LwMEM

Tilen MAJERLE

Feb 16, 2020

CONTENTS

Welcome to the documentation for version latest-develop.

LwMEM is lightweight dynamic memory manager optimized for embedded systems.

Download library · *Getting started* · Open Github

CHAPTER

ONE

FEATURES

- Written in ANSI C99, compatible with `size_t` for size data types
- Implements standard C library functions for memory allocation, malloc, calloc, realloc and free
- Uses *first-fit* algorithm to search for free block
- Supports multiple allocation instances to split between memories and/or CPU cores
- Supports different memory regions to allow use of fragmented memories
- Highly configurable for memory allocation and reallocation
- Supports embedded applications with fragmented memories
- Supports automotive applications
- Supports advanced free/realloc algorithms to optimize memory usage
- Operating system ready, thread-safe API
- User friendly MIT license

CHAPTER

TWO

REQUIREMENTS

- C compiler
- Less than `2kB` of non-volatile memory

CHAPTER

THREE

CONTRIBUTE

Fresh contributions are always welcome. Simple instructions to proceed:

1. Fork Github repository
2. Respect C style & coding rules used by the library
3. Create a pull request to `develop` branch with new features or bug fixes

Alternatively you may:

1. Report a bug
2. Ask for a feature request

CHAPTER

FOUR

LICENSE

```
MIT License

Copyright (c) 2020 Tilen MAJERLE

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Getting started

5.1.1 Download library

Library is primarly hosted on Github.

- Download latest release from releases area on Github
- Clone *develop* branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

- Download and install `git` if not already
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available `3` options
  - Run `git clone --recurse-submodules https://github.com/MaJerle/lwmem` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/lwmem` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch master https://github.com/MaJerle/lwmem` to clone *latest stable* branch, including submodules
- Navigate to `examples` directory and run favourite example

Update cloned to latest version

- Open console and navigate to path in the system where your resources repository is. Use command `cd your_path`
- Run `git pull origin master --recurse-submodules` command to pull latest changes and to fetch latest changes from submodules
- Run `git submodule foreach git pull origin master` to update & merge all submodules

Note: This is preferred option to use when you want to evaluate library and run prepared examples. Repository consists of multiple submodules which can be automatically downloaded when cloning and pulling changes from root repository.

5.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page.

- Copy `lwmem` folder to your project
- Add `lwmem/src/include` folder to *include path* of your toolchain
- Add source files from `lwmem/src/` folder to toolchain build
- Copy `lwmem/src/include/lwmem/lwmem_config_template.h` to project folder and rename it to `lwmem_config.h`
- Build the project

5.1.3 Configuration file

Library comes with template config file, which can be modified according to needs. This file shall be named `lwmem_config.h` and its default template looks like the one below:

Tip: Check *LwMEM Configuration* section for possible configuration settings

---

Listing 1: Config file template

```
/**
 * \file            lwmem_config_template.h
 * \brief           LwMEM configuration file
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
```

(continues on next page)

(continued from previous page)

```
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of LwMEM - Lightweight dynamic memory manager library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.3.0
 */
#ifndef LWMEM_HDR_CONFIG_H
#define LWMEM_HDR_CONFIG_H

/* Rename this file to "lwmem_config.h" for your application */

/*
 * Open "include/lwmem/lwmem_config_default.h" and
 * copy & replace here settings you want to change values
 */

/* After user configuration, call default config to merge config together */
#include "lwmem/lwmem_config_default.h"

#endif /* LWMEM_HDR_CONFIG_H */
```

5.1.4 Minimal example code

Run below example to test and verify library

Listing 2: Absolute minimum example

```
#include "lwmem/lwmem.h"

/* Create regions, address and length of regions */
static
lwmem_region_t regions[] = {
    /* Set start address and size of each region */
    { (void *)0x10000000, 0x00001000 },
    { (void *)0xA0000000, 0x00008000 },
    { (void *)0xC0000000, 0x00008000 },
};

/* Later in the initialization process */
/* Assign regions for manager */
lwmem_assignmem(regions, sizeof(regions) / sizeof(regions[0]));

/* Usage in program... */
```

(continues on next page)

(continued from previous page)

```

void* ptr;
/* Allocate 8 bytes of memory */
ptr = lwmem_malloc(8);
if (ptr != NULL) {
    /* Allocation successful */
}

/* Later... */
/* Free allocated memory when not used */
lwmem_free(ptr);
ptr = NULL;
/* .. or */
lwmem_free_s(&ptr);
```

5.2 User manual

5.2.1 How it works

This section shows different buffer corner cases and provides basic understanding how memory allocation works within firmware.

As it is already known, library supports multiple memory regions (or addresses) to allow multiple memory locations within embedded systems:

- Internal RAM memory
- External RAM memory
- Optional fragmented internal memory

For the sake of this understanding, application is using `3` regions

- Region `1` memory starts at `0x1000 0000` and is `0x0000 1000` bytes long
- Region `2` memory starts at `0xA000 0000` and is `0x0000 8000` bytes long
- Region `3` memory starts at `0xC000 0000` and is `0x0000 8000` bytes long

Note: Total size of memory used by application for memory manager is `0x0001 1000` bytes or `69 kB`. This is a sum of all `3` regions.

Example also assumes that:

- Size of any kind of pointer is `4-bytes`, `sizeof(any_pointer_type) = 4`
- Size of `size_t` type is `4-bytes`, `sizeof(size_t) = 4`

First step is to define custom regions and assign them to memory manager.

Listing 3: Definitions of different memory regions

```
#include "lwmem/lwmem.h"

/*
 * \brief           Define regions for memory manager
```

(continues on next page)

(continued from previous page)

```
 */
static
lwmem_region_t regions[] = {
    /* Set start address and size of each region */
    { (void *)0x10000000, 0x00001000 },
    { (void *)0xA0000000, 0x00008000 },
    { (void *)0xC0000000, 0x00008000 },
};

