

RTSHA

Generated by Doxygen 1.9.7

1 Real Time Safety Heap Allocator	2
2 Module Documentation	2
2.1 RTSHA Error Codes	2
2.1.1 Detailed Description	3
3 Class Documentation	3
3.1 rtsha::BigMemoryPage Class Reference	3
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.3 Member Function Documentation	6
3.2 internal::FreeList Class Reference	8
3.2.1 Detailed Description	8
3.2.2 Constructor & Destructor Documentation	8
3.2.3 Member Function Documentation	8
3.3 internal::FreeMap Class Reference	9
3.3.1 Detailed Description	9
3.3.2 Constructor & Destructor Documentation	10
3.3.3 Member Function Documentation	10
3.4 rtsha::Heap Class Reference	12
3.4.1 Detailed Description	13
3.4.2 Member Function Documentation	13
3.5 HeapCallbacksStruct Struct Reference	18
3.5.1 Detailed Description	18
3.6 internal::HeapInternal Class Reference	18
3.6.1 Detailed Description	20
3.6.2 Member Function Documentation	20
3.6.3 Member Data Documentation	22
3.7 internal::InternListAllocator< T > Class Template Reference	23
3.7.1 Detailed Description	23
3.7.2 Constructor & Destructor Documentation	24
3.7.3 Member Function Documentation	24
3.8 internal::InternMapAllocator< T > Struct Template Reference	25
3.8.1 Detailed Description	25
3.8.2 Constructor & Destructor Documentation	26
3.8.3 Member Function Documentation	26
3.9 rtsha::MemoryBlock Class Reference	27
3.9.1 Detailed Description	29
3.9.2 Constructor & Destructor Documentation	29
3.9.3 Member Function Documentation	29
3.10 rtsha::MemoryPage Class Reference	33
3.10.1 Detailed Description	34
3.10.2 Constructor & Destructor Documentation	34

3.10.3 Member Function Documentation	35
3.11 rtsha::PowerTwoMemoryPage Class Reference	41
3.11.1 Detailed Description	43
3.11.2 Constructor & Destructor Documentation	43
3.11.3 Member Function Documentation	43
3.12 internal::PREALLOC_MEMORY< T, n > Struct Template Reference	45
3.12.1 Detailed Description	46
3.12.2 Constructor & Destructor Documentation	46
3.12.3 Member Function Documentation	46
3.13 rtsha::rtsha_block Struct Reference	47
3.13.1 Detailed Description	47
3.13.2 Member Data Documentation	47
3.14 rtsha::rtsha_page Struct Reference	48
3.14.1 Detailed Description	48
3.15 rtsha::RTSHAllocator< T > Struct Template Reference	49
3.15.1 Detailed Description	49
3.15.2 Constructor & Destructor Documentation	49
3.15.3 Member Function Documentation	50
3.16 rtsha::SmallFixMemoryPage Class Reference	51
3.16.1 Detailed Description	52
3.16.2 Constructor & Destructor Documentation	52
3.16.3 Member Function Documentation	52
4 File Documentation	53
4.1 allocator.h	53
4.2 BigMemoryPage.h	54
4.3 errors.h	54
4.4 FastPlusAllocator.h	55
4.5 FreeList.h	56
4.6 FreeMap.h	56
4.7 Heap.h	57
4.8 HeapCallbacks.h	58
4.9 internal.h	58
4.10 InternListAllocator.h	60
4.11 InternMapAllocator.h	61
4.12 MemoryBlock.h	62
4.13 MemoryPage.h	65
4.14 PowerTwoMemoryPage.h	68
4.15 SmallFixMemoryPage.h	68
Index	71

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. Deallocation 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 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. Big Memory Pages "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. The only distinction between the 'Power Two Memory Page' and this system is that the memory need not be divided into power-of-two blocks; variable block sizes are permitted. It promptly merges or coalesces memory blocks larger than 512 bytes after they are released. 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.

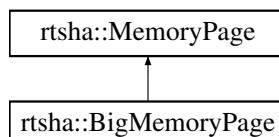
3 Class Documentation

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 (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 (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.
- **virtual void * allocate_block_at_current_pos (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** (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 * **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** (size_t val)
Increments the page position by the given value.
- void **decPosition** (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** (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

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

3.1.3 Member Function Documentation

allocate_block()

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

Allocates a block of memory of the specified size.

Parameters

<i>size</i>	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::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](#).
- **void push (const size_t address)**
Pushes a memory address onto the free list.
- **size_t pop ()**
Pops and retrieves a memory address from the free list.

3.2.1 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.

3.2.2 Constructor & Destructor Documentation

FreeList()

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

Constructs a [FreeList](#) with the given memory page.

Parameters

<code>page</code>	The <code>rtsha_page</code> that this FreeList should manage.
-------------------	---

3.2.3 Member Function Documentation

pop()

```
size_t internal::FreeList::pop ( )
```

Pops and retrieves a memory address from the free list.

Returns

The memory address retrieved from the free list.

push()

```
void internal::FreeList::push (
    const size_t address )
```

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`

3.3 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.
- void **insert** (const uint64_t key, size_t block)
Inserts a key-value pair into the multimap.
- bool **del** (const uint64_t key, size_t block)
Deletes a key-value pair from the map based on the given key.
- size_t **find** (const uint64_t key)
Finds the value associated with a given key.
- 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.

3.3.1 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'

3.3.2 Constructor & Destructor Documentation

FreeMap()

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

Constructs a [FreeMap](#) with the provided memory page.

Parameters

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

3.3.3 Member Function Documentation

del()

```
bool internal::FreeMap::del (
    const uint64_t key,
    size_t block )
```

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

Parameters

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

Returns

True if the deletion was successful, otherwise false.

exists()

```
bool internal::FreeMap::exists (
    const uint64_t key,
    size_t block )
```

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()

```
size_t internal::FreeMap::find (
    const uint64_t key )
```

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()

```
void internal::FreeMap::insert (
    const uint64_t key,
    size_t block )
```

