RTSHA

Generated by Doxygen 1.9.7

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 simplies 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 internaly by ¨Power Two Memory Page¨ and ¨Big Memory Page¨ algorithms.
2. Power Two Memory Pages This is a more complex system, which only allows blocks of sizes that are powers of two. This makes merging free blocks back together easier and reduces fragmentation. A specialised search algorithm is used for fast storage and retrieval of ordered information which are stored in space of 'freed blocks'. This is a fairly efficient method of allocating memory, particularly useful for systems where memory fragmentation is an important concern. The algorithm divides memory into partitions to try to minimize fragmentation and the 'Best Fit' algorithm searches the page to find the smallest block that is large enough to satisfy the allocation. Furthermore, this system is resistant to breakdowns due to its algorithmic approach to allocating and deallocating memory. The coalescing operation helps ensure that large contiguous blocks of memory can be reformed after they are freed, reducing the likelihood of fragmentation over time. Coalescing relies on having free blocks of the same size available, which is not always the case, and so this system does not completely eliminate fragmentation but rather aims to minimize it.
3. Big Memory Pages

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

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

2 Module Documentation

2.1 RTSHA Error Codes

Macros

- #define **RTSHA_OK** (0U)

  Represents a successful operation or status.
- #define **RTSHA_ErrorInit** (16U)

  Error code indicating an initialization error.
- #define **RTSHA_ErrorInitPageSize** (32U)

  Error code indicating an invalid page size during initialization.
- #define **RTSHA_ErrorInitOutOfHeap** (33U)

  Error code indicating an out-of-heap error during initialization.
- #define **RTSHA_OutOfMemory** (64U)

  Error code indicating the system has run out of memory.
- #define **RTSHA_NoPages** (128U)

  Error code indicating no pages are available.
- #define **RTSHA_NoPage** (129U)

  Error code indicating a specific page is not available.
- #define **RTSHA_NoFreePage** (130U)

  Error code indicating there is no free page available.
- #define **RTSHA_InvalidBlock** (256U)

  Error code indicating the memory block is invalid.
- #define **RTSHA_InvalidBlockDistance** (257U)

  Error code indicating an invalid distance between blocks.
- #define **RTSHA_InvalidNumberOfFreeBlocks** (258U)

  Error code indicating an invalid number of free blocks.

2.1.1 Detailed Description

These are the error codes used throughout the RTSHA system.

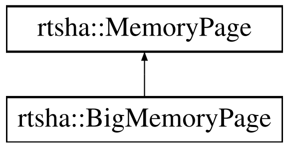
3 Class Documentation

3.1 rtsha::BigMemoryPage Class Reference

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

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

Inheritance diagram for rtsha::BigMemoryPage:



Public Member Functions

- **BigMemoryPage** ()=delete

  Default constructor is deleted to prevent default instantiation.
- BigMemoryPage (rtsha_page ∗page)

  Constructor that initializes the BigMemoryPage with a given page.
- virtual ∼**BigMemoryPage** ()

  Virtual destructor for the BigMemoryPage.
- virtual void ∗ allocate_block (const size_t &size) final

  Allocates a block of memory of the specified size.
- virtual void free_block (MemoryBlock &block) final

  Frees the specified memory block.
- void **createInitialFreeBlocks** ()

  Creates the initial first two free blocks within the page.

Public Member Functions inherited from rtsha::MemoryPage

- **MemoryPage** ()=delete

  Default constructor is deleted.
- MemoryPage (rtsha_page ∗page) noexcept

  Constructor that initializes the MemoryPage with a given page.
- virtual ∼**MemoryPage** ()

  Virtual destructor.
- bool checkBlock (size_t address)

  Check if a block exists at the given address and if the block is valid.
- virtual void ∗ allocate_block (const size_t &size)=0

  Pure virtual function to allocate a block of memory.
- virtual void free_block (MemoryBlock &block)=0

  Pure virtual function to free a block of memory.
- void ∗ allocate_block_at_current_pos (const size_t &size)

  Allocates a block of memory at the current position.
- void incFreeBlocks ()

  Increments the count of free blocks.

Protected Member Functions

- void splitBlock (MemoryBlock &block, size_t size)

    Splits a memory block based on the specified size.
- void mergeLeft (MemoryBlock &block)

    Merges the specified block with its left neighbor.
- void mergeRight (MemoryBlock &block)

    Merges the specified block with its right neighbor.

Protected Member Functions inherited from rtsha::MemoryPage

- void lock ()

    Locks the page for thread-safe operations.
- void unlock ()

    Unlocks the page after thread-safe operations.
- void reportError (uint32_t error)

    Reports an error using the specified error callback.
- void setFreeBlockAllocatorsAddress (const size_t &address)

    Sets the address of the last free block temporary. The addres will be used by InternListAllocator as storage for the elements of the 'std::forward_list'.
- rtsha_page_size_type getPageType () const

    Retrieves the type of the page.
- void ∗ getFreeList () const

    Gets the free list pointer of the page.
- void ∗ getFreeListArray () const

    Gets the free list array pointer of the page.
- void ∗ getFreeMap () const

    Gets the free map pointer of the page.
- size_t getFreeBlocks () const

    Retrieves the number of free blocks in the page.
- size_t getMinBlockSize () const

    Retrieves the number of free blocks in the page.
- address_t getPosition () const

    Gets the current page position.
- void setPosition (address_t pos)

    Sets the current page position.
- void incPosition (const size_t &val)

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

    Decreases the oage position by the given value.
- void **decFreeBlocks** ()

    Decreases the number of free blocks in the page.
- address_t getEndPosition () const

    Gets the end position of the page.
- address_t getStartPosition () const

    Gets the start position of the page.
- bool fitOnPage (const size_t &size) const

    Checks if a block of the specified size fits on the page.
- bool hasLastBlock () const

    Checks if the page 'last block' has ben 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

page	The rtsha_page structure to initialize the BigMemoryPage with.

3.1.3 Member Function Documentation

allocate_block()

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

Allocates a block of memory of the specified size.

Parameters

size	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

block	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

block	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

block	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

block	The block to be split.
size	The desired size of the block after splitting.

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

- BigMemoryPage.h

3.2 internal::FreeLinkedList Class Reference

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

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

Public Member Functions

- FreeLinkedList (rtsha_page ∗page)

  Constructs a FreeLinkedList with the given memory page.
- ∼**FreeLinkedList** ()

  Destructor for the FreeLinkedList.
- rtsha_attr_inline void push (const size_t &data)

  Adds a new node with the given memory block address to the head of the list.
- rtsha_attr_inline bool is_empty () const

  Checks if the list is empty.
- rtsha_attr_inline size_t pop ()

  Removes and retrieves a memory block address from the head of the list.
- rtsha_attr_inline bool delete_address (const size_t &address, void ∗block)

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

3.2.1 Detailed Description

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

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

3.2.2 Constructor & Destructor Documentation

FreeLinkedList()

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

Constructs a FreeLinkedList with the given memory page.

Parameters

page	The rtsha_page that this FreeLinkedList should manage.

3.2.3 Member Function Documentation

delete_address()

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

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

Parameters

address	Memory block address of the node to be deleted.
block	Pointer to the memory block associated with the address.

Returns

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

is_empty()

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

Checks if the list is empty.

Returns

Returns true if the list is empty, otherwise false.

pop()

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

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

Returns

The memory block address retrieved from the list.

push()

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

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

Parameters

data	The memory block address to be added.

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

- FreeLinkedList.h

3.3 internal::FreeList Class Reference

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

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

Public Member Functions

- **FreeList** ()=delete

    Default constructor is deleted to prevent default instantiation.
- FreeList (rtsha_page ∗page)

    Constructs a FreeList with the given memory page.
- ∼**FreeList** ()

    Destructor for the FreeList.
- rtsha_attr_inline void push (const size_t &address)

    Pushes a memory address onto the free list.
- rtsha_attr_inline size_t pop ()

    Pops and retrieves a memory address from the free list.
- rtsha_attr_inline bool **delete_address** (const size_t &address, void ∗block)

3.3.1 Detailed Description

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

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

3.3.2 Constructor & Destructor Documentation

FreeList()

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

Constructs a FreeList with the given memory page.

Parameters

page	The rtsha_page that this FreeList should manage.

3.3.3 Member Function Documentation

pop()

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

Pops and retrieves a memory address from the free list.

Returns

The memory address retrieved from the free list.

push()

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

Pushes a memory address onto the free list.

Parameters

address	The memory address to be added to the free list.

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

- FreeList.h

3.4 internal::FreeListArray Class Reference

Manages memory allocation of free blocks using a free list.

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

Public Member Functions

- **FreeListArray** ()=delete

  Default constructor is deleted to prevent default instantiation.
- FreeListArray (rtsha_page ∗page, size_t min_block_size, size_t page_size)

  Constructs a FreeListArray with the given memory page.
- ∼**FreeListArray** ()

  Destructor for the FreeListArray.

- rtsha_attr_inline void push (const size_t data, size_t size)

  Pushes a memory address onto the appropriate free list.
- rtsha_attr_inline size_t pop (size_t size)

  Pops and retrieves a memory address from the first appropriate free list.
- rtsha_attr_inline bool delete_address (const size_t &address, void ∗block, const size_t &size)

  Attempts to delete a memory address from the free lists.

3.4.1 Detailed Description

Manages memory allocation of free blocks using a free list.

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

3.4.2 Constructor & Destructor Documentation

FreeListArray()

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

Constructs a FreeListArray with the given memory page.

Parameters

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

3.4.3 Member Function Documentation

delete_address()

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

Attempts to delete a memory address from the free lists.

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

Parameters

address	The memory address to be deleted.
block	Pointer to the memory block associated with the address.
size	Size of the memory block.

Returns

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

pop()

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

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

Returns

The memory address retrieved from the free list.

push()

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

Pushes a memory address onto the appropriate free list.