/* Later in the initialization process */
/* Assign regions for manager */
lwmem_assignmem(regions, sizeof(regions) / sizeof(regions[0]));
/* or */
lwmem_assignmem_ex(NULL, regions, sizeof(regions) / sizeof(regions[0]));
```

Note: Order of regions must be lower address first. Regions must not overlap with their sizes.

When calling *lwmem_assignmem*, manager prepares memory blocks and assigns default values.

Fig. 1: Default memory structure after initialization

Memory managers sets some default values, these are:

- All regions are connected through single linked list. Each member of linked list represents free memory slot
- Variable `Start block` is by default included in library and points to first free memory on the list
- Each region has `2 free slot` indicators
  - One at the end of each region. It takes `8 bytes` of memory:
    - Size of slot is set to `0` which *means no available memory*
    - Its next value points to next free slot in another region. Set to `NULL` if there is no free slot available anymore after and is *last region* indicator
  - One at the beginning of region. It also takes `8 bytes` of memory:
    - Size of slot is set to `region_size - 8`, ignoring size of last slot. Effective size of memory, application may allocate in region, is always for `2` meta slots less than region size, which means `max_app_malloc_size = region_size - 2 - 8 bytes`
    - Its next value points to end slot in the same region

When application tries to allocate piece of memory, library will check linked list of empty blocks until it finds first with sufficient size. If there is a block bigger than requested size, it will be marked as allocated and removed from linked list.

Note: Further optimizations are implemented, such as possibility to split block when requested size is smaller than empty block size is.

Fig. 2: Memory structure after first allocation

- Light red background slot indicates memory in use.

- All blocks marked in use have
  - `next` value is set to `NULL`
  - `size` value has MSB bit set to `1`, indicating block *is allocated* and the rest of bits represent size of block, including metadata size
  - If application asks for 8 bytes, fields are written as `next = 0x0000 0000` and `size = 0x8000 000F`
- `Start block` now points to free slot somewhere in the middle of region

Fig. 3: Step-by-step memory structure after multiple allocations and deallocations

Image shows only first region to simplify process. Same procedure applies to other regions too.

- `Case A`: Second block allocated. Remaining memory is now smaller and `Start block points` to it
- `Case B`: Third block allocated. Remaining memory is now smaller and `Start block points` to it
- `Case C`: Forth block allocated. Remaining memory is now smaller and `Start block points` to it
- `Case D`: Third block freed and added back to linked list of free slots.
- `Case E`: Forth block freed. Manager detects blocks before and after current are free and merges all to one big contiguous block
- `Case F`: First block freed. `Start block` points to it as it has been added back to linked list
- `Case G`: Second block freed. Manager detects blocks before and after current are free and merges all to one big contiguous block.
  - No any memory allocated anymore, regions are back to default state

Allocate at specific region

When memory allocation is in progress, LwMEM manager will start at first free block and will loop through all regions until first free block of sufficient size has been found. At this stage, application really does not have any control which region has been used for allocation.

Especially in the world of embedded systems, sometimes application uses external RAM device, which are by definition slower than internal one. Let's take an example below.

Fig. 4: Region definition with one internal and two external regions

And code example:

Listing 4: Region definition with one internal and two external regions

```
#include "lwmem/lwmem.h"

/*
 * \brief           Define regions for memory manager
 */
static
lwmem_region_t regions[] = {
    /* Set start address and size of each region */
    { (void *)0x10000000, 0x00001000 },
```

(continues on next page)

(continued from previous page)

```
    { (void *)0xA0000000, 0x00008000 },
    { (void *)0xC0000000, 0x00008000 },
};

/* Later in the initialization process */
/* Assign regions for manager */
lwmem_assignmem(regions, sizeof(regions) / sizeof(regions[0]));
/* or */
lwmem_assignmem_ex(NULL, regions, sizeof(regions) / sizeof(regions[0]));
```

For the sake of this example, let's say that:

- First region is in very fast internal RAM, coupled with CPU core * Application shall use this only for small chunks of memory, frequently used, not to disturb external RAM interface
- Second and third regions are used for bigger RAM blocks used less frequently and interface is not overloaded when used

Size of first region is `0x1000` bytes. When application tries to allocate (example) `512` bytes, it will find first free block in first region. However, application wants to use (if possible) external RAM for this size of allocation.

There is a way to specify in which region memory shall be allocated, using extended functions.

Listing 5: Allocate memory from specific region

```
#include "lwmem/lwmem.h"

/* Assignment has been done previously... */

/* ptr1 will be allocated in first free block */
/* ptr2 will be allocated from second region */
void* ptr1, *ptr2;

/* Allocate 8 bytes of memory in any region */
/* Use one of 2 options, both have same effect */
ptr1 = lwmem_malloc(8);
ptr1 = lwmem_malloc_ex(NULL, NULL, 8);

/* Allocate memory from specific region only */
/* Use second region */
ptr2 = lwmem_malloc_ex(NULL, &regions[1], 512);
```

Tip: Check *`lwmem_malloc_ex()`* for more information about parameters and return values

5.2.2 LwMEM instances

LwMEM architecture allows multiple instances, to completely isolate memory management between different memories. This may allow separation of memory management at hardware level with different security feature.

By default, LwMEM has single instance created at library level, called *default instance*. Default instance does need any special attention as it is embedded at library core, instead application has to assign memory regions for the instance.

Every instance has:

- Instance control block

- Multiple regions assigned to each instance

Note: Control block of default instance is already initialized by library core, hence it does not need any special attention at application layer.

Fig. 5: LwMEM internal architecture with control block

Picture above shows internal architecture of LwMEM. Control block holds info about first free block for allocation and other private data, such as mutex handle when operating system is in use.

Yellow part of the image shows customized, application-defined, regions, which must be manually assigned to the instance during application start-up.

Known example for assinging regions to LwMEM is shown below. Default instance is used, therefore no special attention needs to be added when assigning regions or allocating memory.

Listing 6: Definition and assignment of regions for default LwMEM instance

```
#include "lwmem/lwmem.h"

/*
 * \brief           Define regions for memory manager
 */
static
lwmem_region_t regions[] = {
    /* Set start address and size of each region */
    { (void *)0x10000000, 0x00001000 },
    { (void *)0xA0000000, 0x00008000 },
    { (void *)0xC0000000, 0x00008000 },
};

/* Later in the initialization process */
/* Assign regions for manager */
lwmem_assignmem(regions, sizeof(regions) / sizeof(regions[0]));
/* or */
lwmem_assignmem_ex(NULL, regions, sizeof(regions) / sizeof(regions[0]));
```

When application adds second LwMEM instance, then special functions with *_ex* must be used. These allow application to specify for which LwMEM instance specific operation is intended.

Tip: Check *lwmem_assignmem_ex()* description for more information about input parameters.

Listing 7: Definition and assignment of regions for custom LwMEM instance

```
#include "lwmem/lwmem.h"

/**
 * \brief           Custom LwMEM instance
 */
static
lwmem_t lw_custom;
```

(continues on next page)

(continued from previous page)

```
8
9  /*
10  * \brief           Define regions for memory manager
11  */
12 static
13 lwmem_region_t regions[] = {
14     /* Set start address and size of each region */
15     { (void *)0x10000000, 0x00001000 },
16     { (void *)0xA0000000, 0x00008000 },
17     { (void *)0xC0000000, 0x00008000 },
18 };
19
20 /* Later in the initialization process */
21 /* Assign regions for custom instance */
22 lwmem_assignmem_ex(&lw_custom, regions, sizeof(regions) / sizeof(regions[0]));
```

5.2.3 Reallocation algorithm

What makes this library different to others is its ability for memory re-allocation. This section explains how it works and how it achieves best performances and less memory fragmentation vs others.

Sometimes application uses variable length of memory, especially when number of (as an example) elements in not fully known in advance. For the sake of this example, application anticipates `12` numbers (*integers*) but may (due to unknown reason in some cases) receive more than this. If application needs to hold all received numbers, it may be necessary to:

- Option `1`: Increase memory block size using reallocations
- Option `2`: Use very big (do we know how big?) array, allocated statically or dynamically, which would hold all numbers at any time possible

---

Note: LwMEM has been optimized to handle well option `1`.

Application needs to define at least single region:

Listing 8: Memory region assignment

```
1  #include "lwmem/lwmem.h"
2
3  /* Define one region used by lwmem */
4  static unsigned char region_mem[128];
5
6  /*
7   * \brief           Define regions for memory manager
8   */
9  static
10 lwmem_region_t regions[] = {
11     /* Set start address and size of each region */
12     { region_mem, sizeof(region_mem) }
13 };
14
15 /* Later in the initialization process */
16 /* Assign regions for manager */
```

(continues on next page)

(continued from previous page)

```
lwmem_assignmem(regions, sizeof(regions) / sizeof(regions[0]));
lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 9: Memory region assignment output

```
B = Free block; A = Address of free block; S = Free size
Allocation available bytes:  120 bytes

B  0: A: 0x013CB160, S:    0, next B: 0x013CB520; Start block
B  1: A: 0x013CB520, S:  120, next B: 0x013CB598
B  2: A: 0x013CB598, S:    0, next B: 0x00000000; End of region
```

Note: Please check *How it works* section for more information

After region has been defined, application tries to allocate memory for `12 integers`.