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
```

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:

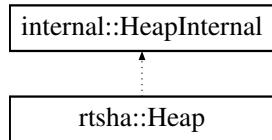
- FreeMap.h

3.4 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.

3.4.1 Detailed Description

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

3.4.2 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

<i>callbacks</i>	The HeapCallbacksStruct with callbac functions when used. NULL if 'callback' functions are not used. The callback functions for 'lock' and 'unlock' must be specified when used in multithreding enviroment.
<i>size</i>	The size of the page.
<i>size_type</i>	The type of the memory page.
<i>max_objects</i>	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.
<i>min_block_size</i>	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'.
<i>max_block_size</i>	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 successfully 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

<code>size</code>	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

<code>_Dst</code>	Pointer to the destination.
<code>_Src</code>	Pointer to the source.
<code>_Size</code>	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

<code>_Dst</code>	Pointer to the destination.
<code>_Val</code>	Value to be set.
<code>_Size</code>	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

<code>ptr</code>	Pointer to the memory allocated with 'rtsha_malloc' or 'rtsha_calloc'
<code>size</code>	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

3.5 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.

3.5.1 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:

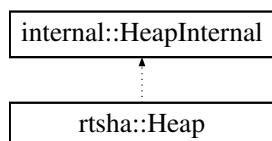
- HeapCallbacks.h

3.6 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' 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.

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 = NULL**
Starting address of the heap.
- **size_t _heap_size = 0U**
Total size of the heap in bytes.
- **address_t _heap_current_position = NULL**
Current position (or pointer) within the heap.
- **address_t _heap_top = NULL**
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.
- **PREALLOC_MEMORY< FreeList,(MAX_SMALL_PAGES+MAX_BIG_PAGES)> _storage_free_lists = 0U**
Reserved storage on the stack for FreeList objects.
- **PREALLOC_MEMORY< FreeMap, MAX_BIG_PAGES > _storage_free_maps = 0U**
Reserved storage on the stack for FreeMap objects.

3.6.1 Detailed Description

This class encapsulates Heap.

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

3.6.2 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' 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

<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

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

3.6.3 Member Data Documentation

_heap_current_position

```
address_t internal::HeapInternal::_heap_current_position = NULL [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 `RTSZA_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_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`

3.7 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.
- **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.*
- **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.

3.7.1 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

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

3.7.2 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

<i>page</i>	Pointer to the rtsha_page the allocator should use to allocate the memory for 'free list'.
<i>_ptrSmallStorage</i>	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.
------------	--

3.7.3 Member Function Documentation

allocate()

```
template<class T >
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 >
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`

3.8 `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.
- `T * allocate (std::size_t n) noexcept`
Allocate memory.
- `void deallocate (T *p, std::size_t) noexcept`
Deallocate memory.

Public Attributes

- `rtsha_page * _page`
Memory page context.

3.8.1 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

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

3.8.2 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.
------------	-----------------------

3.8.3 Member Function Documentation

allocate()

```
template<class T >
```

```
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 >
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

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

3.9.1 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.

3.9.2 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 rtsha_block to initialize the MemoryBlock with.
--------------	---

3.9.3 Member Function Documentation

getAllocAddress()

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

Retrieves the underlying [rtsha_block](#) pointer.

Returns

A pointer to the associated [rtsha_block](#).

getFreeBlockAddress()

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

Retrieves the address of the free block.

Returns

The address of the free block.

getNextBlock()

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

Retrieves the previous block relative to the current block.

Returns

A pointer to the previous [rtsha_block](#).

getSize()

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

Gets the size of the current block.

Returns

The size of the block.

hasPrev()

```
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()

```
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()

```
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()

```
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 MemoryBlock instance to assign from.
------------	--

Returns

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

setPrev()

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

Sets the previous block for the current block.

Parameters

<i>prev</i>	The previous MemoryBlock to set.
-------------	--

setSize()

```
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 (
    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:

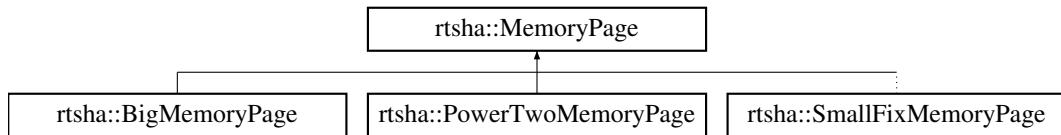
- MemoryBlock.h

3.10 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 (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 (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 (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 * getFreeMap () const**

- `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 (size_t val)`
Increments the page position by the given value.
- `void decPosition (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 (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.

3.10.1 Detailed Description

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

3.10.2 Constructor & Destructor Documentation

MemoryPage()

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

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

Parameters

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

3.10.3 Member Function Documentation**allocate_block()**

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

Pure virtual function to allocate a block of memory.

Parameters

<i>size</i>	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 (
    size_t size )
```

Allocates a block of memory at the current position.

Parameters

<i>size</i>	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

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

Returns

True if the block exists, otherwise false.

decPosition()

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

Decreases the page position by the given value.

Parameters

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

fitOnPage()

```
bool rtsha::MemoryPage::fitOnPage (
    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.

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 (
    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

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

setFreeBlockAllocatorsAddress()

```
void rtsha::MemoryPage::setFreeBlockAllocatorsAddress (
    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:

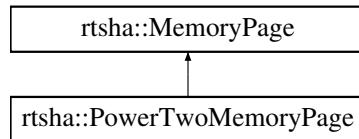
- `MemoryPage.h`

3.11 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 (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 (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 (size_t size)**
Allocates a block of memory at the current position.
- **void incFreeBlocks ()**
Increments the count of free blocks.

Protected Member Functions

- void `splitBlockPowerTwo (MemoryBlock &block, size_t end_size)`
Splits a memory block into two blocks 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 (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 * 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 (size_t val)`
Increments the page position by the given value.
- `void decPosition (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 (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.

3.11.1 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.

3.11.2 Constructor & Destructor Documentation

`PowerTwoMemoryPage()`

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

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

Parameters

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

3.11.3 Member Function Documentation

`allocate_block()`

```
rtsha::PowerTwoMemoryPage::allocate_block (
    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

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

Implements [rtsha::MemoryPage](#).

mergeLeft()

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

Merges the specified block with its left neighbor.

This function is typically used when freeing a block of memory to see if it can be coalesced with its left neighbor to create a larger block.

Parameters

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

mergeRight()

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

Merges the specified block with its right neighbor.

This function is typically used when freeing a block of memory to see if it can be coalesced with its right neighbor to create a larger block.

Parameters

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

splitBlockPowerTwo()