Parameters

address	The memory address to be added to the free list.

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

- FreeListArray.h

3.5 internal::FreeMap Class Reference

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

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

Public Member Functions

- **FreeMap** ()=delete

  Default constructor is deleted to prevent default instantiation.
- FreeMap (rtsha_page ∗page)

  Constructs a FreeMap with the provided memory page.
- ∼**FreeMap** ()

  Destructor for the FreeMap.
- rtsha_attr_inline void insert (const uint64_t key, size_t block)

  Inserts a key-value pair into the multimap.

- rtsha_attr_inline bool del (const uint64_t key, size_t block)

  Deletes a key-value pair from the map based on the given key.
- rtsha_attr_inline size_t find (const uint64_t key)

  Finds the value associated with a given key.
- rtsha_attr_inline bool exists (const uint64_t key, size_t block)

  Checks if a given key-value pair exists in the map.
- size_t size () const

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

3.5.1 Detailed Description

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

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

3.5.2 Constructor & Destructor Documentation

FreeMap()

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

Constructs a FreeMap with the provided memory page.

Parameters

page	The rtsha_page that this FreeMap will manage.

3.5.3 Member Function Documentation

del()

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

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

Parameters

key	The key of the key-value pair to be deleted.
block	The associated value for the key.

Returns

True if the deletion was successful, otherwise false.

exists()

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

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

Parameters

key	The key to be checked.
block	The associated value for the key.

Returns

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

find()

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

Finds the value associated with a given key.

Parameters

key	The key to be looked up.

Returns

The value associated with the key.

insert()

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

Inserts a key-value pair into the multimap.

Parameters

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

size()

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

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

Returns

The size of the map.

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

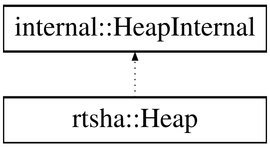
- FreeMap.h

3.6 rtsha::Heap Class Reference

This class encapsulates Heap in RTSHA.

`#include <Heap.h>`

Inheritance diagram for rtsha::Heap:



Public Member Functions

- **Heap** ()
  Standard constructor.
- ∼**Heap** ()
  Standard destructor.
- bool init (void ∗start, size_t size)
  This function initializes heap.
- bool add_page (HeapCallbacksStruct ∗callbacks, rtsha_page_size_type size_type, size_t size, size_t max↩_objects=0U, size_t min_block_size=0U, size_t max_block_size=0U)
  This function creates memory page and adds it to the heap. RTSHA supports more than one pages per heap.
- size_t get_free_space () const
  This function reurns deww space of the heap.
- rtsha_page ∗ get_big_memorypage () const
  This function returns first Big Memory Page page on the heap.
- rtsha_page ∗ get_block_page (address_t block_address)
  This function returns the page of allocated block.
- void ∗ malloc (size_t size)
  This function allocates block of memory on the heap.
- void free (void ∗ptr)
  This function deallocates memory block.
- void ∗ calloc (size_t nitems, size_t size)

This function allocates the block of memory on the heap and initializes it to zero.
- void ∗ realloc (void ∗ptr, size_t size)

    This function reallocates the block of memory on the heap.
- void ∗ memcpy (void ∗_Dst, void const ∗_Src, size_t _Size)

    This function copies the values of num bytes from the location pointed to by source directly to the memory block pointed to by destination.
- void ∗ memset (void ∗_Dst, int _Val, size_t _Size)

    This function sets values of num bytes from the location pointed to by _Dst to the specified value.
- rtsha_page_size_type get_ideal_page (size_t size) const

    This function returns ideal page type based on size criteria.
- rtsha_page ∗ select_page (rtsha_page_size_type ideal_page, size_t size, bool no_big=false) const

    This function gets page from the list of heap pages.

3.6.1 Detailed Description

This class encapsulates Heap in RTSHA.

3.6.2 Member Function Documentation

add_page()

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

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

This function takes a singleton instance of the heap.

Parameters

callbacks	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.
size	The size of the page.
size_type	The type of the memory page.
max_objects	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.
min_block_size	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'.
max_block_size	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 sucessfuly 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

nitems	An unsigned integral value which represents number of elements. If the size is zero, a null pointer will be returned.
size	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

ptr	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 reurns deww 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

size	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

start	The beginning of heap memory.
size	The size of heap memory.

Returns

Returns true when the heap has been sucessfuly 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

size	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

_Dst	Pointer to the destination.
_Src	Pointer to the source.
_Size	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

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

Returns

On success, a pointer todestination 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

ptr	Pointer to the memory allocated with 'rtsha_malloc' or 'rtsha_calloc'
size	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

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

Returns

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

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

- Heap.h

3.7 HeapCallbacksStruct Struct Reference

Represents a collection of callback functions for heap operations.

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

Public Attributes

- rtshLockPagePtr **ptrLockFunction**

  Function to lock a page.
- rtshLockPagePtr **ptrUnLockFunction**

  Function to unlock a page.
- rtshErrorPagePtr **ptrErrorFunction**

  Function to handle errors.

3.7.1 Detailed Description

Represents a collection of callback functions for heap operations.

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

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

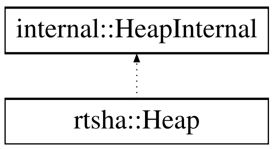
- HeapCallbacks.h

3.8 internal::HeapInternal Class Reference

This class encapsulates Heap.

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

Inheritance diagram for internal::HeapInternal:



Public Member Functions

- **HeapInternal** ()

  Standard constructor.
- ∼**HeapInternal** ()

  Standard destructor.
- FreeList ∗ createFreeList (rtsha_page ∗page)

  This function creates a 'Free List Object' that will be used for the management of the 'free blocks' Free List object is simple 'forward_list' or 'free linked list' used wit custom memory allocator The object is created in the predifined place on the stack using 'new placement' operator.
- FreeMap ∗ createFreeMap (rtsha_page ∗page)

  This function creates a 'Free Map Object' that will be used for the management of the 'free blocks' Free Map is C++ Standard Library 'multimap' object that implements a sorted associative container, allowing for fast retrieval of values based on key.
- FreeListArray ∗ createFreeListArray (rtsha_page ∗page, size_t page_size)

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

Protected Member Functions

- void init_small_fix_page (rtsha_page ∗page, size_t a_size)

  Initialize a small fixed-sized page.
- void init_power_two_page (rtsha_page ∗page, size_t a_size, size_t max_objects, size_t min_block_size, size_t max_block_size)

  Initialize a page for handling power-of-two sized memory blocks.
- void init_big_block_page (rtsha_page ∗page, size_t a_size, size_t max_objects)

  Initialize a page for handling large memory blocks.

Protected Attributes

- std::array< rtsha_page *, MAX_PAGES > _pages

    An array of pointers to manage the pages.
- size_t **_number_pages** = 0U

    Current number of pages managed by the heap.
- address_t **_heap_start** = 0U

    Starting address of the heap.
- size_t **_heap_size** = 0U

    Total size of the heap in bytes.
- address_t _heap_current_position = 0U

    Current position (or pointer) within the heap.
- address_t **_heap_top** = 0U

    The address marking the end of the heap.
- bool **_heap_init** = false

    Flag indicating if the heap has been initialized.
- uint32_t _last_heap_error = RTSHA_OK

    Last error code related to heap operations.
- bool _big_page_used

    Indicates that heap uses 'big memory' page.
- PREALLOC_MEMORY< FreeList,(MAX_SMALL_PAGES+MAX_BIG_PAGES)> _storage_free_lists = 0U

    Reserved storage on the stack for `FreeList` objects.
- PREALLOC_MEMORY< FreeListArray, MAX_POWER_TWO_PAGES > _storage_free_list_array = 0U

    Reserved storage on the stack for `FreeListArray` objects.
- PREALLOC_MEMORY< FreeMap, MAX_BIG_PAGES > _storage_free_maps = 0U

    Reserved storage on the stack for `FreeMap` objects.

3.8.1 Detailed Description

This class encapsulates Heap.

This class encapsulates HeapInternal in RTSHA.

3.8.2 Member Function Documentation

createFreeList()

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

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

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

Parameters

page	Pointer to page object's memory.

Returns

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

createFreeListArray()

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

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

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

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

Parameters

page	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 predifined place on the stack using 'new placement' operator.

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

Parameters

page	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

page	Pointer to the `rtsha_page` structure to be initialized.
a_size	Total size of the memory that the page will manage.
max_objects	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

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

page	Pointer to the `rtsha_page` structure to be initialized.
a_size	Size of each fixed-sized memory block in the page.

3.8.3 Member Data Documentation

_big_page_used

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

Indicates that heap uses 'big memory' page.

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

_heap_current_position

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

Current position (or pointer) within the heap.

Typically indicates where the next memory allocation will take place.

_last_heap_error

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

Last error code related to heap operations.

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

_pages

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

An array of pointers to manage the pages.

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

_storage_free_list_array

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

Reserved storage on the stack for `FreeListArray` objects.

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

_storage_free_lists

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

Reserved storage on the stack for `FreeList` objects.

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

_storage_free_maps

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

Reserved storage on the stack for `FreeMap` objects.

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

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

- Heap.h

3.9 internal::InternListAllocator< T > Class Template Reference

Custom allocator tailored for internal list management.

`#include <InternListAllocator.h>`

Public Types

- typedef T **value_type**

  Defines the type of elements managed by the allocator.