Listing 10: First memory allocation

```
int* ints = lwmem_malloc(12 * sizeof(*ints));  /* Allocate memory for 12 integers */

/* Check for successful allocation */
if (ints == NULL) {
    printf("Allocation failed!\r\n");
    return -1;
}
lwmem_debug_free();     /* This is debug function for sake of this example */

/* ints is a pointer to memory size for our integers */
/* Do not forget to free it when not used anymore */
lwmem_free_s(&ints);
```

When executed on test machine, it prints:

Listing 11: First memory allocation output

```
B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   64 bytes

B  0: A: 0x013CB160, S:    0, next B: 0x013CB558; Start block
B  1: A: 0x013CB558, S:   64, next B: 0x013CB598
B  2: A: 0x013CB598, S:    0, next B: 0x00000000; End of region
```

At first, manager had `120` bytes of available memory while after allocation of `48` bytes, it only left `64` bytes. Effectively `120 - 64 = 56` bytes have been used to allocate `48` bytes of memory.

Note: Every allocated block holds meta data. On test machine, `sizeof(int) = 4` therefore `8` bytes are used for metadata as `56 - 12 * sizeof(int) = 8`. Size of meta data header depends on CPU architecture and may be different between architectures

Application got necessary memory for `12` integers. How to proceed when application needs to extend size for one more integer?

Easiest would be to:

1. Allocate new memory block with new size and check if allocation was successful
2. Manually copy content from old block to new block
3. Free old memory block
4. Use new block for all future operations

Here is the code:

Listing 12: Custom reallocation

```
int* ints = lwmem_malloc(12 * sizeof(*ints));  /* Allocate memory for 12 integers */

/* Check for successful allocation */
if (ints == NULL) {
    printf("Allocation failed ints!\r\n");
    return -1;
}
printf("ints allocated for 12 integers\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Now allocate new one for new size */
int* ints2 = lwmem_malloc(13 * sizeof(*ints)); /* Allocate memory for 13 integers */
if (ints2 == NULL) {
    printf("Allocation failed ints2!\r\n");
    return -1;
}

printf("ints2 allocated for 13 integers\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Copy content of 12-integers to 13-integers long array */
memcpy(ints2, ints, 12 * sizeof(12));

/* Free first block */
lwmem_free(ints);       /* Free memory */
ints = ints2;           /* Use ints2 as new array now */
ints2 = NULL;           /* Set it to NULL to prevent accessing same memory from␣
↪different pointers */

printf("old ints freed\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Do not forget to free it when not used anymore */
lwmem_free_s(&ints);

printf("ints and ints2 freed\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 13: Custom reallocation output

```
ints allocated for 12 integers

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   64 bytes
```

(continues on next page)

(continued from previous page)

```
B  0: A: 0x00B5B160, S:    0, next B: 0x00B5B558; Start block
B  1: A: 0x00B5B558, S:   64, next B: 0x00B5B598
B  2: A: 0x00B5B598, S:    0, next B: 0x00000000; End of region

ints2 allocated for 13 integers

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:    0 bytes

B  0: A: 0x00B5B160, S:    0, next B: 0x00B5B598; Start block
B  1: A: 0x00B5B598, S:    0, next B: 0x00000000; End of region

old ints freed

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   56 bytes

B  0: A: 0x00B5B160, S:    0, next B: 0x00B5B520; Start block
B  1: A: 0x00B5B520, S:   56, next B: 0x00B5B598
B  2: A: 0x00B5B598, S:    0, next B: 0x00000000; End of region

ints and ints2 freed

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:  120 bytes

B  0: A: 0x00B5B160, S:    0, next B: 0x00B5B520; Start block
B  1: A: 0x00B5B520, S:  120, next B: 0x00B5B598
B  2: A: 0x00B5B598, S:    0, next B: 0x00000000; End of region
```

Outcome of the debug messages:

1. Memory was successfully allocated for `12` integers, it took `56` bytes
2. Memory was successfully allocated for another `13` integers , it took `64` bytes
3. There is no more free memory available
4. First `12` integers array was successfully freed, manager has `56` bytes of free memory
5. Second `13` integers block was successfully freed, manager has all `120` bytes available for new allocations

This was therefore successful custom reallocation from `12` to `13` integers. Next step is to verify what would happen when application wants to reallocate to `15` integers instead. When same code is executed (but with `15` instead of `12`), it prints:

Listing 14: Custom reallocation for 15 integers

```
ints allocated for 12 integers

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   64 bytes

B  0: A: 0x00D2B160, S:    0, next B: 0x00D2B558; Start block
B  1: A: 0x00D2B558, S:   64, next B: 0x00D2B598
B  2: A: 0x00D2B598, S:    0, next B: 0x00000000; End of region

Allocation failed ints2!
```

Oooops! It is not anymore possible to allocate new block for new `15` integers as there was no available block with at least `15 * sizeof(int) + metadata_size` bytes of free memory.

Note: With this reallocation approach, maximal size of application block is only `50%` of region size. This is not the most effective memory manager!

Fortunately there is a solution. Every time application wants to resize existing block, manager tries to manipulate existing block and shrink or expand it.

Shrink existing block

Easiest reallocation algorithm is when application wants to decrease size of previously allocated memory. When this is the case, manager only needs to change the size of existing block to lower value.

Listing 15: Shrink existing block to smaller size

```
int* ints, *ints2;

ints = lwmem_malloc(15 * sizeof(*ints));  /* Allocate memory for 15 integers */
if (ints == NULL) {
    printf("Allocation failed ints!\r\n");
    return -1;
}
printf("ints allocated for 15 integers\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Now reallocte ints and write result to new variable */
ints2 = lwmem_realloc(ints, 12 * sizeof(*ints));
if (ints == NULL) {
    printf("Allocation failed ints2!\r\n");
    return -1;
}
printf("ints re-allocated for 12 integers\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* ints is successfully reallocated and it is no longer valid pointer to read/write␣
 ↪from/to */

/* For the sake of example, let's test pointers */
if (ints2 == ints) {
        printf("New block reallocated to the same address as previous one\r\n");
} else {
        printf("New block reallocated to new address\r\n");
}