```
void rtsha::PowerTwoMemoryPage::splitBlockPowerTwo (
    MemoryBlock & block,
    size_t end_size ) [protected]
```

Splits a memory block into two blocks based on the specified size.

This function is used for power-of-two allocators where blocks can be efficiently split in half multiple times. After splitting, the original block is resized to 'end_size', and the remainder of the same size becomes a new block.

Parameters

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

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

- PowerTwoMemoryPage.h

3.12 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.

3.12.1 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

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

3.12.2 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.
-------------	--------------------------------------

3.12.3 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

3.13 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.

3.13.1 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.

3.13.2 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

3.14 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.

3.14.1 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

3.15 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.

3.15.1 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.
----------	---------------------------------------

3.15.2 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.
----------	------------------------------------

3.15.3 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:

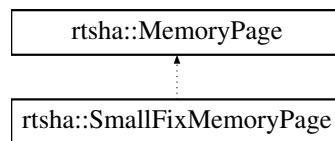
- FastPlusAllocator.h

3.16 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 (size_t size) final**
Allocates a memory block of fixed size.
- **virtual void free_block (MemoryBlock &block) final**
This function deallocates memory block.

3.16.1 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.

3.16.2 Constructor & Destructor Documentation

[SmallFixMemoryPage\(\)](#)

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

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

Parameters

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

3.16.3 Member Function Documentation

[allocate_block\(\)](#)

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

Allocates a memory block of fixed size.

Parameters

<code>size</code>	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_calloc(size_t nitems, size_t size);
00060
00061     rtsha_decl_nodiscard rtsha_decl_export void* rtsha_realloc(void* ptr, 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 BigMemoryPage.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryPage.h"
00004
00005
00006 namespace rtsha
00007 {
00008     using namespace std;
00009
00018     class BigMemoryPage : public MemoryPage
00019     {
00020         public:
00021             BigMemoryPage() = delete;
00024
00029             explicit BigMemoryPage(rtsha_page* page) : MemoryPage(page)
00030             {
00031             }
00032
00034             virtual ~BigMemoryPage()
00035             {
00036             }
00037
00045             virtual void* allocate_block(size_t size) final;
00046
00051             virtual void free_block(MemoryBlock& block) final;
00052
00056             void createInitialFreeBlocks();
00057
00058     protected:
00059
00065             void splitBlock(MemoryBlock& block, size_t size);
00066
00071             void mergeLeft(MemoryBlock& block);
00072
00077             void mergeRight(MemoryBlock& block);
00078     };
00079 }

```

4.3 errors.h

```
00001 #include "internal.h"
```

```

00002
00003 #pragma once
00004
00011 #define RTSHA_OK (0U)
00012
00014 #define RTSHA_ErrorInit (16U)
00015
00017 #define RTSHA_ErrorInitPageSize (32U)
00018
00020 #define RTSHA_ErrorInitOutOfHeap (33U)
00021
00023 #define RTSHA_OutOfMemory (64U)
00024
00026 #define RTSHA_NoPages (128U)
00027
00029 #define RTSHA_NoPage (129U)
00030
00032 #define RTSHA_NoFreePage (130U)
00033
00035 #define RTSHA_InvalidBlock (256U)
00036
00038 #define RTSHA_InvalidBlockDistance (257U)
00039
00041 #define RTSHA_InvalidNumberOfFreeBlocks (258U)
00042 // end of RTSHA_ERRORS group
00044

```

4.4 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
00031         RTSHAllocator() = default;
00032
00033
00042         template<class U>
00043         constexpr RTSHAllocator(const RTSHAllocator <U>&) noexcept {}
00044
00055         [[nodiscard]] T* allocate(std::size_t n)
00056         {
00057             if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
00058                 throw std::bad_array_new_length();
00059
00060             if (auto p = static_cast<T*>(rtsha_malloc(n * sizeof(T))))
00061             {
00062                 //report(p, n);
00063                 return p;
00064             }
00065
00066             throw std::bad_alloc();
00067         }
00068
00077         void deallocate(T* p, std::size_t n) noexcept
00078         {
00079             //report(p, n, 0);
00080             rtsha_free(p);
00081         }
00082     private:
00083
00093         void report(T* p, std::size_t n, bool alloc = true) const
00094         {
00095             std::cout << (alloc ? "Alloc: " : "Dealloc: ") << sizeof(T) * n
00096             << " bytes at " << std::hex << std::showbase
00097             << reinterpret_cast<void*>(p) << std::dec << '\n';
00098         }
00099     };
00100
00108     template<class T, class U>
00109     bool operator==(const RTSHAllocator <T>&, const RTSHAllocator <U>&) { return true; }
00110
00118     template<class T, class U>
00119     bool operator!=(const RTSHAllocator <T>&, const RTSHAllocator <U>&) { return false; }
00120 }

```

4.5 FreeList.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryBlock.h"
00004 #include "InternListAllocator.h"
00005 #include "forward_list"
00006
00007 namespace internal
00008 {
00009     using namespace std;
0010
0011     using flist = std::forward_list<size_t, InternListAllocator<size_t>>;
0012
0021     class alignas(sizeof(size_t)) FreeList
0022     {
0023         public:
0024
0026             FreeList() = delete;
0027
0032             explicit FreeList(rtsha_page* page);
0033
0035             ~FreeList()
0036             {
0037             }
0038
0044             void push(const size_t address);
0045
0051             size_t pop();
0052
0053         private:
0054             rtsha_page* _page;
0055             InternListAllocator<std::size_t>* _lallocator;
0056             flist* ptrLlist;
0057
0058         private:
0065             PREALLOC_MEMORY<size_t> _internal_list_storage = 0U;
0066
0068             PREALLOC_MEMORY<InternListAllocator<size_t>> _storage_allocator = 0U;
0069             PREALLOC_MEMORY<flist> _storage_list = 0U;
0070     };
0071 }
```

4.6 FreeMap.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryBlock.h"
00004 #include "InternMapAllocator.h"
00005 #include "map"
00006
00007 namespace internal
00008 {
00009     using namespace std;
0010     using namespace rtsha;
0011
0012     using mmap_allocator = InternMapAllocator<std::pair<const uint64_t, size_t>>;
0013
0014     using mmap = std::multimap<const uint64_t, size_t, std::less<const uint64_t>,
0015     internal::InternMapAllocator<std::pair<const uint64_t, size_t>>>;
0016
0025     class alignas(sizeof(size_t)) FreeMap
0026     {
0027         public:
0028
0030             FreeMap() = delete;
0031
0037             explicit FreeMap(rtsha_page* page);
0038
0040             ~FreeMap()
0041             {
0042             }
0043
0050             void insert(const uint64_t key, size_t block);
0051
0059             bool del(const uint64_t key, size_t block);
0060
0067             size_t find(const uint64_t key);
0068
0076             bool exists(const uint64_t key, size_t block);
0077
0083             size_t size() const;
0084 }
```