Public Member Functions

- InternListAllocator (rtsha_page ∗page, size_t ∗_ptrSmallStorage)

  Constructs the allocator with a given page and a pointer to a small storage.
- template< class U >
  constexpr InternListAllocator (const InternListAllocator< U > &rhs) noexcept

  Copy constructor allowing for type conversion.
- rtsha_attr_inline T ∗ allocate (std::size_t n) noexcept

  Allocates a memory for an array of `n` objects of type `T`. It is called from 'forward_list' every time when 'push' method is called Memory of the free block specified in lastFreeBlockAddress is used as a storage.
- rtsha_attr_inline void deallocate (T ∗, std::size_t) noexcept

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

Public Attributes

- rtsha_page * **_page**

  The memory page managed by the allocator.
- size_t **_allocated_intern** = 0U

  Counter to track a first few internal allocations.
- size_t * **_ptrInternalSmallStorage** = NULL

  Pointer to the small reserved storage.

3.9.1 Detailed Description

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

Custom allocator tailored for internal list management.

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

Template Parameters

T	The data type the allocator is responsible for.

3.9.2 Constructor & Destructor Documentation

InternListAllocator() [1/2]

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

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

Parameters

page	Pointer to the rtsha_page the allocator should use to allocate the memory for 'free list'.
_ptrSmallStorage	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

U	The data type of the other allocator.

Parameters

rhs	The other allocator to be copied from.

3.9.3 Member Function Documentation

allocate()

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

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

Parameters

n	Number of objects to allocate memory for.

Returns

Pointer to the allocated block of memory.

deallocate()

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

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

It is called from 'forward_list' every time when 'pop' method is called

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

- InternListAllocator.h

3.10 internal::InternMapAllocator< T > Struct Template Reference

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

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

Public Types

- typedef T **value_type**

  Defines the type of elements managed by the allocator.

Public Member Functions

- InternMapAllocator (rtsha_page ∗page)

  Construct a new Intern Map Allocator object.
- template< class U >
  constexpr InternMapAllocator (const InternMapAllocator< U > &rhs) noexcept

  Copy constructor.
- rtsha_attr_inline T ∗ allocate (std::size_t n) noexcept

  Allocate memory.
- rtsha_attr_inline void deallocate (T ∗p, std::size_t) noexcept

  Deallocate memory.

Public Attributes

- rtsha_page ∗ **_page**

  Memory page context.

3.10.1 Detailed Description

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

Allocator designed to handle internal memory allocations used with FreeMap classthat 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

T	The type of elements being allocated.

3.10.2 Constructor & Destructor Documentation

InternMapAllocator() [1/2]

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

Construct a new Intern Map Allocator object.

Parameters

page	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

U	The source type for the allocator.

Parameters

rhs	The source allocator.

3.10.3 Member Function Documentation

allocate()

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

Allocate memory.

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

Memory in predefined 'map_page' which is page of memory page will be used as storage. Allocator uses Small←FixMemoryPage together with 64 bytes blocks.

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

Parameters

n	Number of items of type `T` to allocate memory for.

Returns

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

deallocate()

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

Deallocate memory.

Release previously allocated memory back to the 'map_page' pool.

Parameters

p	Pointer to the memory to be deallocated.

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

- InternMapAllocator.h

3.11 rtsha::MemoryBlock Class Reference

Provides an abstraction for handling blocks of memory.

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

Public Member Functions

- **MemoryBlock** ()=delete

    Default constructor is deleted to prevent default instantiation.
- MemoryBlock (rtsha_block ∗block)

    Constructor that initializes the MemoryBlock with a given block.
- ∼**MemoryBlock** ()

    Destructor for the MemoryBlock.
- MemoryBlock & operator= (const MemoryBlock &rhs)

    Overloaded assignment operator for the MemoryBlock class.
- void splitt (const 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.
- rtsha_attr_inline void **setAllocated** ()

Marks the current block as allocated.

- rtsha_attr_inline void **setFree** ()

  Marks the current block as free.
- rtsha_attr_inline void **setLast** ()

  Marks the current block as the last block in the chain.
- rtsha_attr_inline void **clearIsLast** ()

  Clears the 'is last' status of the current block.
- rtsha_attr_inline rtsha_block ∗ getBlock () const

  Retrieves the underlying rtsha_block pointer.
- rtsha_attr_inline void ∗ getAllocAddress () const

  Retrieves the memory address allocated for the block.
- rtsha_attr_inline size_t getSize () const

  Gets the size of the current block.
- rtsha_attr_inline bool isValid () const

  Checks if the current MemoryBlock instance is valid.
- rtsha_attr_inline void setSize (size_t size)

  Sets the size of the current block.
- rtsha_attr_inline size_t getFreeBlockAddress () const

  Retrieves the address of the free block.
- rtsha_attr_inline void setPrev (const MemoryBlock &prev)

  Sets the previous block for the current block.
- rtsha_attr_inline void **setAsFirst** ()

  Sets the current block as the first block in a chain.
- rtsha_attr_inline bool isFree ()

  Checks if the current block is free.
- rtsha_attr_inline bool isLast ()

  Checks if the current block is the last block in the chain.
- rtsha_attr_inline bool hasPrev ()

  Checks if the current block has a predecessor.
- rtsha_attr_inline rtsha_block ∗ getNextBlock () const

  Retrieves the next block in memory relative to the current block.
- rtsha_attr_inline rtsha_block ∗ getPrev () const

  Retrieves the previous block relative to the current block.
- rtsha_attr_inline void **prepare** ()

  Prepares the current block for use by resetting its attributes in memory.

3.11.1 Detailed Description

Provides an abstraction for handling blocks of memory.

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

3.11.2 Constructor & Destructor Documentation

MemoryBlock()

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

Constructor that initializes the MemoryBlock with a given block.

Parameters

block	The rtsha_block to initialize the MemoryBlock with.

3.11.3 Member Function Documentation

getAllocAddress()

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

Retrieves the memory address allocated for the block.

Returns

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

getBlock()

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

Retrieves the underlying rtsha_block pointer.

Returns

A pointer to the associated rtsha_block.

getFreeBlockAddress()

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

Retrieves the address of the free block.

Returns

The address of the free block.

getNextBlock()

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

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

Returns

A pointer to the next rtsha_block in the chain.

getPrev()

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

Retrieves the previous block relative to the current block.

Returns

A pointer to the previous rtsha_block.

getSize()

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

Gets the size of the current block.

Returns

The size of the block.

hasPrev()

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

Checks if the current block has a predecessor.

Returns

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

isFree()

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

Checks if the current block is free.

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

Returns

True if the block is free, otherwise false.

isLast()

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

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

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

Returns

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

isValid()

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

Checks if the current MemoryBlock instance is valid.

Returns

True if the block is valid, otherwise false.

operator=()

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

Overloaded assignment operator for the MemoryBlock class.

This allows one MemoryBlock to be assigned to another, copying its underlying block reference.

Parameters

rhs	The right-hand side MemoryBlock instance to assign from.

Returns

A reference to the updated MemoryBlock.

setPrev()

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

Sets the previous block for the current block.

Parameters

prev	The previous MemoryBlock to set.

setSize()

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

Sets the size of the current block.

Parameters

size	The new size to set for the block.

splitt()

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

Splits the current block into two blocks.

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

Parameters

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

splitt_22()

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

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

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

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

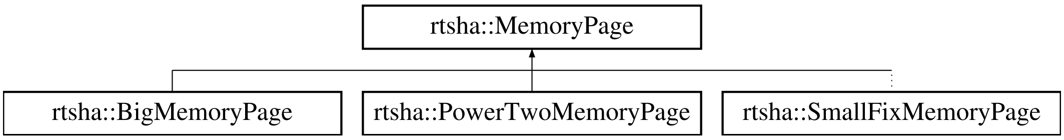
- MemoryBlock.h

3.12 rtsha::MemoryPage Class Reference

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

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

Inheritance diagram for rtsha::MemoryPage:



Public Member Functions

- **MemoryPage** ()=delete

  Default constructor is deleted.
- MemoryPage (rtsha_page ∗page) noexcept

  Constructor that initializes the MemoryPage with a given page.
- virtual ∼**MemoryPage** ()

  Virtual destructor.
- bool checkBlock (size_t address)

  Check if a block exists at the given address and if the block is valid.
- virtual void ∗ allocate_block (const size_t &size)=0

  Pure virtual function to allocate a block of memory.
- virtual void free_block (MemoryBlock &block)=0

  Pure virtual function to free a block of memory.
- void ∗ allocate_block_at_current_pos (const size_t &size)

  Allocates a block of memory at the current position.
- void incFreeBlocks ()

  Increments the count of free blocks.

Protected Member Functions

- void lock ()

  Locks the page for thread-safe operations.
- void unlock ()

  Unlocks the page after thread-safe operations.
- void reportError (uint32_t error)

  Reports an error using the specified error callback.
- void setFreeBlockAllocatorsAddress (const size_t &address)

  Sets the address of the last free block temporary. The addres will be used by InternListAllocator as storage for the elements of the 'std::forward_list'.
- rtsha_page_size_type getPageType () const

  Retrieves the type of the page.
- void ∗ getFreeList () const

  Gets the free list pointer of the page.
- void ∗ getFreeListArray () const

  Gets the free list array pointer of the page.
- void ∗ getFreeMap () const

  Gets the free map pointer of the page.
- size_t getFreeBlocks () const

  Retrieves the number of free blocks in the page.
- size_t getMinBlockSize () const

  Retrieves the number of free blocks in the page.
- address_t getPosition () const

  Gets the current page position.
- void setPosition (address_t pos)

  Sets the current page position.
- void incPosition (const size_t &val)

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

  Decreases the oage position by the given value.
- void **decFreeBlocks** ()

Decreases the number of free blocks in the page.
- address_t getEndPosition () const

  Gets the end position of the page.
- address_t getStartPosition () const

  Gets the start position of the page.
- bool fitOnPage (const size_t &size) const

  Checks if a block of the specified size fits on the page.
- bool hasLastBlock () const

  Checks if the page 'last block' has ben set.
- bool isLastPageBlock (MemoryBlock &block) const

  Determines if the provided block is the last block of the page.
- rtsha_block ∗ getLastBlock () const

  Gets the last block of the page.
- void setLastBlock (const MemoryBlock &block)

  Sets the last block of the page.