/* Free ints2 */
lwmem_free_s(&ints2);
/* ints is already freed by successful realloc function */
ints = NULL;            /* It is enough to set it to NULL */

lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 16: Shrink existing block to smaller size output

```
ints allocated for 15 integers

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   52 bytes

B  0: A: 0x00B6B160, S:    0, next B: 0x00B6B564; Start block
B  1: A: 0x00B6B564, S:   52, next B: 0x00B6B598
B  2: A: 0x00B6B598, S:    0, next B: 0x00000000; End of region

ints re-allocated for 12 integers

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   64 bytes

B  0: A: 0x00B6B160, S:    0, next B: 0x00B6B558; Start block
B  1: A: 0x00B6B558, S:   64, next B: 0x00B6B598
B  2: A: 0x00B6B598, S:    0, next B: 0x00000000; End of region

New block reallocated to the same address as previous one

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:  120 bytes

B  0: A: 0x00B6B160, S:    0, next B: 0x00B6B520; Start block
B  1: A: 0x00B6B520, S:  120, next B: 0x00B6B598
B  2: A: 0x00B6B598, S:    0, next B: 0x00000000; End of region
```

Outcome of our reallocation:

- Memory was successfully allocated for `15` integers, it took `68` bytes; part `A` on image
- Memory was successfully re-allocated to `12` integers, now it takes `56` bytes, part `B` on image
- In both cases on image, final returned memory points to the same address
  - Manager does not need to copy data from existing memory to new address as it is the same memory used in both cases
- Empty block start address has been modified and its size has been increased, part `B` on image
- Reallocated block was successfully freed, manager has all `120` bytes for new allocations

Tip: This was a success now, much better.

It is not always possible to increase block size of next free block on linked list. Consider new example and dedicated image below.

Fig. 6: Shrinking fragmented memory block

Listing 17: Shrink fragmented memory block

```
1 void* ptr1, *ptr2, *ptr3, *ptr4, *ptrt;
2
3 /* We are now at case A */
```

(continues on next page)

(continued from previous page)

```
printf("State at case A\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Each ptr points to its own block of allocated data */
/* Each block size is 24 bytes; 16 for user data and 8 for metadata */
ptr1 = lwmem_malloc(16);
ptr2 = lwmem_malloc(16);
ptr3 = lwmem_malloc(16);
ptr4 = lwmem_malloc(16);

/* We are now at case B */
printf("State at case B\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Reallocate ptr1, decrease its size to 12 user bytes */
/* Now we expect block size to be 20; 12 for user data and 8 for metadata */
ptrt = lwmem_realloc(ptr1, 12);
if (ptrt == NULL) {
    ptr1 = ptrt;
}

printf("State after first realloc\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* At this point we are still at case B */
/* There was no modification of internal structure */
/* Difference between existing and new size (16 - 12 = 4) is too small
    to create new empty block, therefore block it is left unchanged */

/* Reallocate again, now to new size of 4 bytes */
/* Now we expect block size to be 16; 8 for user data and 8 for metadata */
ptrt = lwmem_realloc(ptr1, 8);
printf("State at case C\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* We are now at case C */

/* Now free all memories */
```

When executed on test machine, it prints:

Listing 18: Shrink fragmented memory block output

```
State at case A

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:  120 bytes

B  0: A: 0x00E7B160, S:    0, next B: 0x00E7B520; Start block
B  1: A: 0x00E7B520, S:  120, next B: 0x00E7B598
B  2: A: 0x00E7B598, S:    0, next B: 0x00000000; End of region

State at case B

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   24 bytes
```

(continues on next page)

(continued from previous page)

```
B  0: A: 0x00E7B160, S:    0, next B: 0x00E7B580; Start block
B  1: A: 0x00E7B580, S:   24, next B: 0x00E7B598
B  2: A: 0x00E7B598, S:    0, next B: 0x00000000; End of region

State after first realloc

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   24 bytes

B  0: A: 0x00E7B160, S:    0, next B: 0x00E7B580; Start block
B  1: A: 0x00E7B580, S:   24, next B: 0x00E7B598
B  2: A: 0x00E7B598, S:    0, next B: 0x00000000; End of region

State at case C

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   32 bytes

B  0: A: 0x00E7B160, S:    0, next B: 0x00E7B530; Start block
B  1: A: 0x00E7B530, S:    8, next B: 0x00E7B580
B  2: A: 0x00E7B580, S:   24, next B: 0x00E7B598
B  3: A: 0x00E7B598, S:    0, next B: 0x00000000; End of region
```

Outcome of this example:

- Size of all `4` blocks is `24` bytes; `16` for user data, `8` for metadata
- Reallocating block first time from `16` to `12` user data bytes did not affect internal memory structure
  - It is not possible to create new empty block as it would be too small, only `4` bytes available, minimum is `8` bytes for meta data
  - It is not possible to enlarge next empty block due to *current* and *next empty* do not create contiguous block
  - Block is internally left unchanged
- Reallocating block second time to `8` bytes was a success
  - Difference between old and new size is `8` bytes which is enough for new empty block
    - Its size is `8` bytes, effectively `0` for user data due to meta size

Shrink existing block - summary

When reallocating already allocated memory block, one of *3* cases will happen:

- Case `1`: When *current* block and *next free* block could create contigouos block of memory, *current* block is decreased (size parameter) and *next free* is enlarged by the size difference
- Case `2`: When difference between *current* size and *new* size is more or equal to minimal size for new empty block, new empty block is created with size `current_size - new_size` and added to list of free blocks
- Case `3`: When difference between *current* size and *new* size is less than minimal size for new empty block, block is left unchanged

Enlarge existing block

Now that you master procedure to shrink (or decrease) size of existing allocated memory block, it is time to understand how to enlarge it. Things here are more complicated, however, they are still easy to understand.

Manager covers 3 potential cases:

- Case 1: Increase size of currently allocated block
- Case 2: Merge previous empty block with existing one and shift data up
- Case 3: Block before and after existing block together create contiguous block of memory

Free block after + *allocated block* create one big contiguous block

Fig. 7: *Free block* after + *allocated block* create one big contiguous block

Listing 19: Enlarge existing block

```
void* ptr1, *ptr2;

/* Allocate initial block */
ptr1 = lwmem_malloc(24);

/* We assume allocation is successful */

printf("State at case 1a\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Now let's reallocate ptr1 */
ptr2 = lwmem_realloc(ptr1, 32);

printf("State at case 1b\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 20: Enlarge existing block output

```
State at case 1a

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   88 bytes

B  0: A: 0x00CBB160, S:    0, next B: 0x00CBB540; Start block
B  1: A: 0x00CBB540, S:   88, next B: 0x00CBB598
B  2: A: 0x00CBB598, S:    0, next B: 0x00000000; End of region

State at case 1b

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   80 bytes

B  0: A: 0x00CBB160, S:    0, next B: 0x00CBB548; Start block
B  1: A: 0x00CBB548, S:   80, next B: 0x00CBB598
B  2: A: 0x00CBB598, S:    0, next B: 0x00000000; End of region
```

- Allocation for first block of memory (`24` user bytes) uses `32` bytes of data
- Reallocation is successful, block has been extended to `40` bytes and next free block has been shrinked down to `80` bytes

Free block before + *allocated block* create one big contiguous block

Fig. 8: *Free block* before + *allocated block* create one big contiguous block

Listing 21: Enlarge existing block

```
void* ptr1, *ptr2;