```

00085     private:
00086         rtsha_page*           _page;
00087         mmap_allocator*       _mallocator;
00088         mmap*                 _ptrMap;
00089
00090
00091     private:
00092
00094         PREALLOC_MEMORY <mmap_allocator>           _storage_allocator = 0U;
00095         PREALLOC_MEMORY<mmap>                      _storage_map = 0U;
00096     };
00097 }

```

4.7 Heap.h

```

00001 #pragma once
00002 #include <stdint.h>
00003 #include <array>
00004 #include "MemoryPage.h"
00005 #include "errors.h"
00006 #include "FreeList.h"
00007 #include "FreeMap.h"
00008
00009 namespace internal
00010 {
00011     using namespace std;
00012
00024     class HeapInternal
00025     {
00026     public:
00030         HeapInternal()
00031         {
00032             for (size_t i = 0; i < _pages.size(); i++)
00033             {
00034                 _pages[i] = nullptr;
00035             }
00036         }
00037
00041         ~HeapInternal()
00042         {
00043
00044         }
00045
00046
00059         FreeList* createFreeList(rtsha_page* page);
00060
00075         FreeMap* createFreeMap(rtsha_page* page);
00076
00077
00078     protected:
00079
00086         void init_small_fix_page(rtsha_page* page, size_t a_size);
00087
00101         void init_power_two_page(rtsha_page* page, size_t a_size, size_t max_objects, size_t
00102             min_block_size, size_t max_block_size);
00114         void init_big_block_page(rtsha_page* page, size_t a_size, size_t max_objects);
00115
00116     protected:
00117
00123         std::array<rtsha_page*, MAX_PAGES> _pages;
00124
00128         size_t      _number_pages = 0U;
00129
00133         address_t   _heap_start = NULL;
00134
00138         size_t      _heap_size = 0U;
00139
00145         address_t   _heap_current_position = NULL;
00146
00150         address_t   _heap_top = NULL;
00151
00155         bool        _heap_init = false;
00156
00162         uint32_t    _last_heap_error = RTSHA_OK;
00163
00164
00171         PREALLOC_MEMORY<FreeList, (MAX_SMALL_PAGES + MAX_BIG_PAGES)> _storage_free_lists = 0U;
00172
00173
00180         PREALLOC_MEMORY<FreeMap, MAX_BIG_PAGES>      _storage_free_maps = 0U;
00181     };
00182 }
00183

```

```

00184 namespace rtsha
00185 {
00186     using namespace std;
00187     using namespace internal;
00188
00194     class Heap : HeapInternal
00195     {
00196     public:
00197         Heap();
00201
00202
00203         ~Heap();
00208
00209
00218         bool init(void* start, size_t size);
00219
00220
00243         bool add_page(HeapCallbacksStruct* callbacks, rtsha_page_size_type size_type, size_t size,
00244             size_t max_objects = 0U, size_t min_block_size = 0U, size_t max_block_size = 0U);
00244
00250         size_t get_free_space() const;
00251
00257         rtsha_page* get_big_memorypage() const;
00258
00264         rtsha_page* get_block_page(address_t block_address);
00265
00266
00281         void* malloc(size_t size);
00282
00292         void free(void* ptr);
00293
00306         void* calloc(size_t nitems, size_t size);
00307
00327         void* realloc(void* ptr, size_t size);
00328
00346         void* memcpy(void* _Dst, void const* _Src, size_t _Size);
00347
00364         void* memset(void* _Dst, int _Val, size_t _Size);
00365
00366
00376         rtsha_page_size_type get_ideal_page(size_t size) const;
00377
00391         rtsha_page* select_page(rtsha_page_size_type ideal_page, size_t size, bool no_big = false)
00392             const;
00392
00393     };
00394 }
00395

```

4.8 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

```

4.9 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 # include <cpuid.h>

```

```
00012 # include <emmintrin.h>
00013 #endif
00014
00015
00016
00049 #define MULTITHREADING_SUPPORT
00050
00051
00052 #define MAX_BLOCKS_PER_PAGE UINT32_MAX
00053
00054 #define MAX_SMALL_PAGES      32U
00055 #define MAX_BIG_PAGES        2U
00056 #define MAX_PAGES           (MAX_SMALL_PAGES+MAX_BIG_PAGES)
00057
00058 #if defined _WIN64 || defined _ARM64
00059 #define RTSHA_BLOCK_HEADER_SIZE (2 * sizeof(size_t))
00060 #define MIN_BLOCK_SIZE_FOR_SPLIT 56U /*todo*/
00061 #else
00062 #define RTSHA_LIST_ITEM_SIZE (2 * sizeof(size_t))
00063 #define MIN_BLOCK_SIZE_FOR_SPLIT 512U /*todo*/
00064 #endif
00065
00066 #ifdef __cplusplus
00067 #if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
00068 #define rtsha_attr_noexcept noexcept
00069 #else
00070 #define rtsha_attr_noexcept throw()
00071 #endif
00072 #else
00073 #define rtsha_attr_noexcept
00074 #endif
00075
00076
00077 #if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
00078 #define rtsha_decl_nodiscard [[nodiscard]]
00079 #elif (defined(__GNUC__) && (__GNUC__ >= 4)) || defined(__clang__)
00080 #define rtsha_decl_nodiscard __attribute__((warn_unused_result))
00081 #elif defined(_HAS_NODISCARD)
00082 #else
00083 #define rtsha_decl_nodiscard
00084 #endif
00085
00086
00087
00088
00089
00090 #define RTSHA_ALIGNMENT      sizeof(size_t) /*4U or 8U*/
00091
00092 #define is_bit(val,n) ( (val >> n) & 0x01U )
00093
00094 #ifndef rtsha_assert
00095     #define rtsha_assert(x) assert(x)
00096 #endif
00097
00098
00099 #if _WIN32 || _WIN64
00100     #if defined(WIN64) || defined(__amd64__)
00101         #define ENV64BIT
00102     #else
00103         #define ENV32BIT
00104     #endif
00105 #else
00106     #if __GNUC__
00107         #if __x86_64__ || __ppc64__
00108             #define ENV64BIT
00109         #else
00110             #define ENV32BIT
00111         #endif
00112     #endif
00113 #endif
00114
00115
00116 #ifdef __arm__ //ARM architecture
00117 #define ARCH_ARM
00118 #endif
00119
00120 #ifdef __aarch64__ //ARM 64-bit
00121 #define ARCH_ARM
00122 #define ARCH_ARM_64
00123 #define ARCH_64BIT
00124 #endif
00125
00126
00127 namespace internal
00128 {
00129     using address_t = uintptr_t;
00130 }
```