Protected Attributes

- rtsha_page ∗ **_page**

  Pointer to the page structure in memory.

3.12.1 Detailed Description

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

3.12.2 Constructor & Destructor Documentation

MemoryPage()

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

Constructor that initializes the MemoryPage with a given page.

Parameters

page	The pointer to page in memory

3.12.3 Member Function Documentation

allocate_block()

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

Pure virtual function to allocate a block of memory.

Parameters

size	Size of the block to allocate.

Returns

A pointer to the allocated block.

Implemented in rtsha::BigMemoryPage, rtsha::PowerTwoMemoryPage, and rtsha::SmallFixMemoryPage.

allocate_block_at_current_pos()

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

Allocates a block of memory at the current position.

Parameters

size	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

address	The address to check.

Returns

True if the block exists, otherwise false.

decPosition()

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

Decreases the oage position by the given value.

Parameters

val	The value to decrement the position by.

fitOnPage()

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

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

Parameters

size	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

block	The block of memory to be freed.

Implemented in rtsha::BigMemoryPage, rtsha::PowerTwoMemoryPage, and rtsha::SmallFixMemoryPage.

getEndPosition()

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

Gets the end position of the page.

Returns

The end position.

getFreeBlocks()

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

Retrieves the number of free blocks in the page.

Returns

The number of free blocks.

getFreeList()

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

Gets the free list pointer of the page.

Returns

A pointer to the free list.

getFreeListArray()

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

Gets the free list array pointer of the page.

Returns

A pointer to the free list array.

getFreeMap()

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

Gets the free map pointer of the page.

Returns

A pointer to the free map.

getLastBlock()

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

Gets the last block of the page.

Returns

A pointer to the last block.

getMinBlockSize()

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

Retrieves the number of free blocks in the page.

Returns

The number of free blocks.

getPageType()

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

Retrieves the type of the page.

Returns

The type of the page.

getPosition()

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

Gets the current page position.

Returns

The current position.

getStartPosition()

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

Gets the start position of the page.

Returns

The start position.

hasLastBlock()

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

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

Returns

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

incFreeBlocks()

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

Increments the count of free blocks.

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

incPosition()

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

Increments the page position by the given value.

Parameters

val	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

block	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

error	The error code to report.

setFreeBlockAllocatorsAddress()

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

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

Parameters

address	The address to set.

setLastBlock()

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

Sets the last block of the page.

Parameters

block	The block to set as the last block.

setPosition()

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

Sets the current page position.

Parameters

pos	The position to set.

unlock()

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

Unlocks the page after thread-safe operations.

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

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

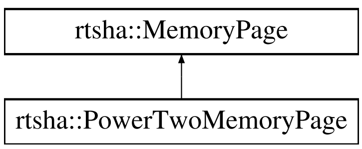
- MemoryPage.h

3.13 rtsha::PowerTwoMemoryPage Class Reference

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

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

Inheritance diagram for rtsha::PowerTwoMemoryPage:



Public Member Functions

- **PowerTwoMemoryPage** ()=delete
  Default constructor is deleted to prevent default instantiation.
- PowerTwoMemoryPage (rtsha_page ∗page)
  Constructor that initializes the PowerTwoMemoryPage with a given page.
- virtual ∼**PowerTwoMemoryPage** ()
  Virtual destructor for the PowerTwoMemoryPage.
- virtual void ∗ allocate_block (const size_t &size) final
  Allocates a memory block of power 2 size.
- virtual void free_block (MemoryBlock &block) final
  This function deallocates memory block.
- void createInitialFreeBlocks ()
  This function creates initial free blocks on empty memory page.

Public Member Functions inherited from rtsha::MemoryPage

- **MemoryPage** ()=delete
  Default constructor is deleted.
- MemoryPage (rtsha_page ∗page) noexcept
  Constructor that initializes the MemoryPage with a given page.
- virtual ∼**MemoryPage** ()
  Virtual destructor.
- bool checkBlock (size_t address)
  Check if a block exists at the given address and if the block is valid.
- virtual void ∗ allocate_block (const size_t &size)=0
  Pure virtual function to allocate a block of memory.
- virtual void free_block (MemoryBlock &block)=0
  Pure virtual function to free a block of memory.
- void ∗ allocate_block_at_current_pos (const size_t &size)
  Allocates a block of memory at the current position.
- void incFreeBlocks ()
  Increments the count of free blocks.

Additional Inherited Members

Protected Member Functions inherited from rtsha::MemoryPage

- void lock ()
  Locks the page for thread-safe operations.
- void unlock ()
  Unlocks the page after thread-safe operations.
- void reportError (uint32_t error)
  Reports an error using the specified error callback.
- void setFreeBlockAllocatorsAddress (const size_t &address)
  Sets the address of the last free block temporary. The addres will be used by InternListAllocator as storage for the elements of the 'std::forward_list'.
- rtsha_page_size_type getPageType () const
  Retrieves the type of the page.
- void ∗ getFreeList () const

Gets the free list pointer of the page.
- void ∗ getFreeListArray () const

  Gets the free list array pointer of the page.
- void ∗ getFreeMap () const

  Gets the free map pointer of the page.
- size_t getFreeBlocks () const

  Retrieves the number of free blocks in the page.
- size_t getMinBlockSize () const

  Retrieves the number of free blocks in the page.
- address_t getPosition () const

  Gets the current page position.
- void setPosition (address_t pos)

  Sets the current page position.
- void incPosition (const size_t &val)

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

  Decreases the oage position by the given value.
- void **decFreeBlocks** ()

  Decreases the number of free blocks in the page.
- address_t getEndPosition () const

  Gets the end position of the page.
- address_t getStartPosition () const

  Gets the start position of the page.
- bool fitOnPage (const size_t &size) const

  Checks if a block of the specified size fits on the page.
- bool hasLastBlock () const

  Checks if the page 'last block' has ben set.
- bool isLastPageBlock (MemoryBlock &block) const

  Determines if the provided block is the last block of the page.
- rtsha_block ∗ getLastBlock () const

  Gets the last block of the page.
- void setLastBlock (const MemoryBlock &block)

  Sets the last block of the page.

Protected Attributes inherited from rtsha::MemoryPage

- rtsha_page ∗ **_page**

  Pointer to the page structure in memory.

3.13.1 Detailed Description

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

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

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

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

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

3.13.2 Constructor & Destructor Documentation

PowerTwoMemoryPage()

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

Constructor that initializes the PowerTwoMemoryPage with a given page.

Parameters

page	The rtsha_page structure to initialize the PowerTwoMemoryPage with.

3.13.3 Member Function Documentation

allocate_block()

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

Allocates a memory block of power 2 size.

Parameters

size	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

block	Previously allocated memory block.

Implements rtsha::MemoryPage.

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

- PowerTwoMemoryPage.h

3.14 internal::PREALLOC_MEMORY< T, n > Struct Template Reference

Memory storage template for pre-allocation.

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

Public Member Functions

- **PREALLOC_MEMORY** ()
  Default constructor.
- PREALLOC_MEMORY (uint8_t init)
  Constructor that initializes memory with a given value.
- void ∗ get_ptr ()
  Retrieves the pointer to the beginning of the memory block.
- void ∗ get_next_ptr ()
  Retrieves the pointer to the next available memory block.

3.14.1 Detailed Description

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

Memory storage template for pre-allocation.

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

Template Parameters

T	Type of the elements the storage will manage.
n	Number of elements of type T the storage will manage. Default is 1.

3.14.2 Constructor & Destructor Documentation

PREALLOC_MEMORY()

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

Constructor that initializes memory with a given value.

Parameters

init	Value used to initialize the memory.

3.14.3 Member Function Documentation

get_next_ptr()

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

Retrieves the pointer to the next available memory block.

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

Returns

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

get_ptr()

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

Retrieves the pointer to the beginning of the memory block.

Returns

Void pointer to the beginning of the memory block.

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

- internal.h

3.15 rtsha::rtsha_block Struct Reference

Represents a block of memory within a memory page.

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

Public Member Functions

- **rtsha_block** ()

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

Public Attributes

- size_t size
- rtsha_block * **prev**

  Pointer to the previous block.

3.15.1 Detailed Description

Represents a block of memory within a memory page.

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

3.15.2 Member Data Documentation

size

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

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

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

- MemoryBlock.h

3.16 rtsha::rtsha_page Struct Reference

Represents a page within the memory system.

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

Public Member Functions

- **rtsha_page** ()

  Default constructor for initialization.

Public Attributes

- address_t **ptr_list_map** = 0U

  Pointer or address to the list/map associated with the page.
- uint32_t **flags** = 0U

  Flags associated with the page.
- address_t **start_position** = 0U

  Start address of page data.
- address_t **end_position** = 0U

  End address of the page.
- address_t **position** = 0U

  Current position or address within the page.
- size_t **free_blocks** = 0U

  Number of free blocks within the page.
- rtsha_block ∗ **last_block** = NULL

  Pointer to the last block within the page.
- address_t **lastFreeBlockAddress** = 0U

  Address of the last free block within the page.
- address_t **start_map_data** = 0U

  Start address or position of map data for the page.
- rtsha_page ∗ **map_page** = 0U

  Associated map page if any. Used with together with 'Big Memory Page' and 'Power Two Page'.
- size_t **max_blocks** = 0U

  Maximum number of blocks supported by the page.
- size_t **min_block_size** = 0U

  Minimum block size for the page (used with Power Two pages).
- size_t **max_block_size** = 0U

  Maximum block size for the page (used with PowerTwo pages).
- HeapCallbacksStruct ∗ **callbacks** = NULL

  Callback functions associated with the page.
- rtsha_page ∗ **next** = NULL

  Pointer to the next page similar structure.

3.16.1 Detailed Description

Represents a page within the memory system.