/* Allocate initial blocks */
ptr2 = lwmem_malloc(80);
ptr1 = lwmem_malloc(24);
lwmem_free_s(&ptr2);    /* Free first block and mark it free */

/* We assume allocation is successful */

printf("State at case 2a\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */

/* Now let's reallocate ptr1 */
ptr2 = lwmem_realloc(ptr1, 32);

printf("State at case 2b\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 22: Enlarge existing block output

```
State at case 2a

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   88 bytes

B  0: A: 0x0135B160, S:    0, next B: 0x0135B520; Start block
B  1: A: 0x0135B520, S:   88, next B: 0x0135B598
B  2: A: 0x0135B598, S:    0, next B: 0x00000000; End of region

State at case 2b

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   80 bytes

B  0: A: 0x0135B160, S:    0, next B: 0x0135B548; Start block
B  1: A: 0x0135B548, S:   80, next B: 0x0135B598
B  2: A: 0x0135B598, S:    0, next B: 0x00000000; End of region
```

- First application allocates big block (`88` bytes), followed by smaller block (`32` bytes)
- Application then frees big block to mark it as free. This is effectively state `2a`

- During reallocation, manager did not find suitable block after *current* block, but it found suitable block before *current* block:
  - Empty block and allocated block are temporary merged to one big block (`120` bytes)
  - Content of allocated block is shifted up to beginning of new big block
  - Big block is then splitted to required size, the rest is marked as free
- This is effectively state `2b`

Free block before + *free block* after + *allocated block* create one big contiguous block

When application makes many allocations and frees of memory, there is a high risk of memory fragmentations. Essentially small chunks of allocated memory prevent manager to allocate new, fresh, big block of memory.

When it comes to reallocating of existing block, it may happen that *first free block after* and *current block* create a contiguous block, but its combined size is not big enough. Same could happen with *last block before* + *current block*. However, it may be possible to combine *free block before* + *current block* + *free block after* current block together.

Fig. 9: *Free block* before + *free block* after + *allocated block* create one big contiguous block

In this example manager has always `2` allocated blocks and application always wants to reallocate `green` block. `Red` block is acting as an obstacle to show different application use cases.

Note: Image shows `4` use cases. For each of them, case labeled with `3` is initial state.

Initial state `3` is generated using C code:

Listing 23: Initial state of blocks within memory

```
void* ptr1, *ptr2, *ptr3, *ptr4;

/* Allocate 4 blocks */
ptr1 = lwmem_malloc(8);
ptr2 = lwmem_malloc(4);
ptr3 = lwmem_malloc(4);
ptr4 = lwmem_malloc(16);
/* Free first and third block */
lwmem_free_s(&ptr1);
lwmem_free_s(&ptr3);

/* We assume allocation is successful */

printf("Initial state at case 3\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

When executed on test machine, it prints:

Listing 24: Initial state of blocks within memory output

```
Initial state at case 3

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   84 bytes
```

(continues on next page)

(continued from previous page)

```
B  0: A: 0x013BB160, S:    0, next B: 0x013BB520; Start block
B  1: A: 0x013BB520, S:   16, next B: 0x013BB53C
B  2: A: 0x013BB53C, S:   12, next B: 0x013BB560
B  3: A: 0x013BB560, S:   56, next B: 0x013BB598
B  4: A: 0x013BB598, S:    0, next B: 0x00000000; End of region
```

Tip: Image shows (and log confirms) `3` free slots of `16, 12 and 56` bytes in size respectively.

- Case `3a`: Application tries to reallocate green block from `12` to `16` bytes
  - Reallocation is successful, there is a free block just after and green block is successfully enlarged
  - Block after is shrinked from `12` to `8` bytes
  - Code example (follows initial state code example)

Listing 25: Enlarge of existing block for case 3A

```
/* Now reallocate ptr2 */
ptr2 = lwmem_realloc(ptr2, 8);

printf("New state at case 3a\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

  - When executed on test machine, it prints:

Listing 26: Enlarge of existing block for case 3A output

```
New state at case 3a

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   80 bytes

B  0: A: 0x0133B160, S:    0, next B: 0x0133B5C0; Start block
B  1: A: 0x0133B5C0, S:   16, next B: 0x0133B5E0
B  2: A: 0x0133B5E0, S:    8, next B: 0x0133B600
B  3: A: 0x0133B600, S:   56, next B: 0x0133B638
B  4: A: 0x0133B638, S:    0, next B: 0x00000000; End of region
```

- Case `3b`: Application tries to reallocate green block from `12` to `28` bytes
  - Block after green is not big enough to merge them to one block (`12 + 12 < 28`)
  - Block before green is big enough (`16 + 12 >= 28`)
  - Green block is merged with previous free block and content is shifted to the beginning of new block

Listing 27: Enlarge of existing block for case 3B

```
/* Now reallocate ptr2 */
ptr2 = lwmem_realloc(ptr2, 20);

printf("New state at case 3b\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

  - When executed on test machine, it prints:

Listing 28: Enlarge of existing block for case 3B output

```
New state at case 3b

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   68 bytes

B  0: A: 0x0103B160, S:    0, next B: 0x0103B53C; Start block
B  1: A: 0x0103B53C, S:   12, next B: 0x0103B560
B  2: A: 0x0103B560, S:   56, next B: 0x0103B598
B  3: A: 0x0103B598, S:    0, next B: 0x00000000; End of region
```

- Case `3c`: Application tries to reallocate green block from `12` to `32` bytes
  - Block after green is not big enough to merge them to one block (`12 + 12 < 32`)
  - Block before green is also not big enough (`12 + 16 < 32`)
  - All three blocks together are big enough (`16 + 12 + 12 >= 32`)
  - All blocks are effectively merged together and there is a new temporary block with its size set to `40` bytes
  - Content of green block is shifted to the beginning of new block
  - New block is limited to `32` bytes, keeping `8` bytes marked as free at the end

Listing 29: Enlarge of existing block for case 3C

```
/* Now reallocate ptr2 */
ptr2 = lwmem_realloc(ptr2, 24);

printf("New state at case 3c\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

  - When executed on test machine, it prints:

Listing 30: Enlarge of existing block for case 3C output

```
New state at case 3c

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   64 bytes

B  0: A: 0x011AB160, S:    0, next B: 0x011AB540; Start block
B  1: A: 0x011AB540, S:    8, next B: 0x011AB560
B  2: A: 0x011AB560, S:   56, next B: 0x011AB598
B  3: A: 0x011AB598, S:    0, next B: 0x00000000; End of region
```

- Case `3d`: Application tries to reallocate green block from `12` to `44` bytes
  - None of the methods (`3a - 3c`) are available as blocks are too small
  - Completely new block is created and content is copied to it
  - Existing block is marked as free. All `3` free blocks create big contiguous block, they are merged to one block with its size set to `40`

Listing 31: Enlarge of existing block for case 3D

```
/* Now reallocate ptr2 */
ptr2 = lwmem_realloc(ptr2, 36);

printf("New state at case 3d\r\n");
lwmem_debug_free();     /* This is debug function for sake of this example */
```

- When executed on test machine, it prints:

Listing 32: Enlarge of existing block for case 3D output

```
New state at case 3d

B = Free block; A = Address of free block; S = Free size
Allocation available bytes:   52 bytes

B  0: A: 0x002EB160, S:    0, next B: 0x002EB520; Start block
B  1: A: 0x002EB520, S:   40, next B: 0x002EB58C
B  2: A: 0x002EB58C, S:   12, next B: 0x002EB598
B  3: A: 0x002EB598, S:    0, next B: 0x00000000; End of region
```

Full test code with assert

Advanced debugging features have been added for development purposes. It is now possible to simulate different cases within single executable, by storing states to different memories.