```

00131 #if defined(WIN64) // The _BitScanReverse64 intrinsic is only available for 64 bit builds because it
00132     depends on x64
00133     inline uint64_t ExpandToPowerOf2(uint64_t Value)
00134     {
00135         unsigned long Index;
00136         _BitScanReverse64(&Index, Value - 1);
00137         return (1ULL << (Index + 1));
00138     }
00139 #else
00140     inline uint32_t ExpandToPowerOf2(uint32_t Value)
00141     {
00142         unsigned long Index;
00143         _BitScanReverse(&Index, Value - 1);
00144         return (1U << (Index + 1));
00145     }
00146 #endif
00147
00148
00149     template<typename T, size_t n = 1U>
00150     struct alignas(sizeof(size_t)) PREALLOC_MEMORY
00151     {
00152         public:
00153             PREALLOC_MEMORY()
00154             {
00155             }
00156
00157             PREALLOC_MEMORY(uint8_t init)
00158             {
00159                 std::memset(_memory, init, sizeof(_memory));
00160             }
00161
00162         public:
00163             inline void* get_ptr()
00164             {
00165                 return (void*)_memory;
00166             }
00167
00168             inline void* get_next_ptr()
00169             {
00170                 if (_count < n)
00171                 {
00172                     void* ret = (void*)(_memory + _count * sizeof(T));
00173                     _count++;
00174                     return ret;
00175                 }
00176                 return nullptr;
00177             }
00178         private:
00179             uint8_t _memory[n * sizeof(T)];
00180             size_t _count = 0U;
00181     };
00182
00183     static inline uintptr_t rtsha_align(uintptr_t ptr)
00184     {
00185         static_assert(RTSHA_ALIGMENT > 0);
00186
00187         uintptr_t mask = RTSHA_ALIGMENT - 1U;
00188
00189         if ((RTSHA_ALIGMENT & mask) == 0U)
00190         {
00191             return ((ptr + mask) & ~mask);
00192         }
00193         return (((ptr + mask) / RTSHA_ALIGMENT) * RTSHA_ALIGMENT);
00194     }
00195 }
00196 }
```

4.10 InternListAllocator.h

```

00001 #pragma once
00002 #include "MemoryPage.h"
00003 #include <cstdlib>
00004 #include <new>
00005 #include <iostream>
00006
00007 namespace internal
00008 {
00009     using namespace rtsha;
00010
00011     template<class T>
```

```

00021     struct InternListAllocator
00022     {
00026         typedef T value_type;
00027
00034         explicit InternListAllocator(rtsha_page* page, size_t* _ptrSmallStorage)
00035             : _page(page),
00036                 _allocated_intern(0U),
00038                 _ptrInternalSmallStorage(_ptrSmallStorage)
00039         {}
00040     }
00041
00050     template<class U>
00051     constexpr InternListAllocator(const InternListAllocator <U>& rhs) noexcept
00052     {
00053         this->_page = rhs._page;
00054         this->_allocated_intern = rhs._allocated_intern;
00055         this->_ptrInternalSmallStorage = rhs._ptrInternalSmallStorage;
00056     }
00057
00066     [[nodiscard]] T* allocate(std::size_t n) noexcept
00067     {
00068         /*max. 1 block*/
00069         if (n != 1U)
00070         {
00071             return nullptr;
00072         }
00073         if ((-_allocated_intern == 0U) && (_page->lastFreeBlockAddress == 0U))
00074         {
00075             if (n > 1U)
00076             {
00077                 return nullptr;
00078             }
00079             _allocated_intern++;
00080             auto p = static_cast<T*>((void*)-_ptrInternalSmallStorage);
00081             return p;
00082         }
00083         auto p = static_cast<T*>((void*) _page->lastFreeBlockAddress);
00084         return p;
00085     }
00086
00093     void deallocate(T* /*p*/, std::size_t /*n*/) noexcept
00094     {
00095     }
00096
00097     rtsha_page* _page;
00098     size_t _allocated_intern = 0U;
00099     size_t* _ptrInternalSmallStorage = NULL;
00100
00101 private:
00102     void report(T* p, std::size_t n, bool alloc = true) const
00111     {
00112         std::cout << (alloc ? "LAlloc: " : "LDealloc: ") << sizeof(T) * n
00113         << " bytes at " << std::hex << std::showbase
00114         << reinterpret_cast<void*>(p) << std::dec << '\n';
00115     };
00116
00125     template<class T, class U>
00126     bool operator==(const InternListAllocator <T>&, const InternListAllocator <U>&) { return true; }
00127
00135     template<class T, class U>
00136     bool operator!=(const InternListAllocator <T>&, const InternListAllocator <U>&) { return false; }
00137 }
00138

```

4.11 InternMapAllocator.h

```

00001 #pragma once
00002 #include "SmallFixMemoryPage.h"
00003 #include <cstdlib>
00004 #include <new>
00005 #include <iostream>
00006
00007 namespace internal
00008 {
00009     #if defined _WIN64 || defined _ARM64
00010         #define INTERNAL_MAP_STORAGE_SIZE 64U
00011     #else
00012         #define INTERNAL_MAP_STORAGE_SIZE 32U
00013     #endif
00014
00015     using namespace rtsha;

```