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

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

- MemoryPage.h

3.17 rtsha::RTSHMallocator< 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

- **RTSHMallocator** ()=default

  Default constructor.

- template< class U >
  constexpr RTSHMallocator (const RTSHMallocator< U > &) noexcept

  Copy constructor.

- T ∗ allocate (std::size_t n)

  Allocate memory.

- void deallocate (T ∗p, std::size_t n) noexcept

  Deallocate memory.

3.17.1 Detailed Description

template<class T>
struct rtsha::RTSHMallocator< 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

T	The type of elements being allocated.

3.17.2 Constructor & Destructor Documentation

RTSHMallocator()

```
template<class T >
template<class U >
constexpr rtsha::RTSHMallocator< T >::RTSHMallocator (
            const RTSHMallocator< 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

U	The source type for the allocator.

3.17.3 Member Function Documentation

allocate()

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

Allocate memory.

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

Parameters

n	Number of items of type `T` to allocate memory for.

Returns

T∗ Pointer to the allocated memory.

Exceptions

std::bad_alloc	If memory allocation fails.
std::bad_array_new_length	If the allocation size exceeds system limits.

deallocate()

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

Deallocate memory.

Release previously allocated memory.

Parameters

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

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

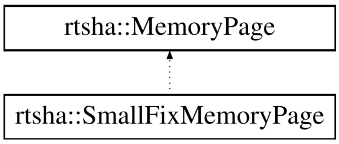
- FastPlusAllocator.h

3.18 rtsha::SmallFixMemoryPage Class Reference

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

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

Inheritance diagram for rtsha::SmallFixMemoryPage:



Public Member Functions

- **SmallFixMemoryPage** ()=delete

  Default constructor is deleted to prevent default instantiation.
- SmallFixMemoryPage (rtsha_page ∗page)

  Constructor that initializes the SmallFixMemoryPage with a given page.
- virtual ∼**SmallFixMemoryPage** ()

  Virtual destructor for the SmallFixMemoryPage.
- virtual void ∗ allocate_block (const size_t &size) final

  Allocates a memory block of fixed size.
- virtual void free_block (MemoryBlock &block) final

  This function deallocates memory block.

3.18.1 Detailed Description

This class provides various memory handling functions that manipulate MemoryBlock's on memory page with fixed blocked size. This algorithm is an approach to memory management that is often used in specific situations where objects of a certain size are frequently allocated and deallocated. By using of uses 'Fixed chunk size' algorithm greatly simplies 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 internaly by ¨Power Two Memory Page¨ and ¨Big Memory Page¨ algorithms.

3.18.2 Constructor & Destructor Documentation

SmallFixMemoryPage()

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

Constructor that initializes the SmallFixMemoryPage with a given page.

Parameters

page	The rtsha_page structure to initialize the SmallFixMemoryPage with.

3.18.3 Member Function Documentation

allocate_block()

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

Allocates a memory block of fixed size.

Parameters

size	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

block	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
```

```
// This entire file will not be documented by Doxygen.


#include "internal.h"
#include "HeapCallbacks.h"


#if defined(_MSC_VER) || defined(__MINGW32__)

#if !defined(RTSHA_SHARED_LIB)
#define rtsha_decl_export
#elif defined(RTSHA_SHARED_LIB_EXPORT)
#define rtsha_decl_export              __declspec(dllexport)
#else
#define rtsha_decl_export              __declspec(dllimport)
#endif

#define rtsha_cdecl                      __cdecl
#elif defined(__GNUC__)
#if defined(rtsha_SHARED_LIB) && defined(rtsha_SHARED_LIB_EXPORT)
#define rtsha_decl_export              __attribute__((visibility("default")))
#else
#define rtsha_decl_export
#endif
#define rtsha_cdecl
#else
#define rtsha_cdecl
#define rtsha_decl_export
#endif



#define RTSHA_PAGE_TYPE_32          32U
#define RTSHA_PAGE_TYPE_64          64U
#define RTSHA_PAGE_TYPE_128         128U
#define RTSHA_PAGE_TYPE_256         256U
#define RTSHA_PAGE_TYPE_512         512U
#define RTSHA_PAGE_TYPE_BIG         613U
#define RTSHA_PAGE_TYPE_POWER_TWO   713U



#ifdef __cplusplus
extern "C"
{
#endif
    /*rtsha_decl_nodiscard rtsha_decl_export*/  bool  rtsha_create_heap(void* start, size_t size);

    rtsha_decl_export                       bool  rtsha_add_page(HeapCallbacksStruct* callbacks, uint16_t page_type, size_t size, size_t max_objects = 0U, size_t min_block_size = 0U, size_t max_block_size = 0U);

    rtsha_decl_nodiscard rtsha_decl_export  void* rtsha_malloc(size_t size);

    rtsha_decl_export                       void  rtsha_free(void* ptr);

    rtsha_decl_nodiscard rtsha_decl_export  void* rtsha_calloc(size_t nitems, size_t size);

    rtsha_decl_nodiscard rtsha_decl_export  void* rtsha_realloc(void* ptr, size_t size);

    rtsha_decl_export                       void* rtsha_memcpy(void* _Dst, void const* _Src, size_t _Size);

    rtsha_decl_export                       void* rtsha_memset(void* _Dst, int _Val, size_t _Size);

#ifdef __cplusplus
}
#endif

```

4.2 arm_spec_functions.h

```
#pragma once

#ifdef __arm__ //ARM architecture
#include <cstdint>
#include <stddef.h>

void* arm_wide64_memcpy(void* dst, const void* src, size_t n);
#endif
```

4.3 BigMemoryPage.h

```cpp
#pragma once
#include <stdint.h>
#include "MemoryPage.h"
namespace rtsha
{
    using namespace std;
    class BigMemoryPage : public MemoryPage
    {
    public:
        BigMemoryPage() = delete;
        explicit BigMemoryPage(rtsha_page* page) : MemoryPage(page)
        {
        }
        virtual ~BigMemoryPage()
        {
        }
        virtual void* allocate_block(const size_t& size) final;
        virtual void free_block(MemoryBlock& block) final;
        void createInitialFreeBlocks();
    protected:
        void splitBlock(MemoryBlock& block, size_t size);
        void mergeLeft(MemoryBlock& block);
        void mergeRight(MemoryBlock& block);
    };
}
```

4.4 errors.h

```cpp
#include "internal.h"
#pragma once
#define RTSHA_OK                            (0U)
#define RTSHA_ErrorInit                     (16U)
#define RTSHA_ErrorInitPageSize             (32U)
#define RTSHA_ErrorInitOutOfHeap            (33U)
#define RTSHA_OutOfMemory                   (64U)
#define RTSHA_NoPages                       (128U)
#define RTSHA_NoPage                        (129U)
#define RTSHA_NoFreePage                    (130U)
#define RTSHA_InvalidBlock                  (256U)
#define RTSHA_InvalidBlockDistance          (257U)
#define RTSHA_InvalidNumberOfFreeBlocks     (258U)
 // end of RTSHA_ERRORS group
```

4.5 FastPlusAllocator.h

```cpp
#pragma once
#include <cstdlib>
#include <new>
#include <limits>
#include <iostream>
#include "allocator.h"
namespace rtsha
```

```
{

    template<class T>
    struct RTSHMallocator
    {
        typedef T value_type;

        RTSHMallocator() = default;


        template<class U>
        constexpr RTSHMallocator(const RTSHMallocator <U>&) noexcept {}

        [[nodiscard]] T* allocate(std::size_t n)
        {
            if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
                throw std::bad_array_new_length();

            if (auto p = static_cast<T*>(rtsha_malloc(n * sizeof(T))))
            {
                //report(p, n);
                return p;
            }

            throw std::bad_alloc();
        }

        void deallocate(T* p, std::size_t n) noexcept
        {
            //report(p, n, 0);
            rtsha_free(p);
        }
    private:

        void report(T* p, std::size_t n, bool alloc = true) const
        {
            std::cout « (alloc ? "Alloc: " : "Dealloc: ") « sizeof(T) * n
                « " bytes at " « std::hex « std::showbase
                « reinterpret_cast<void*>(p) « std::dec « '\n';
        }
    };

    template<class T, class U>
    bool operator==(const RTSHMallocator <T>&, const RTSHMallocator <U>&) { return true; }

    template<class T, class U>
    bool operator!=(const RTSHMallocator <T>&, const RTSHMallocator <U>&) { return false; }
}
```

4.6 FreeLinkedList.h

```
#pragma once
#include <stdint.h>
#include "internal.h"
#include "MemoryPage.h"
#include <bitset>

namespace internal
{
    using namespace rtsha;

    class alignas(sizeof(size_t))  FreeLinkedList
    {
        struct alignas(sizeof(size_t)) Node
        {
            size_t data;
            Node* prev;
            Node* next;

            explicit Node(const size_t& data, Node* head, Node* prev = nullptr) : data(data), next(head), prev(prev)
            {
            };
        };

    public:

        explicit FreeLinkedList(rtsha_page* page) : head(NULL), _page(page), count(0U)
        {
        }

        ~FreeLinkedList()
        {
        }
```

```
        rtsha_attr_inline void push(const size_t& data)
        {
            Node* newNode = new (reinterpret_cast<void*>(_page->lastFreeBlockAddress)) Node(data, head, nullptr);

            if (head)
            {
                head->prev = newNode;
            }
            head = newNode;

            count++;
        }

        rtsha_attr_inline bool is_empty() const
        {
            if (head == nullptr)
            {
                assert(count == 0);
                return true;
            }
            if (count == 0)
            {
                return true;
            }
            return false;
        }

        rtsha_attr_inline size_t pop()
        {
            size_t ret(0U);
            if (!is_empty() && head != NULL)
            {
                Node* temp = head;
                ret = temp->data;
                head = head->next;

                if( count > 0U )
                    count--;

                if (count == 0U)
                {
                    head = NULL;
                }
            }
            else
            {
                count = 0U;
            }
            return ret;
        }

        rtsha_attr_inline bool delete_address(const size_t& address, void* block)
        {
            if (head == nullptr)
            {
                // the list is empty
                count = 0U;
                return false;
            }
            Node* temp = reinterpret_cast<Node*>(address);

            if (head->data == temp->data)
            {
                //the node to delete is the head
                if (head->next)
                    head->next->prev = nullptr;
                head = head->next;

                temp->data = 0U;

                if (count)
                    count--;
                return true;
            }
            size_t blockAddress = reinterpret_cast<size_t>(block);
            if (temp->data == blockAddress)
            {
                temp->data = 0U;
                if (temp->prev)
                {
                    temp->prev->next = temp->next;
                }
                if (temp->next)
                {
                    temp->next->prev = temp->prev;
```

```
                }
                if (count)
                    count--;
                return true;
            }
            return false;
        }

    private:
        rtsha_page* _page;
        Node* head;
        size_t count = 0U;
    };
}
```