Example has been implemented for WIN32 and relies on dynamic allocation using `malloc` standard C function for main block data preparation.

How it works:

- Code prepares state 3 and saves memory to temporary memory for future restore
- Code restores latest saved state (case `3`) and executes case `3a`
- Code restores latest saved state (case `3`) and executes case `3b`
- Code restores latest saved state (case `3`) and executes case `3c`
- Code restores latest saved state (case `3`) and executes case `3d`

Initial state `3` is generated using C code:

Listing 33: Full test code with asserts

```
#define ASSERT(x)           do {        \
    if (!(x)) {                         \
        printf("Assert failed with condition (" # x ")\r\n");  \
    } else {\
        printf("Assert passed with condition (" # x ")\r\n");  \
    }\
} while (0)

/* For debug purposes */
lwmem_region_t* regions_used;
size_t regions_count = 1;       /* Use only 1 region for debug purposes of non-free
↪areas */
```

(continues on next page)

(continued from previous page)

```c

int
main(void) {
    uint8_t* ptr1, *ptr2, *ptr3, *ptr4;
    uint8_t* rptr1, *rptr2, *rptr3, *rptr4;

    /* Create regions for debug purpose */
    if (!lwmem_debug_create_regions(&regions_used, regions_count, 128)) {
        printf("Cannot allocate memory for regions for debug purpose!\r\n");
        return -1;
    }
    lwmem_assignmem(regions_used, regions_count);
    printf("Manager is ready!\r\n");
    lwmem_debug_print(1, 1);

    /* Test case 1, allocate 3 blocks, each of different size */
    /* We know that sizeof internal metadata block is 8 bytes on win32 */
    printf("\r\n\r\nAllocating 4 pointers and freeing first and third..\r\n");
    ptr1 = lwmem_malloc(8);
    ptr2 = lwmem_malloc(4);
    ptr3 = lwmem_malloc(4);
    ptr4 = lwmem_malloc(16);
    lwmem_free(ptr1);               /* Free but keep value for future comparison */
    lwmem_free(ptr3);               /* Free but keep value for future comparison */
    lwmem_debug_print(1, 1);
    printf("Debug above is effectively state 3\r\n");
    lwmem_debug_save_state();       /* Every restore operations rewinds here */

    /* We always try to reallocate pointer ptr2 */

    /* Create 3a case */
    printf("\r\n------------------------------------------------------------------------
↪--\r\n");
    lwmem_debug_restore_to_saved();
    printf("State 3a\r\n");
    rptr1 = lwmem_realloc(ptr2, 8);
    lwmem_debug_print(1, 1);
    ASSERT(rptr1 == ptr2);

    /* Create 3b case */
    printf("\r\n------------------------------------------------------------------------
↪--\r\n");
    lwmem_debug_restore_to_saved();
    printf("State 3b\r\n");
    rptr2 = lwmem_realloc(ptr2, 20);
    lwmem_debug_print(1, 1);
    ASSERT(rptr2 == ptr2);

    /* Create 3c case */
    printf("\r\n------------------------------------------------------------------------
↪--\r\n");
    lwmem_debug_restore_to_saved();
    printf("State 3c\r\n");
    rptr3 = lwmem_realloc(ptr2, 24);
    lwmem_debug_print(1, 1);
    ASSERT(rptr3 == ptr1);

```

(continues on next page)

(continued from previous page)

```
    /* Create 3d case */
    printf("\r\n----------------------------------------------------------------------
↪--\r\n");
    lwmem_debug_restore_to_saved();
    printf("State 3d\r\n");
    rptr4 = lwmem_realloc(ptr2, 36);
    lwmem_debug_print(1, 1);
    ASSERT(rptr4 != ptr1 && rptr4 != ptr2 && rptr4 != ptr3 && rptr4 != ptr4);

    return 0;
}
```

When executed on test machine, it prints:

Listing 34: Full test code with asserts output

```
Manager is ready!
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      1 |  120 |              112 |Free block      |
|     2 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|


Allocating 4 pointers and freeing first and third..
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      1 |   16 |                8 |Free block      |
|     2 | 0034A518 |      0 |   12 |                0 |Allocated block |
|     3 | 0034A524 |      1 |   12 |                4 |Free block      |
|     4 | 0034A530 |      0 |   24 |                0 |Allocated block |
|     5 | 0034A548 |      1 |   56 |               48 |Free block      |
|     6 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|
Debug above is effectively state 3
 -- > Current state saved!

----------------------------------------------------------------------
 -- > State restored to last saved!
State 3a
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      1 |   16 |                8 |Free block      |
|     2 | 0034A518 |      0 |   16 |                0 |Allocated block |
|     3 | 0034A528 |      1 |    8 |                0 |Free block      |
|     4 | 0034A530 |      0 |   24 |                0 |Allocated block |
|     5 | 0034A548 |      1 |   56 |               48 |Free block      |
|     6 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|
Assert passed with condition (rptr1 == ptr2)
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------
 -- > State restored to last saved!
State 3b
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      0 |   28 |                0 |Allocated block |
|     2 | 0034A524 |      1 |   12 |                4 |Free block      |
|     3 | 0034A530 |      0 |   24 |                0 |Allocated block |
|     4 | 0034A548 |      1 |   56 |               48 |Free block      |
|     5 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|
Assert failed with condition (rptr2 == ptr2)

--------------------------------------------------------------------------
 -- > State restored to last saved!
State 3c
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      0 |   32 |                0 |Allocated block |
|     2 | 0034A528 |      1 |    8 |                0 |Free block      |
|     3 | 0034A530 |      0 |   24 |                0 |Allocated block |
|     4 | 0034A548 |      1 |   56 |               48 |Free block      |
|     5 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|
Assert passed with condition (rptr3 == ptr1)

--------------------------------------------------------------------------
 -- > State restored to last saved!
State 3d
|-------|----------|--------|------|------------------|----------------|
| Block | Address  | IsFree | Size | MaxUserAllocSize | Meta           |
|-------|----------|--------|------|------------------|----------------|
|     0 | 00179154 |      0 |    0 |                0 |Start block     |
|     1 | 0034A508 |      1 |   40 |               32 |Free block      |
|     2 | 0034A530 |      0 |   24 |                0 |Allocated block |
|     3 | 0034A548 |      0 |   44 |                0 |Allocated block |
|     4 | 0034A574 |      1 |   12 |                4 |Free block      |
|     5 | 0034A580 |      0 |    0 |                0 |End of region   |
|-------|----------|--------|------|------------------|----------------|
Assert passed with condition (rptr4 != ptr1 && rptr4 != ptr2 && rptr4 != ptr3 &&␣
↪rptr4 != ptr4)
```

5.2.4 Thread safety

With default configuration, LwMEM library is *not* thread safe. This means whenever it is used with operating system, user must resolve it with care.