```

00016
00025     template<class T>
00026     struct InternMapAllocator
00027     {
00031         typedef T value_type;
00032
00038         InternMapAllocator(rtsha_page* page)
00039             :_page(page)
00040         {
00041         }
00042
00054         template<class U>
00055         constexpr InternMapAllocator(const InternMapAllocator <U>& rhs) noexcept
00056         {
00057             this->_page = rhs._page;
00058         }
00059
00073         [[nodiscard]] T* allocate(std::size_t n) noexcept
00074         {
00075             if (_page->map_page != nullptr)
00076             {
00077                 SmallFixMemoryPage map_page(_page->map_page);
00078                 if ((size_t)_page->map_page->flags >= (n * sizeof(T)))
00079                 {
00080                     auto p =
00081                     reinterpret_cast<T*>(map_page.allocate_block((size_t)_page->map_page->flags));
00082                     if (p != nullptr)
00083                     {
00084                         //report(p, n, 1);
00085                         return p;
00086                     }
00087                 }
00088                 return NULL;
00089             }
00090
00098         void deallocate(T*p, std::size_t /*n*/) noexcept
00099         {
00100             if (_page->map_page != nullptr)
00101             {
00102                 SmallFixMemoryPage map_page(_page->map_page);
00103
00104                 size_t address = reinterpret_cast<size_t>(p);
00105                 address -= (2U * sizeof(size_t)); /*skip size and pointer to prev*/
00106
00107                 MemoryBlock block((rtsha_block*)(void*)address);
00108
00109                 if (block.isValid())
00110                 {
00111                     map_page.free_block(block);
00112                     //report(p, n, 0);
00113                 }
00114             }
00115         }
00117         rtsha_page* _page;
00118
00119     private:
00120         void report(T* p, std::size_t n, bool alloc = true) const
00131         {
00132             std::cout << (alloc ? "MAlloc: " : "MDalloc: ") << sizeof(T) * n
00133             << " bytes at " << std::hex << std::showbase
00134             << reinterpret_cast<void*>(p) << std::dec << '\n';
00135         }
00136     };
00137
00145     template<class T, class U>
00146     bool operator==(const InternMapAllocator <T>&, const InternMapAllocator <U>&) { return true; }
00147
00155     template<class T, class U>
00156     bool operator!=(const InternMapAllocator <T>&, const InternMapAllocator <U>&) { return false; }
00157
00158 }

```

4.12 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
00017     struct rtsha_block
00018     {
00020         rtsha_block()
00021             :size(0U)
00022             ,prev(NULL)
00023         {
00024         }
00025
00026         size_t           size;
00029         rtsha_block* prev;
00030     };
00031
00039     class MemoryBlock
00040     {
00041     public:
00042         MemoryBlock() = delete;
00045
00050         explicit MemoryBlock(rtsha_block* block) : _block(block)
00051         {
00052         }
00053
00055         ~MemoryBlock()
00056         {
00057         }
00058
00066         MemoryBlock& operator = (const MemoryBlock& rhs)
00067         {
00068             this->_block = rhs._block;
00069             return *this;
00070         }
00071
00072
00080         void splitt(size_t new_size, bool last);
00081
00087         void splitt_22();
00088
00092         void merge_left();
00093
00097         void merge_right();
00098
00099
00103         inline void setAllocated()
00104         {
00105             _block->size = (_block->size >> 1U) << 1U;
00106         }
00107
00111         inline void setFree()
00112         {
00113             _block->size = (_block->size | 1U);
00114         }
00115
00119         inline void setLast()
00120         {
00121             _block->size = (_block->size | 2U);
00122         }
00123
00127         inline void clearIsLast()
00128         {
00129             _block->size &= ~(1UL << 1U);
00130         }
00131
00136         inline rtsha_block* getBlock() const
00137         {
00138             return _block;
00139         }
00140
00145         inline void* getAllocAddress() const
00146         {
00147             return reinterpret_cast<void*>((size_t)_block + 2U * sizeof(size_t));
00148         }
00149
00154         inline size_t getSize() const
00155         {
00156             return (_block->size >> 2U) << 2U;
00157         }
00158
00163         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 -
```

```

        sizeof(size_t));
00171     }           return (*ptrSize2 == size);
00172     }
00173   }
00174   return false;
00175 }
00176
00177 inline void setSize( size_t size )
00178 {
00179   if (size > sizeof(size_t))
00180   {
00181     bool free = isFree();
00182     bool last = isLast();
00183     _block->size = size;
00184     if (free)
00185     {
00186       setFree();
00187     }
00188     if (last)
00189     {
00190       setLast();
00191     }
00192     size_t* ptrSize2 = reinterpret_cast<size_t*>((size_t)_block + size - sizeof(size_t));
00193     *ptrSize2 = size;
00194   }
00195   else
00196   {
00197     _block->size = 0U;
00198   }
00199 }
00200
00201 inline size_t getFreeBlockAddress() const
00202 {
00203   return ((size_t)_block + 2U * sizeof(size_t));
00204 }
00205
00206 inline void setPrev(const MemoryBlock& prev)
00207 {
00208   if (prev.isValid())
00209   {
00210     _block->prev = prev.getBlock();
00211   }
00212   else
00213   {
00214     _block->prev = NULL;
00215   }
00216 }
00217
00218 inline void setAsFirst()
00219 {
00220   _block->prev = NULL;
00221 }
00222
00223 inline bool isFree()
00224 {
00225   return is_bit(_block->size, 0U);
00226 }
00227
00228 inline bool isLast()
00229 {
00230   return is_bit(_block->size, 1U);
00231 }
00232
00233 inline bool hasPrev()
00234 {
00235   return (_block->prev != NULL);
00236 }
00237
00238 inline rtsha_block* getNextBlock() const
00239 {
00240   return reinterpret_cast<rtsha_block*>((size_t)_block + this->getSize());
00241 }
00242
00243 inline rtsha_block* getPrev() const
00244 {
00245   return _block->prev;
00246 }
00247
00248 inline void prepare()
00249 {
00250   _block->prev = NULL;
00251   _block->size = 0;
00252 }
00253
00254 private:
00255   rtsha_block* _block;
00256 };

```

```
00302 }
```

4.13 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(size_t size) = 0;
00116 }
```

```

00121     virtual void free_block(MemoryBlock& block) = 0;
00122
00128     void* allocate_block_at_current_pos(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         #endif
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         #endif
00176     }
00182     inline void reportError(uint32_t error)
00183     {
00184         if ((nullptr != this->_page->callbacks) && (nullptr !=
00185             this->_page->callbacks->ptrErrorFunction))
00186         {
00187             this->_page->callbacks->ptrErrorFunction(error);
00188         }
00189     }
00196     inline void setFreeBlockAllocatorsAddress(size_t address)
00197     {
00198         _page->lastFreeBlockAddress = address;
00199     }
00200
00206     inline rtsha_page_size_type getPageType() const
00207     {
00208         return (rtsha_page_size_type) _page->flags;
00209     }
00216     inline void* getFreeList() const
00217     {
00218         return reinterpret_cast<void*>(_page->ptr_list_map);
00219     }
00220
00226     inline void* getFreeMap() const
00227     {
00228         return reinterpret_cast<void*>(_page->ptr_list_map);
00229     }
00230
00236     inline size_t getFreeBlocks() const
00237     {
00238         if (_page != nullptr)
00239         {
00240             return _page->free_blocks;
00241         }
00242         return 0U;
00243     }
00244
00250     inline size_t getMinBlockSize() const
00251     {
00252         if (_page != nullptr)
00253         {
00254             return _page->min_block_size;
00255         }
00256         return 0U;
00257     }
00264     inline address_t getPosition() const
00265     {
00266         if (_page != nullptr)
00267         {

```