4.7 FreeList.h

```
#pragma once
#include <stdint.h>
#include "MemoryBlock.h"
#include "InternListAllocator.h"
#include "FreeLinkedList.h"

namespace internal
{
    using namespace std;

    using flist = FreeLinkedList;

    class alignas(sizeof(size_t)) FreeList
    {
    public:

        FreeList() = delete;

        explicit FreeList(rtsha_page* page);

        ~FreeList()
        {
        }

        rtsha_attr_inline void push(const size_t& address)
        {
            ptrLlist->push(address);
        }

        rtsha_attr_inline size_t pop()
        {
            return ptrLlist->pop();
        }

        rtsha_attr_inline bool delete_address(const size_t& address, void* block)
        {
            return  (ptrLlist->delete_address(address, block));
        }

    private:

        rtsha_page* _page;


        flist* ptrLlist;

        PREALLOC_MEMORY<flist> _storage_list = 0U;
    };
}

```

4.8 FreeListArray.h

```
#pragma once
#include <stdint.h>
#include "MemoryBlock.h"
#include "FreeLinkedList.h"
#include "MemoryPage.h"
#include <bit>

namespace internal
{
    #define MIN_BLOCK_SIZE  32U
    #define MAX_BLOCK_SIZE 0x4000000U
```

```
class alignas(sizeof(size_t)) FreeListArray
{

public:

    FreeListArray() = delete;

    explicit FreeListArray(rtsha_page* page, size_t min_block_size, size_t page_size);

    ~FreeListArray()
    {
    }

    rtsha_attr_inline void push(const size_t data, size_t size)
    {
        if ((data > _page->start_position) && (data < _page->end_position))
        {
            int32_t index = std::bit_width(size) - min_bin - 1;
            assert(index >= 0);
            if (index >= 0)
            {
                arrPtrLists[index]->push(data);
            }
        }
    }

    rtsha_attr_inline size_t pop(size_t size)
    {
        size_t ret(0U);
        size_t log2_size = std::bit_width(size);
        assert(log2_size > min_bin);
        if (log2_size > min_bin)
        {
            for (size_t n = log2_size; n <= max_bin; n++)
            {
                int32_t index = n - min_bin - 1;
                assert(index >= 0);
                if (index >= 0)
                {
                    if (false == arrPtrLists[index]->is_empty())
                    {
                        ret = arrPtrLists[index]->pop();
                        if (ret > 0U)
                        {
                            if ((ret > _page->start_position) && (ret < _page->end_position))
                            {
                                return ret;
                            }
                        }
                    }
                }
            }
        }
        return 0U;
    }

    rtsha_attr_inline bool delete_address(const size_t& address, void* block, const size_t& size)
    {
        if ((address > _page->start_position) && (address < _page->end_position))
        {
            size_t log2_size = std::bit_width(size);
            if (log2_size >= min_bin)
            {
                for (size_t n = log2_size; n <= max_bin; n++)
                {
                    int32_t index = n - min_bin - 1;
                    assert(index >= 0);
                    if (index >= 0)
                    {
                        if (arrPtrLists[n - min_bin - 1]->delete_address(address, block))
                        {
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }

private:

    size_t min_bin;

    size_t max_bin;

```

```
        rtsha_page* _page;

        FreeLinkedList* arrPtrLists[MAX_BINS];

        PREALLOC_MEMORY<FreeLinkedList, MAX_BINS> _storage_list_array = 0U;
    };
}
```

4.9 FreeMap.h

```
#pragma once
#include <stdint.h>
#include "MemoryBlock.h"
#include "InternMapAllocator.h"
#include "internal.h"
#include "map"

namespace internal
{
    using namespace std;
    using namespace rtsha;

    using mmap_allocator = InternMapAllocator<std::pair<const uint64_t, size_t>>;

    using mmap = std::multimap<const uint64_t, size_t, std::less<const uint64_t>,
internal::InternMapAllocator<std::pair<const uint64_t, size_t>»;

    class alignas(sizeof(size_t)) FreeMap
    {
    public:

        FreeMap() = delete;

        explicit FreeMap(rtsha_page* page);

        ~FreeMap()
        {
        }

        rtsha_attr_inline void insert(const uint64_t key, size_t block)
        {
            if ((_ptrMap != nullptr))
            {
                _ptrMap->insert(std::pair<const uint64_t, size_t>(key, block));
            }
        }

        rtsha_attr_inline bool del(const uint64_t key, size_t block)
        {
            if ((_ptrMap != nullptr))
            {
                mmap::iterator it = _ptrMap->find(key);
                while(it != _ptrMap->end())
                {
                    if ((it->first == key) && (it->second == block))
                    {
                        it = _ptrMap->erase(it);
                        return true;
                    }
                    it++;
                }
            }
            return false;
        }

        rtsha_attr_inline size_t find(const uint64_t key)
        {
            if ((_ptrMap != nullptr))
            {
                mmap::iterator it = _ptrMap->lower_bound(key);
                if (it != _ptrMap->end())
                {
                    return it->second;
                }
            }
            return 0U;
        }


        rtsha_attr_inline bool exists(const uint64_t key, size_t block)
        {
            if ((_ptrMap != nullptr))
            {
                mmap::iterator it = _ptrMap->find(key);
```

```
                if (it != _ptrMap->end())
                {
                    if ((it->first == key) && (it->second == block))
                    {
                        return true;
                    }
                }
            }
            return false;
        }

        size_t size() const
        {
            if ((_ptrMap != nullptr))
            {
                return _ptrMap->size();
            }
            return 0U;
        }



    private:
        rtsha_page*           _page;
        mmap_allocator*       _mallocator;
        mmap*               _ptrMap;


    private:

        PREALLOC_MEMORY <mmap_allocator>        _storage_allocator = 0U;
        PREALLOC_MEMORY<mmap>                  _storage_map = 0U;
    };
}
```

4.10 Heap.h

```
#pragma once
#include <stdint.h>
#include "MemoryPage.h"
#include "errors.h"
#include "FreeList.h"
#include "FreeListArray.h"
#include "FreeMap.h"
#include <array>

namespace internal
{
    using namespace std;

    class HeapInternal
    {
    public:
        HeapInternal():_big_page_used(false)
        {
            for (size_t i = 0; i < _pages.size(); i++)
            {
                _pages[i] = nullptr;
            }
        }

        ~HeapInternal()
        {

        }


        FreeList* createFreeList(rtsha_page* page);

        FreeMap* createFreeMap(rtsha_page* page);

        FreeListArray* createFreeListArray(rtsha_page* page, size_t page_size);


    protected:

        void init_small_fix_page(rtsha_page* page, size_t a_size);

        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);

        void init_big_block_page(rtsha_page* page, size_t a_size, size_t max_objects);

    protected:
```

```
        std::array<rtsha_page*, MAX_PAGES>  _pages;

        size_t      _number_pages = 0U;

        address_t   _heap_start = 0U;

        size_t      _heap_size = 0U;

        address_t   _heap_current_position = 0U;

        address_t   _heap_top = 0U;

        bool        _heap_init = false;

        uint32_t    _last_heap_error = RTSHA_OK;

        bool _big_page_used;


        PREALLOC_MEMORY<FreeList, (MAX_SMALL_PAGES + MAX_BIG_PAGES)> _storage_free_lists = 0U;

        PREALLOC_MEMORY<FreeListArray, MAX_POWER_TWO_PAGES>  _storage_free_list_array = 0U;

        PREALLOC_MEMORY<FreeMap, MAX_BIG_PAGES>      _storage_free_maps = 0U;
    };
}

namespace rtsha
{
    using namespace std;
    using namespace internal;

    class Heap : HeapInternal
    {
    public:

        Heap();


        ~Heap();


        bool init(void* start, size_t size);


        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);

        size_t get_free_space() const;

        rtsha_page* get_big_memorypage() const;

        rtsha_page* get_block_page(address_t block_address);


        void* malloc(size_t size);

        void free(void* ptr);

        void* calloc(size_t nitems, size_t size);

        void* realloc(void* ptr, size_t size);

        void* memcpy(void* _Dst, void const* _Src, size_t _Size);

        void* memset(void* _Dst, int _Val, size_t _Size);


        rtsha_page_size_type get_ideal_page(size_t size) const;

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

    };
}

```

4.11 HeapCallbacks.h

```
#pragma once

typedef void (*rtshLockPagePtr)     (void);

```

```
typedef void (*rtshUnLockPagePtr)    (void);

typedef void (*rtshErrorPagePtr)     (uint32_t);

typedef struct HeapCallbacksStruct
{
    rtshLockPagePtr     ptrLockFunction;
    rtshLockPagePtr     ptrUnLockFunction;
    rtshErrorPagePtr    ptrErrorFunction;
} HeapCallbacks;