Library has locking mechanism support for thread safety, which needs to be enabled.

Tip: To enable thread-safety support, parameter `LWMEM_CFG_OS` must be set to `1`. Please check *LwMEM Configuration* for more information about other options.

After thread-safety features has been enabled, it is necessary to implement `4` low-level system functions.

Tip: System function template example is available in `lwmem/src/system/` folder.

Example code for `CMSIS-OS V2`

Note: Check *System functions* section for function description

Listing 35: System function implementation for CMSIS-OS based operating systems

```
/**
 * \file            lwmem_sys_cmsis_os.c
 * \brief           System functions for CMSIS-OS based operating system
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of LwMEM - Lightweight dynamic memory manager library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.3.0
```

(continues on next page)

(continued from previous page)

```
 */
#include "system/lwmem_sys.h"

#if LWMEM_CFG_OS && !__DOXYGEN__

#include "cmsis_os.h"

uint8_t
lwmem_sys_mutex_create(LWMEM_CFG_OS_MUTEX_HANDLE* m) {
    *m = osMutexNew(NULL);
    return 1;
}

uint8_t
lwmem_sys_mutex_isvalid(LWMEM_CFG_OS_MUTEX_HANDLE* m) {
    return *m != NULL;
}

uint8_t
lwmem_sys_mutex_wait(LWMEM_CFG_OS_MUTEX_HANDLE* m) {
    if (osMutexAcquire(*m, osWaitForever) != osOK) {
        return 0;
    }
    return 1;
}

uint8_t
lwmem_sys_mutex_release(LWMEM_CFG_OS_MUTEX_HANDLE* m) {
    if (osMutexRelease(*m) != osOK) {
        return 0;
    }
    return 1;
}

#endif /* LWMEM_CFG_OS && !__DOXYGEN__ */
```

5.3 API reference

List of all the modules:

5.3.1 LwMEM

group **LWMEM**
Lightweight dynamic memory manager.

Defines

LWMEM_ARRAYSIZE(x)
Get size of statically allocated array.

Return Number of elements in array

Parameters

- `[in] x`: Object to get array size of

lwmem_assignmem(regions, len) **Note**
This is a wrapper for *lwmem_assignmem_ex* function

Parameters

- `[in] regions`: Array of regions with address and its size. Regions must be in increasing order (start address) and must not overlap in-between
- `[in] len`: Number of regions in array

lwmem_malloc(size) **Note**
This is a wrapper for *lwmem_malloc_ex* function. It operates in default LwMEM instance and uses first available region for memory operations

Parameters

- `[in] size`: Size to allocate in units of bytes

lwmem_calloc(nitems, size) **Note**
This is a wrapper for *lwmem_calloc_ex* function. It operates in default LwMEM instance and uses first available region for memory operations

Parameters

- `[in] nitems`: Number of elements to be allocated
- `[in] size`: Size of each element, in units of bytes

lwmem_realloc(ptr, size) **Note**
This is a wrapper for *lwmem_realloc_ex* function

Parameters

- `[in] ptr`: Memory block previously allocated with one of allocation functions. It may be set to `NULL` to create new clean allocation
- `[in] size`: Size of new memory to reallocate

lwmem_realloc_s(ptrptr, size) **Note**
This is a wrapper for *lwmem_realloc_s_ex* function

Parameters

- `[in] ptrptr`: Pointer to pointer to allocated memory. Must not be set to `NULL`. If reallocation is successful, it modified where pointer points to, or sets it to `NULL` in case of `free` operation
- `[in] size`: New requested size

lwmem_free (ptr) **Note**

This is a wrapper for *lwmem_free_ex* function

Parameters

- `[in] ptr`: Memory to free. `NULL` pointer is valid input

lwmem_free_s (ptrptr) **Note**

This is a wrapper for *lwmem_free_s_ex* function

Parameters

- `[in] ptrptr`: Pointer to pointer to allocated memory. When set to non `NULL`, pointer is freed and set to `NULL`

Functions

size_t **lwmem_assignmem_ex** (*lwmem_t* ***const** *lw*, **const** *lwmem_region_t* **regions*, **const** size_t *len*)

Initializes and assigns user regions for memory used by allocator algorithm.

Return `0` on failure, number of final regions used for memory manager on success

Note This function is not thread safe when used with operating system. It must be called only once to setup memory regions

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance
- `[in] regions`: Array of regions with address and its size. Regions must be in increasing order (start address) and must not overlap in-between
- `[in] len`: Number of regions in array

void ***lwmem_malloc_ex** (*lwmem_t* ***const** *lw*, **const** *lwmem_region_t* **region*, **const** size_t *size*)

Allocate memory of requested size in specific lwmem instance and optional region.

Note This is an extended malloc version function declaration to support advanced features

Return Pointer to allocated memory on success, `NULL` otherwise

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance
- `[in] region`: Optional region instance within LwMEM instance to force allocation from. Set to `NULL` to use any region within LwMEM instance
- `[in] size`: Number of bytes to allocate

void ***lwmem_calloc_ex** (*lwmem_t* ***const** *lw*, **const** *lwmem_region_t* **region*, **const** size_t *nitems*, **const** size_t *size*)

Allocate contiguous block of memory for requested number of items and its size in specific lwmem instance and region.

It resets allocated block of memory to zero if allocation is successful

Note This is an extended calloc version function declaration to support advanced features

Return Pointer to allocated memory on success, `NULL` otherwise

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance
- `[in] region`: Optional region instance within LwMEM instance to force allocation from. Set to `NULL` to use any region within LwMEM instance
- `[in] nitems`: Number of elements to be allocated
- `[in] size`: Size of each element, in units of bytes

void *__lwmem_realloc_ex__(*lwmem_t* *__const__ *lw*, __const__ *lwmem_region_t* **region*, void *__const__ *ptr*, __const__ size_t *size*)

Reallocates already allocated memory with new size in specific lwmem instance and region.

Function behaves differently, depends on input parameter of `ptr` and `size`:

Note This function may only be used with allocations returned by any of `_from` API functions

- `ptr == NULL; size == 0`: Function returns `NULL`, no memory is allocated or freed
- `ptr == NULL; size > 0`: Function tries to allocate new block of memory with `size` length, equivalent to `malloc(region, size)`
- `ptr != NULL; size == 0`: Function frees memory, equivalent to `free(ptr)`
- `ptr != NULL; size > 0`: Function tries to allocate new memory of copy content before returning pointer on success

Return Pointer to allocated memory on success, `NULL` otherwise

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance
- `[in] region`: Pointer to region to allocate from. Set to `NULL` to use any region within LwMEM instance. Instance must be the same as used during allocation procedure
- `[in] ptr`: Memory block previously allocated with one of allocation functions. It may be set to `NULL` to create new clean allocation
- `[in] size`: Size of new memory to reallocate

unsigned char __lwmem_realloc_s_ex__(*lwmem_t* *__const__ *lw*, __const__ *lwmem_region_t* **region*, void **__const__ *ptr*, __const__ size_t *size*)

Safe version of realloc_ex function.