```
00268         return _page->position;
00269     }
00270     return 0U;
00271 }
00272
00273 inline void setPosition(address_t pos)
00274 {
00275     if (_page != nullptr)
00276     {
00277         _page->position = pos;
00278     }
00279 }
00280
00281 inline void incPosition(size_t val)
00282 {
00283     if (_page != nullptr)
00284     {
00285         _page->position += val;
00286     }
00287 }
00288
00289 inline void decPosition(size_t val)
00290 {
00291     if (_page != nullptr)
00292     {
00293         if (_page->position >= val)
00294         {
00295             _page->position -= val;
00296         }
00297     }
00298 }
00299
00300 inline void decFreeBlocks()
00301 {
00302     if (_page != nullptr) && (_page->free_blocks > 0U)
00303     {
00304         _page->free_blocks--;
00305     }
00306 }
00307
00308 inline address_t getEndPosition() const
00309 {
00310     return _page->end_position;
00311 }
00312
00313 inline address_t getStartPosition() const
00314 {
00315     return _page->start_position;
00316 }
00317
00318 inline bool fitOnPage(size_t size) const
00319 {
00320     if ((_page->position + size) < (_page->end_position))
00321     {
00322         if (_page->start_map_data == 0U)
00323         {
00324             return true;
00325         }
00326         else
00327         {
00328             if ((_page->position + size) < _page->start_map_data)
00329             {
00330                 return true;
00331             }
00332         }
00333     }
00334     return false;
00335 }
00336
00337 inline bool hasLastBlock() const
00338 {
00339     return (_page->last_block != nullptr);
00340 }
00341
00342 inline bool isLastPageBlock(MemoryBlock& block) const
00343 {
00344     if (this->getPosition() == ((size_t)block.getBlock() + block.getSize()))
00345     {
00346         return (block.getBlock() == _page->last_block);
00347     }
00348     return false;
00349 }
00350
00351 inline rtsha_block* getLastBlock() const
00352 {
00353     return _page->last_block;
00354 }
```

```
00405     inline void setLastBlock(const MemoryBlock& block)
00411 {
00412     _page->last_block = block.getBlock();
00413 }
00414
00415     rtsha_page* _page;
00416
00417 };
00422 }
00423
```

4.14 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
00033     PowerTwoMemoryPage() = delete;
00034
00039     PowerTwoMemoryPage(rtsha_page* page) : MemoryPage(page)
00040     {
00041     }
00042
00044     virtual ~PowerTwoMemoryPage()
00045     {
00046     }
00047
00055     virtual void* allocate_block(size_t size) final;
00056
00064     virtual void free_block(MemoryBlock& block) final;
00065
00070     void createInitialFreeBlocks();
00071
00072     protected:
00073
00084     void splitBlockPowerTwo(MemoryBlock& block, size_t end_size);
00085
00094     void mergeLeft(MemoryBlock& block);
00095
00104     void mergeRight(MemoryBlock& block);
00105
00106 };
00107 }
```

4.15 SmallFixMemoryPage.h

```
00001 #pragma once
00002 #include <stdint.h>
00003 #include "MemoryPage.h"
00004
00005 namespace rtsha
00006 {
00007     using namespace std;
00008
00009
00032     class SmallFixMemoryPage : MemoryPage
00033     {
00034         public:
00035
00037     SmallFixMemoryPage() = delete;
00038
00043     explicit SmallFixMemoryPage(rtsha_page* page) : MemoryPage(page)
00044     {
00045     }
00046
00048     virtual ~SmallFixMemoryPage()
00049     {
00050     }
00051
00059     virtual void* allocate_block(size_t size) final;
00060
```

```
00068     virtual void free_block(MemoryBlock& block) final;
00069 };
00070 }
```


Index

_heap_current_position
 internal::HeapInternal, 22

_last_heap_error
 internal::HeapInternal, 22

_pages
 internal::HeapInternal, 22

_storage_free_lists
 internal::HeapInternal, 22

_storage_free_maps
 internal::HeapInternal, 22

add_page
 rtsha::Heap, 13

allocate
 internal::InternListAllocator< T >, 24
 internal::InternMapAllocator< T >, 26
 rtsha::RTSHMemoryPage, 50

allocate_block
 rtsha::BigMemoryPage, 6
 rtsha::MemoryPage, 35
 rtsha::PowerTwoMemoryPage, 43
 rtsha::SmallFixMemoryPage, 52

allocate_block_at_current_pos
 rtsha::MemoryPage, 35

allocator.h, 53

BigMemoryPage
 rtsha::BigMemoryPage, 6

BigMemoryPage.h, 54

calloc
 rtsha::Heap, 13

checkBlock
 rtsha::MemoryPage, 35

createFreeList
 internal::HeapInternal, 20

createFreeMap
 internal::HeapInternal, 20

createInitialFreeBlocks
 rtsha::PowerTwoMemoryPage, 44

deallocate
 internal::InternListAllocator< T >, 25
 internal::InternMapAllocator< T >, 27
 rtsha::RTSHMemoryPage, 50

decPosition
 rtsha::MemoryPage, 36

del
 internal::FreeMap, 10

errors.h, 54

exists
 internal::FreeMap, 10

FastPlusAllocator.h, 55

find
 internal::FreeMap, 11

fitOnPage
 rtsha::MemoryPage, 36

free
 rtsha::Heap, 14

free_block
 rtsha::BigMemoryPage, 6
 rtsha::MemoryPage, 36
 rtsha::PowerTwoMemoryPage, 44
 rtsha::SmallFixMemoryPage, 53

FreeList
 internal::FreeList, 8

FreeList.h, 56

FreeMap
 internal::FreeMap, 10

FreeMap.h, 56

get_big_memorypage
 rtsha::Heap, 14

get_block_page
 rtsha::Heap, 14

get_free_space
 rtsha::Heap, 14

get_ideal_page
 rtsha::Heap, 15

get_next_ptr
 internal::PREALLOC_MEMORY< T, n >, 46

get_ptr
 internal::PREALLOC_MEMORY< T, n >, 46

getAllocAddress
 rtsha::MemoryBlock, 29

getBlock
 rtsha::MemoryBlock, 29

getEndPosition
 rtsha::MemoryPage, 36

getFreeBlockAddress
 rtsha::MemoryBlock, 29

getFreeBlocks
 rtsha::MemoryPage, 37

getFreeList
 rtsha::MemoryPage, 37

getFreeMap
 rtsha::MemoryPage, 37

getLastBlock
 rtsha::MemoryPage, 37

getMinBlockSize
 rtsha::MemoryPage, 37

getNextBlock
 rtsha::MemoryBlock, 29

getPageType
 rtsha::MemoryPage, 38

getPosition
 rtsha::MemoryPage, 38

getPrev
 rtsha::MemoryBlock, 30

getSize

rtsha::MemoryBlock, 30
 getStartPosition
 rtsha::MemoryPage, 38

 hasLastBlock
 rtsha::MemoryPage, 38
 hasPrev
 rtsha::MemoryBlock, 30
 Heap.h, 57
 HeapCallbacks.h, 58
 HeapCallbacksStruct, 18