```

4.12 internal.h

```
#pragma once

#include <assert.h>
#include <cstring>
#include "stdint.h"

#ifdef _MSC_VER
#  include <immintrin.h>
#  include <intrin.h>
#else
#endif



#define MULTITHREADING_SUPPORT


#define MAX_BLOCKS_PER_PAGE UINT32_MAX

#define MAX_SMALL_PAGES         32U
#define MAX_BIG_PAGES           2U
#define MAX_POWER_TWO_PAGES     2U

#define MAX_PAGES               (MAX_SMALL_PAGES+MAX_BIG_PAGES+MAX_POWER_TWO_PAGES)

#define MAX_BINS 27U

#if (__MSC_VER >= 1930 )
#define rtsha_attr_inline inline __forceinline
#else
#define rtsha_attr_inline inline
#endif



#if defined _WIN64 || defined _ARM64
#define RTSHA_BLOCK_HEADER_SIZE  (2 * sizeof(size_t))
#define MIN_BLOCK_SIZE_FOR_SPLIT    56U /*todo*/
#else
#define RTSHA_LIST_ITEM_SIZE  (2 * sizeof(size_t))
#define MIN_BLOCK_SIZE_FOR_SPLIT    512U /*todo*/
#endif

#ifdef __cplusplus
#if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
#define rtsha_attr_noexcept   noexcept
#else
#define rtsha_attr_noexcept   throw()
#endif
#else
#define rtsha_attr_noexcept
#endif


#if (__cplusplus >= 201103L) || (_MSC_VER >= 1930)
#define rtsha_decl_nodiscard    [[nodiscard]]
#elif (defined(__GNUC__) && (__GNUC__ >= 4)) || defined(__clang__)
#define rtsha_decl_nodiscard    __attribute__((warn_unused_result))
#elif defined(_HAS_NODISCARD)
#else
#define rtsha_decl_nodiscard
#endif





#define RTSHA_ALIGMENT          sizeof(size_t)  /*4U or 8U*/

#define is_bit(val,n) ( (val » n) & 0x01U )
```

```
#ifndef rtsha_assert
    #define rtsha_assert(x) assert(x)
#endif


#if _WIN32 || _WIN64
    #if defined(WIN64) || defined(__amd64__)
        #define ENV64BIT
    #else
        #define ENV32BIT
    #endif
#else
    #if __GNUC__
        #if __x86_64__ || __ppc64__
            #define ENV64BIT
        #else
            #define ENV32BIT
        #endif
    #endif
#endif


#ifdef __arm__ //ARM architecture
#define ARCH_ARM
#endif

#ifdef __aarch64__ //ARM 64-bit
#define ARCH_ARM
#define ARCH_ARM_64
#define ARCH_64BIT
#endif


namespace internal
{
    using address_t = uintptr_t;

#if defined(WIN64) // The _BitScanReverse64 intrinsic is only available for 64 bit builds because it depends on x64

    inline uint64_t ExpandToPowerOf2(uint64_t Value)
    {
        unsigned long Index;
        _BitScanReverse64(&Index, Value - 1);
        return (1ULL « (Index + 1));
    }
#else


#ifdef __arm__ //ARM architecture

    rtsha_attr_inline uint32_t ExpandToPowerOf2(uint32_t Value)
    {
        unsigned long leading_zeros = __builtin_clz(Value);
        return (1U « (32U - leading_zeros));
    }

#else
    inline uint32_t ExpandToPowerOf2(uint32_t Value)
    {
        unsigned long Index;
        _BitScanReverse(&Index, Value - 1);
        return (1U « (Index + 1));
    }
#endif
#endif


    template<typename T, size_t n = 1U>
    struct alignas(sizeof(size_t)) PREALLOC_MEMORY
    {
    public:

        PREALLOC_MEMORY()
        {
        }

        PREALLOC_MEMORY(uint8_t init)
        {
            std::memset(_memory, init, sizeof(_memory));
        }

    public:

        inline void* get_ptr()
        {
```

```
            return (void*)_memory;
        }

        inline void* get_next_ptr()
        {
            if (_count < n)
            {
                void* ret = (void*)(_memory + _count * sizeof(T));
                _count++;
                return ret;
            }
            return nullptr;
        }
    private:
        uint8_t _memory[n * sizeof(T)];
        size_t _count = 0U;
    };

    static inline uintptr_t rtsha_align(uintptr_t ptr, size_t aligment)
    {
        uintptr_t mask = aligment - 1U;

        if ((aligment & mask) == 0U)
        {
            return ((ptr + mask) & ~mask);
        }
        return (((ptr + mask) / aligment) * aligment);
    }

}
```

4.13 InternListAllocator.h

```
#pragma once
#include "MemoryPage.h"
#include <cstdlib>

#ifdef _RTSHA_DIAGNOSTIK
#include <iostream>
#endif

namespace internal
{
    using namespace rtsha;

    template<class T>
    struct InternListAllocator
    {
        typedef T value_type;

        explicit InternListAllocator(rtsha_page* page, size_t* _ptrSmallStorage)
            : _page(page),

            _allocated_intern(0U),
            _ptrInternalSmallStorage(_ptrSmallStorage)
        {
        }

        template<class U>
        constexpr InternListAllocator(const InternListAllocator <U>& rhs) noexcept
        {
            this->_page              = rhs._page;
            this->_allocated_intern = rhs._allocated_intern;
            this->_ptrInternalSmallStorage = rhs._ptrInternalSmallStorage;
        }

        [[nodiscard]] rtsha_attr_inline T* allocate(std::size_t n)  noexcept
        {
            /*max. 1 block*/
            if (n != 1U)
            {
                return nullptr;
            }
            if ((_allocated_intern == 0U) && (_page->lastFreeBlockAddress == 0U))
            {
                if (n > 1U)
                {
                    return nullptr;
                }
                _allocated_intern++;
                auto p = static_cast<T*>((void*)_ptrInternalSmallStorage);
                return p;
            }
            auto p = static_cast<T*>( (void*) _page->lastFreeBlockAddress);
```

```
            return p;
        }

        rtsha_attr_inline void deallocate(T* /*p*/, std::size_t /*n*/) noexcept
        {
        }

        rtsha_page* _page;
        size_t      _allocated_intern           = 0U;
        size_t*     _ptrInternalSmallStorage    = NULL;

    private:

#ifdef _RTSHA_DIAGNOSTIK
        void report(T* p, std::size_t n, bool alloc = true) const
        {
            std::cout « (alloc ? "LAlloc: " : "LDealloc: ") « sizeof(T) * n
                « " bytes at " « std::hex « std::showbase
                « reinterpret_cast<void*>(p) « std::dec « '\n';
        }
#endif
    };

    template<class T, class U>
    bool operator==(const InternListAllocator <T>&, const InternListAllocator <U>&) { return true; }

    template<class T, class U>
    bool operator!=(const InternListAllocator <T>&, const InternListAllocator <U>&) { return false; }
}

```

4.14 InternMapAllocator.h

```
#pragma once
#include "SmallFixMemoryPage.h"
#include <cstdlib>
#ifdef _RTSHA_DIAGNOSTIK
#include <iostream>
#endif

namespace internal
{
    #if defined _WIN64 || defined _ARM64
        #define INTERNAL_MAP_STORAGE_SIZE   64U
    #else
        #define INTERNAL_MAP_STORAGE_SIZE   32U
    #endif

    using namespace rtsha;

    template<class T>
    struct InternMapAllocator
    {
        typedef T value_type;

        InternMapAllocator(rtsha_page* page)
            :_page(page)
        {
        }

        template<class U>
        constexpr InternMapAllocator(const InternMapAllocator <U>& rhs) noexcept
        {
            this->_page              = rhs._page;
        }

        [[nodiscard]] rtsha_attr_inline T* allocate(std::size_t n)   noexcept
        {
            if (_page->map_page != nullptr)
            {
                SmallFixMemoryPage map_page(_page->map_page);
                if ((size_t)_page->map_page->flags >= (n * sizeof(T)))
                {
                    auto p =
reinterpret_cast<T*>(map_page.allocate_block((size_t)_page->map_page->flags));
                    if (p != nullptr)
                    {
                        //report(p, n, 1);
                        return p;
                    }
                }
            }
            return NULL;
        }
```

```
        rtsha_attr_inline void deallocate(T*p, std::size_t /*n*/) noexcept
        {
            if (_page->map_page != nullptr)
            {
                SmallFixMemoryPage map_page(_page->map_page);

                size_t address = reinterpret_cast<size_t>(p);
                address -= (2U * sizeof(size_t)); /*skip size and pointer to prev*/

                MemoryBlock block((rtsha_block*)(void*)address);

                if (block.isValid())
                {
                    map_page.free_block(block);
                    //report(p, n, 0);
                }
            }
        }

        rtsha_page* _page;

    private:
#ifdef _RTSHA_DIAGNOSTIK
        void report(T* p, std::size_t n, bool alloc = true) const
        {
            std::cout << (alloc ? "MAlloc: " : "MDealloc: ") << sizeof(T) * n
                << " bytes at " << std::hex << std::showbase
                << reinterpret_cast<void*>(p) << std::dec << '\n';
        }
#endif
    };

    template<class T, class U>
    bool operator==(const InternMapAllocator <T>&, const InternMapAllocator <U>&) { return true; }

    template<class T, class U>
    bool operator!=(const InternMapAllocator <T>&, const InternMapAllocator <U>&) { return false; }

}
```

4.15 MemoryBlock.h

```
#pragma once
#pragma once
#include <stdint.h>
#include "internal.h"

namespace rtsha
{
    using namespace std;

    struct rtsha_block
    {
        rtsha_block()
            :size(0U)
            ,prev(NULL)
        {
        }

        size_t                      size;
        rtsha_block* prev;
    };

    class MemoryBlock
    {
    public:

        MemoryBlock() = delete;

        explicit MemoryBlock(rtsha_block* block) : _block(block)
        {
        }

        ~MemoryBlock()
        {
        }

        MemoryBlock& operator = (const MemoryBlock& rhs)
        {
            this->_block = rhs._block;
            return *this;
        }

