  
 getLastBlock  
     rtsha::MemoryPage, 43  
 getMinBlockSize  
     rtsha::MemoryPage, 43  
 getNextBlock  
     rtsha::MemoryBlock, 35  
 getPageType  
     rtsha::MemoryPage, 43  
 getPosition  
     rtsha::MemoryPage, 44  
 getPrev  
     rtsha::MemoryBlock, 35  
 getSize  
     rtsha::MemoryBlock, 36  
 getStartPosition  
     rtsha::MemoryPage, 44  
  
 hasLastBlock  
     rtsha::MemoryPage, 44  
 hasPrev  
     rtsha::MemoryBlock, 36  
 Heap.h, 65  
 HeapCallbacks.h, 66  
 HeapCallbacksStruct, 22  
  
 incFreeBlocks  
     rtsha::MemoryPage, 44  
 incPosition  
     rtsha::MemoryPage, 44  
 init  
     rtsha::Heap, 19  
 init\_big\_block\_page  
     internal::HeapInternal, 25  
 init\_power\_two\_page  
     internal::HeapInternal, 26  
 init\_small\_fix\_page  
     internal::HeapInternal, 26  
 insert  
     internal::FreeMap, 15  
 internal.h, 67  
 internal::FreeLinkedList, 8  
     delete\_address, 9  
     FreeLinkedList, 8  
     is\_empty, 9  
     pop, 9  
     push, 9  
 internal::FreeList, 10  
     FreeList, 10  
     pop, 11  
     push, 11  
 internal::FreeListArray, 11  
     delete\_address, 12  
     FreeListArray, 12  
     pop, 13  
  
 push, 13  
 internal::FreeMap, 13  
     del, 14  
     exists, 15  
     find, 15  
     FreeMap, 14  
     insert, 15  
     size, 16  
  
 internal::HeapInternal, 23  
     big\_page\_used, 27  
     heap\_current\_position, 27  
     last\_heap\_error, 27  
     pages, 27  
     storage\_free\_list\_array, 27  
     storage\_free\_lists, 27  
     storage\_free\_maps, 28  
     createFreeList, 24  
     createFreeListArray, 25  
     createFreeMap, 25  
     init\_big\_block\_page, 25  
     init\_power\_two\_page, 26  
     init\_small\_fix\_page, 26  
  
 internal::InternListAllocator< T >, 28  
     allocate, 30  
     deallocate, 30  
     InternListAllocator, 29  
  
 internal::InternMapAllocator< T >, 30  
     allocate, 32  
     deallocate, 33  
     InternMapAllocator, 31, 32  
  
 internal::PREALLOC\_MEMORY< T, n >, 50  
     get\_next\_ptr, 51  
     get\_ptr, 51  
     PREALLOC\_MEMORY, 51  
  
 InternListAllocator  
     internal::InternListAllocator< T >, 29  
  
 InternListAllocator.h, 69  
  
 InternMapAllocator  
     internal::InternMapAllocator< T >, 31, 32  
  
 InternMapAllocator.h, 70  
  
 is\_empty  
     internal::FreeLinkedList, 9  
  
 isFree  
     rtsha::MemoryBlock, 36  
  
 isLast  
     rtsha::MemoryBlock, 36  
  
 isLastPageBlock  
     rtsha::MemoryPage, 45  
  
 isValid  
     rtsha::MemoryBlock, 36  
  
 lock  
     rtsha::MemoryPage, 45  
  
 malloc  
     rtsha::Heap, 20  
  
 memcpy  
     rtsha::Heap, 20  
  
 MemoryBlock

rtsha::MemoryBlock, 34  
MemoryBlock.h, 71  
MemoryPage  
    rtsha::MemoryPage, 40  
MemoryPage.h, 73  
memset  
    rtsha::Heap, 21  
mergeLeft  
    rtsha::BigMemoryPage, 6  
mergeRight  
    rtsha::BigMemoryPage, 6  
  
operator=  
    rtsha::MemoryBlock, 37  
  
pop  
    internal::FreeLinkedList, 9  
    internal::FreeList, 11  
    internal::FreeListArray, 13  
PowerTwoMemoryPage  
    rtsha::PowerTwoMemoryPage, 49  
PowerTwoMemoryPage.h, 76  
PREALLOC\_MEMORY  
    internal::PREALLOC\_MEMORY< T, n >, 51  
push  
    internal::FreeLinkedList, 9  
    internal::FreeList, 11  
    internal::FreeListArray, 13  
  