 incFreeBlocks
 rtsha::MemoryPage, 38
 incPosition
 rtsha::MemoryPage, 39
 init
 rtsha::Heap, 15
 init_big_block_page
 internal::HeapInternal, 20
 init_power_two_page
 internal::HeapInternal, 21
 init_small_fix_page
 internal::HeapInternal, 21
 insert
 internal::FreeMap, 11
 internal.h, 58
 internal::FreeList, 8
 FreeList, 8
 pop, 8
 push, 9
 internal::FreeMap, 9
 del, 10
 exists, 10
 find, 11
 FreeMap, 10
 insert, 11
 size, 11
 internal::HeapInternal, 18
 _heap_current_position, 22
 _last_heap_error, 22
 _pages, 22
 _storage_free_lists, 22
 _storage_free_maps, 22
 createFreeList, 20
 createFreeMap, 20
 init_big_block_page, 20
 init_power_two_page, 21
 init_small_fix_page, 21
 internal::InternListAllocator< T >, 23
 allocate, 24
 deallocate, 25
 InternListAllocator, 24
 internal::InternMapAllocator< T >, 25
 allocate, 26
 deallocate, 27
 InternMapAllocator, 26
 internal::PREALLOC_MEMORY< T, n >, 45
 get_next_ptr, 46

 get_ptr, 46
 PREALLOC_MEMORY, 46
 InternListAllocator
 internal::InternListAllocator< T >, 24
 InternListAllocator.h, 60
 InternMapAllocator
 internal::InternMapAllocator< T >, 26
 InternMapAllocator.h, 61
 isFree
 rtsha::MemoryBlock, 30
 isLast
 rtsha::MemoryBlock, 30
 isLastPageBlock
 rtsha::MemoryPage, 39
 isValid
 rtsha::MemoryBlock, 31
 lock
 rtsha::MemoryPage, 39

 malloc
 rtsha::Heap, 15
 memcpy
 rtsha::Heap, 16
 MemoryBlock
 rtsha::MemoryBlock, 29
 MemoryBlock.h, 62
 MemoryPage
 rtsha::MemoryPage, 34
 MemoryPage.h, 65
 memset
 rtsha::Heap, 16
 mergeLeft
 rtsha::BigMemoryPage, 7
 rtsha::PowerTwoMemoryPage, 44
 mergeRight
 rtsha::BigMemoryPage, 7
 rtsha::PowerTwoMemoryPage, 44

 operator=
 rtsha::MemoryBlock, 31

 pop
 internal::FreeList, 8
 PowerTwoMemoryPage
 rtsha::PowerTwoMemoryPage, 43
 PowerTwoMemoryPage.h, 68
 PREALLOC_MEMORY
 internal::PREALLOC_MEMORY< T, n >, 46
 push
 internal::FreeList, 9

 Real Time Safety Heap Allocator, 2
 realloc
 rtsha::Heap, 17
 reportError
 rtsha::MemoryPage, 39
 RTSHA Error Codes, 2
 rtsha::BigMemoryPage, 3

allocate_block, 6
BigMemoryPage, 6
free_block, 6
mergeLeft, 7
mergeRight, 7
splitBlock, 7
rtsha::Heap, 12
add_page, 13
calloc, 13
free, 14
get_big_memorypage, 14
get_block_page, 14
get_free_space, 14
get_ideal_page, 15
init, 15
malloc, 15
memcpy, 16
memset, 16
realloc, 17
select_page, 17
rtsha::MemoryBlock, 27
getAllocAddress, 29
getBlock, 29
getFreeBlockAddress, 29
getNextBlock, 29
getPrev, 30
getSize, 30
hasPrev, 30
isFree, 30
isLast, 30
isValid, 31
MemoryBlock, 29
operator=, 31
setPrev, 31
setSize, 32
splitt, 32
splitt_22, 32
rtsha::MemoryPage, 33
allocate_block, 35
allocate_block_at_current_pos, 35
checkBlock, 35
decPosition, 36
fitOnPage, 36
free_block, 36
getEndPosition, 36
getFreeBlocks, 37
getFreeList, 37
getFreeMap, 37
getLastBlock, 37
getMinBlockSize, 37
getPageType, 38
getPosition, 38
getStartPosition, 38
hasLastBlock, 38
incFreeBlocks, 38
incPosition, 39
isLastPageBlock, 39
lock, 39
MemoryPage, 34
reportError, 39
setFreeBlockAllocatorsAddress, 40
setLastBlock, 40
setPosition, 40
unlock, 40
rtsha::PowerTwoMemoryPage, 41
allocate_block, 43
createInitialFreeBlocks, 44
free_block, 44
mergeLeft, 44
mergeRight, 44
PowerTwoMemoryPage, 43
splitBlockPowerTwo, 45
rtsha::rtsha_block, 47
size, 47
rtsha::rtsha_page, 48
rtsha::RTSHAllocator< T >, 49
allocate, 50
deallocate, 50
RTSHAllocator, 49
rtsha::SmallFixMemoryPage, 51
allocate_block, 52
free_block, 53
SmallFixMemoryPage, 52
RTSHAllocator
rtsha::RTSHAllocator< T >, 49
select_page
rtsha::Heap, 17
setFreeBlockAllocatorsAddress
rtsha::MemoryPage, 40
setLastBlock
rtsha::MemoryPage, 40
setPosition
rtsha::MemoryPage, 40
setPrev
rtsha::MemoryBlock, 31
setSize
rtsha::MemoryBlock, 32
size
internal::FreeMap, 11
rtsha::rtsha_block, 47
SmallFixMemoryPage
rtsha::SmallFixMemoryPage, 52
SmallFixMemoryPage.h, 68
splitBlock
rtsha::BigMemoryPage, 7
splitBlockPowerTwo
rtsha::PowerTwoMemoryPage, 45
splitt
rtsha::MemoryBlock, 32
splitt_22
rtsha::MemoryBlock, 32
unlock
rtsha::MemoryPage, 40