```

```
    void splitt(const size_t& new_size, bool last);

    void splitt_22();

    void merge_left();

    void merge_right();


    rtsha_attr_inline void setAllocated()
    {
        _block->size = (_block->size » 1U) « 1U;
    }

    rtsha_attr_inline void setFree()
    {
        _block->size = (_block->size | 1U);
    }

    rtsha_attr_inline void setLast()
    {
        _block->size = (_block->size | 2U);
    }

    rtsha_attr_inline void clearIsLast()
    {
        _block->size &= ~(1UL « 1U);
    }

    rtsha_attr_inline rtsha_block* getBlock() const
    {
        return _block;
    }

    rtsha_attr_inline void* getAllocAddress() const
    {
        return reinterpret_cast<void*>((size_t)_block + 2U * sizeof(size_t));
    }

    rtsha_attr_inline size_t getSize() const
    {
        return (_block->size » 2U) « 2U;
    }

    rtsha_attr_inline bool isValid() const
    {
        if (_block != nullptr)
        {
            size_t size = getSize();
            if ((_block != _block->prev) && (size > sizeof(size_t)))
            {
                size_t* ptrSize2 = reinterpret_cast<size_t*>((size_t)_block + size - sizeof(size_t));
                return (*ptrSize2 == size);
            }
        }
        return false;
    }

    rtsha_attr_inline void setSize( size_t size )
    {
        if (size > sizeof(size_t))
        {
            bool free = isFree();
            bool last = isLast();
            _block->size = size;
            if (free)
            {
                setFree();
            }
            if (last)
            {
                setLast();
            }
            size_t* ptrSize2 = reinterpret_cast<size_t*>((size_t)_block + size - sizeof(size_t));
            *ptrSize2 = size;
        }
        else
        {
            _block->size = 0U;
        }
    }

    rtsha_attr_inline size_t getFreeBlockAddress() const
    {
        return ((size_t)_block + 2U * sizeof(size_t));
```

```
        }

        rtsha_attr_inline void setPrev(const MemoryBlock& prev)
        {
            if (prev.isValid())
            {
                _block->prev = prev.getBlock();
            }
            else
            {
                _block->prev = NULL;
            }
        }

        rtsha_attr_inline void setAsFirst()
        {
            _block->prev = NULL;
        }

        rtsha_attr_inline bool isFree()
        {
            return is_bit(_block->size, 0U);
        }

        rtsha_attr_inline bool isLast()
        {
            return is_bit(_block->size, 1U);
        }

        rtsha_attr_inline bool hasPrev()
        {
            return (_block->prev != NULL);
        }

        rtsha_attr_inline rtsha_block* getNextBlock() const
        {
            return reinterpret_cast<rtsha_block*>((size_t)_block + this->getSize());
        }

        rtsha_attr_inline rtsha_block* getPrev() const
        {
            return _block->prev;
        }

        rtsha_attr_inline void prepare()
        {
            _block->prev = NULL;
            _block->size = 0;
        }

    private:
        rtsha_block* _block;
    };
}
```

4.16 MemoryPage.h

```
#pragma once
#include "internal.h"
#include <stdint.h>
#include "MemoryBlock.h"
#include "HeapCallbacks.h"


namespace rtsha
{
    using namespace std;
    using namespace internal;

    enum struct rtsha_page_size_type : uint16_t
    {
        PageTypeNotDefined = 0U,

        PageType16  = 16U,
        PageType32  = 32U,
        PageType64  = 64U,
        PageType128 = 128U,
        PageType256 = 256U,
        PageType512 = 512U,

        PageTypeBig         = 613U,
        PageTypePowerTwo    = 713U
    };

```

```
    struct rtsha_page
    {
        rtsha_page()
        {
        }

        address_t                   ptr_list_map            = 0U;

        uint32_t                    flags                   = 0U;

        address_t                   start_position            = 0U;
        address_t                   end_position            = 0U;

        address_t                   position                = 0U;
        size_t                      free_blocks              = 0U;

        rtsha_block*             last_block                = NULL;

        address_t                   lastFreeBlockAddress    = 0U;

        address_t                   start_map_data            = 0U;

        rtsha_page*                   map_page                = 0U;

        size_t                      max_blocks                = 0U;

        size_t                      min_block_size            = 0U;
        size_t                      max_block_size            = 0U;

        HeapCallbacksStruct*     callbacks              = NULL;

        rtsha_page*                   next                      = NULL;
    };

    class MemoryPage
    {
    public:

        MemoryPage() = delete;

        explicit MemoryPage(rtsha_page* page) noexcept
            : _page(page)
        {
        }

        virtual ~MemoryPage()
        {
        }

        bool checkBlock(size_t address);

        virtual void* allocate_block(const size_t& size) = 0;

        virtual void free_block(MemoryBlock& block) = 0;

        void* allocate_block_at_current_pos(const size_t& size);

        inline void incFreeBlocks()
        {
            if (_page != nullptr)
            {
                _page->free_blocks++;
            }
        }

    protected:

        inline void lock()
        {
            #ifdef MULTITHREADING_SUPPORT
            if ((nullptr != this->_page->callbacks) && (nullptr != this->_page->callbacks->ptrLockFunction))
            {
                this->_page->callbacks->ptrLockFunction();
            }
            #endif
        }

        inline void unlock()
        {
            #ifdef MULTITHREADING_SUPPORT
            if ((nullptr != this->_page->callbacks) && (nullptr != this->_page->callbacks->ptrUnLockFunction))
            {
                this->_page->callbacks->ptrUnLockFunction();
            }
            #endif
```

```
        }

        inline void reportError(uint32_t error)
        {
            if ((nullptr != this->_page->callbacks) && (nullptr != this->_page->callbacks->ptrErrorFunction))
            {
                this->_page->callbacks->ptrErrorFunction(error);
            }
        }

        inline void setFreeBlockAllocatorsAddress(const size_t& address)
        {
            _page->lastFreeBlockAddress = address;
        }

        inline rtsha_page_size_type getPageType() const
        {
            return (rtsha_page_size_type) _page->flags;
        }

        inline void* getFreeList() const
        {
            return reinterpret_cast<void*>(_page->ptr_list_map);
        }

        inline void* getFreeListArray() const
        {
            return reinterpret_cast<void*>(_page->ptr_list_map);
        }

        inline void* getFreeMap() const
        {
            return reinterpret_cast<void*>(_page->ptr_list_map);
        }

        inline size_t getFreeBlocks() const
        {
            if (_page != nullptr)
            {
                return _page->free_blocks;
            }
            return 0U;
        }

        inline size_t getMinBlockSize() const
        {
            if (_page != nullptr)
            {
                return _page->min_block_size;
            }
            return 0U;
        }

        inline address_t getPosition() const
        {
            if (_page != nullptr)
            {
                return _page->position;
            }
            return 0U;
        }

        inline void setPosition(address_t pos)
        {
            if (_page != nullptr)
            {
                _page->position = pos;
            }
        }

        inline void incPosition(const size_t& val)
        {
            if (_page != nullptr)
            {
                _page->position += val;
            }
        }

        inline void decPosition(const size_t& val)
        {
            if (_page != nullptr)
            {
                if (_page->position >= val)
                {
                    _page->position -= val;
                }
```

```
            }
        }

        inline void decFreeBlocks()
        {
            if ( (_page != nullptr) && (_page->free_blocks > 0U) )
            {
                _page->free_blocks--;
            }
        }

        inline address_t getEndPosition() const
        {
            return _page->end_position;
        }

        inline address_t getStartPosition() const
        {
            return _page->start_position;
        }

        inline bool fitOnPage(const size_t& size) const
        {
            if ((_page->position + size) < (_page->end_position))
            {
                if (_page->start_map_data == 0U)
                {
                    return true;
                }
                else
                {
                    if ((_page->position + size) < _page->start_map_data)
                    {
                        return true;
                    }
                }
            }
            return false;
        }

        inline bool hasLastBlock() const
        {
            return (_page->last_block != nullptr);
        }

        inline bool isLastPageBlock(MemoryBlock& block) const
        {
            if (this->getPosition() == ((size_t)block.getBlock() + block.getSize()))
            {
                return (block.getBlock() == _page->last_block);
            }
            return false;
        }

        inline rtsha_block* getLastBlock() const
        {
            return _page->last_block;
        }

        inline void setLastBlock(const MemoryBlock& block)
        {
            _page->last_block = block.getBlock();
        }

        rtsha_page* _page;

    };
}

```

4.17 PowerTwoMemoryPage.h

```
#pragma once
#include <stdint.h>
#include "internal.h"
#include "MemoryPage.h"

namespace rtsha
{
    using namespace std;

    class PowerTwoMemoryPage : public MemoryPage
    {
    public:
```

```

        PowerTwoMemoryPage() = delete;

        PowerTwoMemoryPage(rtsha_page* page) : MemoryPage(page)
        {
        }

        virtual ~PowerTwoMemoryPage()
        {
        }

        virtual void* allocate_block(const size_t& size) final;

        virtual void free_block(MemoryBlock& block) final;

        void createInitialFreeBlocks();

    };
}
```

4.18 SmallFixMemoryPage.h

```
#pragma once
#include <stdint.h>
#include "MemoryPage.h"

namespace rtsha
{
    using namespace std;


    class SmallFixMemoryPage : MemoryPage
    {
    public:

        SmallFixMemoryPage() = delete;

        explicit SmallFixMemoryPage(rtsha_page* page) : MemoryPage(page)
        {
        }

        virtual ~SmallFixMemoryPage()
        {
        }

        virtual void* allocate_block(const size_t& size) final;

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

Index