Real Time Safety Heap Allocator, 1  
realloc  
    rtsha::Heap, 21  
reportError  
    rtsha::MemoryPage, 45  
RTSHA Error Codes, 2  
rtsha::BigMemoryPage, 3  
    allocate\_block, 5  
    BigMemoryPage, 5  
    free\_block, 6  
    mergeLeft, 6  
    mergeRight, 6  
    splitBlock, 6  
rtsha::Heap, 16  
    add\_page, 17  
    calloc, 18  
    free, 18  
    get\_big\_memorypage, 18  
    get\_block\_page, 18  
    get\_free\_space, 19  
    get\_ideal\_page, 19  
    init, 19  
    malloc, 20  
    memcpy, 20  
    memset, 21  
    realloc, 21  
    select\_page, 22  
rtsha::MemoryBlock, 33  
    getAllocAddress, 35  
    getBlock, 35  
    getFreeBlockAddress, 35  
    getNextBlock, 35  
    getPrev, 35  
    getSize, 36  
    hasPrev, 36  
    isFree, 36  
    isLast, 36  
    isValid, 36  
    MemoryBlock, 34  
    operator=, 37  
    setPrev, 37  
    setSize, 37  
    split, 38  
    split\_22, 38  
rtsha::MemoryPage, 38  
    allocate\_block, 40  
    allocate\_block\_at\_current\_pos, 41  
    checkBlock, 41  
    decPosition, 41  
    fitOnPage, 42  
    free\_block, 42  
    getEndPosition, 42  
    getFreeBlocks, 42  
    getFreeList, 42  
    getFreeListArray, 43  
    getFreeMap, 43  
    getLastBlock, 43  
    getMinBlockSize, 43  
    getPageType, 43  
    getPosition, 44  
    getStartPosition, 44  
    hasLastBlock, 44  
    incFreeBlocks, 44  
    incPosition, 44  
    isLastPageBlock, 45  
    lock, 45  
    MemoryPage, 40  
    reportError, 45  
    setFreeBlockAllocatorsAddress, 45  
    setLastBlock, 46  
    setPosition, 46  
    unlock, 46  
rtsha::PowerTwoMemoryPage, 46  
    allocate\_block, 49  
    createInitialFreeBlocks, 49  
    free\_block, 49  
    PowerTwoMemoryPage, 49  
rtsha::rtsha\_block, 51  
    size, 52  
rtsha::rtsha\_page, 52  
rtsha::RTSHAllocator< T >, 53  
    allocate, 55  
    deallocate, 55  
    RTSHAllocator, 54  
rtsha::SmallFixMemoryPage, 56  
    allocate\_block, 57  
    free\_block, 57  
    SmallFixMemoryPage, 56

RTSHAllocator  
rtsha::RTSHAllocator< T >, 54

select\_page  
rtsha::Heap, 22

setFreeBlockAllocatorsAddress  
rtsha::MemoryPage, 45

setLastBlock  
rtsha::MemoryPage, 46

setPosition  
rtsha::MemoryPage, 46

setPrev  
rtsha::MemoryBlock, 37

setSize  
rtsha::MemoryBlock, 37

size  
internal::FreeMap, 16  
rtsha::rtsha\_block, 52

SmallFixMemoryPage  
rtsha::SmallFixMemoryPage, 56

SmallFixMemoryPage.h, 77

splitBlock  
rtsha::BigMemoryPage, 6

splitt  
rtsha::MemoryBlock, 38

splitt\_22  
rtsha::MemoryBlock, 38

unlock  
rtsha::MemoryPage, 46