After memory is reallocated, input pointer automatically points to new memory to prevent use of dangling pointers. When reallocation is not successful, original pointer is not modified and application still has control of it.

It is advised to use this function when reallocating memory.

Function behaves differently, depends on input parameter of `ptr` and `size`:

- `ptr == NULL`: Invalid input, function returns `0`
- `*ptr == NULL; size == 0`: Function returns `0`, no memory is allocated or freed

- `*ptr == NULL; size > 0`: Function tries to allocate new block of memory with `size` length, equivalent to `malloc(size)`
- `*ptr != NULL; size == 0`: Function frees memory, equivalent to `free(ptr)`, sets input pointer pointing to `NULL`
- `*ptr != NULL; size > 0`: Function tries to reallocate existing pointer with new size and copy content to new block

Return `1` if successfully reallocated, `0` otherwise

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance
- `[in] region`: Pointer to region to allocate from. Set to `NULL` to use any region within LwMEM instance. Instance must be the same as used during allocation procedure
- `[in] ptr`: Pointer to pointer to allocated memory. Must not be set to `NULL`. If reallocation is successful, it modified where pointer points to, or sets it to `NULL` in case of `free` operation
- `[in] size`: New requested size

void **lwmem_free_ex**(*lwmem_t* ***const** *lw*, void ***const** *ptr*)
Free previously allocated memory using one of allocation functions in specific lwmem instance.

Note This is an extended free version function declaration to support advanced features

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance. Instance must be the same as used during allocation procedure

Parameters

- `[in] ptr`: Memory to free. `NULL` pointer is valid input

void **lwmem_free_s_ex**(*lwmem_t* ***const** *lw*, void ****const** *ptr*)
Safe version of free function.

After memory is freed, input pointer is safely set to `NULL` to prevent use of dangling pointers.

It is advised to use this function when freeing memory.

Note This function is thread safe when *LWMEM_CFG_OS* is enabled

Parameters

- `[in] lw`: LwMEM instance. Set to `NULL` to use default instance. Instance must be the same as used during allocation procedure
- `[in] ptr`: Pointer to pointer to allocated memory. When set to non `NULL`, pointer is freed and set to `NULL`

struct lwmem_block_t
#include <*lwmem.h*> Memory block structure.

Public Members

struct lwmem_block ***next**
Next free memory block on linked list. Set to LWMEM_BLOCK_ALLOC_MARK when block is allocated and in use

size_t **size**
Size of block, including metadata part. MSB bit is set to `1` when block is allocated and in use, or `0` when block is considered free

struct lwmem_t
#include <*lwmem.h*> LwMEM main structure.

Public Members

lwmem_block_t **start_block**
Holds beginning of memory allocation regions

lwmem_block_t ***end_block**
Pointer to the last memory location in regions linked list

size_t **mem_available_bytes**
Memory size available for allocation

size_t **mem_regions_count**
Number of regions used for allocation

LWMEM_CFG_OS_MUTEX_HANDLE **mutex**
System mutex for OS

struct lwmem_region_t
#include <*lwmem.h*> Memory region descriptor.

Public Members

void ***start_addr**
Region start address

size_t **size**
Size of region in units of bytes

5.3.2 LwMEM Configuration

This is the default configuration of the middleware. When any of the settings shall be modified, it shall be done in dedicated application config `lwmem_config.h` file.

Note: Check *Getting started* to create configuration file.

group **LWMEM_CONFIG**
Configuration for LwMEM library.

Defines

LWMEM_CFG_OS
Enables `1` or disables `0` operating system support in the library.

Note When `LWMEM_CFG_OS` is enabled, user must implement functions in *System functions* group.

LWMEM_CFG_OS_MUTEX_HANDLE
Mutex handle type.

Note This value must be set in case *LWMEM_CFG_OS* is set to `1`. If data type is not known to compiler, include header file with definition before you define handle type

LWMEM_CFG_ALIGN_NUM
Number of bits to align memory address and memory size.

Some CPUs do not offer unaligned memory access (Cortex-M0 as an example) therefore it is important to have alignment of data addresses and potentialy length of data

Note This value must be a power of `2` for number of bytes. Usually alignment of `4` bytes fits to all processors.

5.3.3 System functions

System function are used in conjunction with thread safety. Please check *Thread safety* section for more information

group **LWMEM_SYS**
System functions when used with operating system.

Functions

uint8_t **lwmem_sys_mutex_create**(LWMEM_CFG_OS_MUTEX_HANDLE *m)
Create a new mutex and assign value to handle.

Return `1` on success, `0` otherwise

Parameters

- `[out] m`: Output variable to save mutex handle

uint8_t **lwmem_sys_mutex_isvalid**(LWMEM_CFG_OS_MUTEX_HANDLE *m)
Check if mutex handle is valid.

Return `1` on success, `0` otherwise

Parameters

- `[in] m`: Mutex handle to check if valid

uint8_t **lwmem_sys_mutex_wait**(LWMEM_CFG_OS_MUTEX_HANDLE *m)
Wait for a mutex until ready (unlimited time)

Return `1` on success, `0` otherwise

Parameters

- `[in] m`: Mutex handle to wait for

uint8_t **lwmem_sys_mutex_release**(LWMEM_CFG_OS_MUTEX_HANDLE *m)
Release already locked mutex.

Return `1` on success, `0` otherwise

Parameters

- `[in] m`: Mutex handle to release

5.4 Examples and demos

Various examples are provided for fast library evaluation on embedded systems. These are optimized prepared and maintained for `2` platforms, but could be easily extended to more platforms:

- WIN32 examples, prepared as Visual Studio Community projects
- ARM Cortex-M examples for STM32, prepared as STM32CubeIDE GCC projects

Warning: Library is platform independent and can be used on any platform.

5.4.1 Example architectures

There are many platforms available today on a market, however supporting them all would be tough task for single person. Therefore it has been decided to support (for purpose of examples) `2` platforms only, *WIN32* and *STM32*.

WIN32

Examples for *WIN32* are prepared as Visual Studio Community projects. You can directly open project in the IDE, compile & debug.

STM32

Embedded market is supported by many vendors and STMicroelectronics is, with their STM32 series of microcontrollers, one of the most important players. There are numerous amount of examples and topics related to this architecture.

Examples for *STM32* are natively supported with STM32CubeIDE, an official development IDE from STMicroelectronics.

You can run examples on one of official development boards, available in repository examples.

5.4.2 Examples list

Here is a list of all examples coming with this library.

Tip: Examples are located in `/examples/` folder in downloaded package. Check *Download library* section to get your package.

LwMEM bare-metal

Simple example, not using operating system, showing basic configuration of the library. It can be also called *bare-metal* implementation for simple applications

LwMEM OS

LwMEM library integrated as application memory manager with operating system. It configurex mutual exclusion object `mutex` to allow multiple application threads accessing to LwMEM core functions

LwMEM multi regions

Multi regions example shows how to configure multiple linear regions to be applied to single LwMEM instance. It uses simple varible array to demonstrate memory sections in embedded systems.

LwMEM multi instances & regions

This example shows how can application add custom (or more of them) instances for LwMEM memory management. Each LwMEM instance has its own set of regions to work with.

LwMEM instances are between each-other completely isolated.

INDEX