

LameOS

1.0

Generated by Doxygen 1.9.1



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 heap Struct Reference	5
3.1.1 Member Data Documentation	5
3.1.1.1 saddr	5
3.1.1.2 table	6
3.2 heap_table Struct Reference	6
3.2.1 Member Data Documentation	6
3.2.1.1 entries	6
3.2.1.2 total	6
3.3 idt_desc Struct Reference	6
3.3.1 Detailed Description	7
3.3.2 Member Data Documentation	7
3.3.2.1 offset_1	7
3.3.2.2 offset_2	7
3.3.2.3 selector	7
3.3.2.4 type_attr	7
3.3.2.5 zero	8
3.4 idtr_desc Struct Reference	8
3.4.1 Detailed Description	8
3.4.2 Member Data Documentation	8
3.4.2.1 base	8
3.4.2.2 limit	8
<b>4 File Documentation</b>	<b>9</b>
4.1 /home/fransys/prog/C/bootable_code/src/config.h File Reference	9
4.1.1 Macro Definition Documentation	10
4.1.1.1 KERNEL_CODE_SELECTOR	10
4.1.1.2 KERNEL_DATA_SELECTOR	10
4.1.1.3 LAMEOS_HEAP_ADDRESS	10
4.1.1.4 LAMEOS_HEAP_BLOCK_SIZE	10
4.1.1.5 LAMEOS_HEAP_SIZE_BYTES	10
4.1.1.6 LAMEOS_HEAP_TABLE_ADDRESS	10
4.1.1.7 LAMEOS_TOTAL_INTERRUPTS	11
4.2 /home/fransys/prog/C/bootable_code/src/idt/idt.c File Reference	11
4.2.1 Function Documentation	12
4.2.1.1 idt_init()	12
4.2.1.2 idt_load()	12

4.2.1.3	idt_set()	13
4.2.1.4	idt_zero()	13
4.2.1.5	int21h()	13
4.2.1.6	int21h_handler()	14
4.2.1.7	no_interrupt()	14
4.2.1.8	no_interrupt_handler()	14
4.2.2	Variable Documentation	14
4.2.2.1	idt_descriptors	14
4.2.2.2	idtr_descriptor	14
4.3	/home/fransys/prog/C/bootable_code/src/idt/idt.h File Reference	15
4.3.1	Function Documentation	16
4.3.1.1	__attribute__()	16
4.3.1.2	idt_init()	16
4.3.2	Variable Documentation	17
4.3.2.1	base	17
4.3.2.2	limit	17
4.3.2.3	offset_1	17
4.3.2.4	offset_2	17
4.3.2.5	selector	17
4.3.2.6	type_attr	17
4.3.2.7	zero	17
4.4	/home/fransys/prog/C/bootable_code/src/io/io.h File Reference	18
4.4.1	Function Documentation	18
4.4.1.1	insb()	18
4.4.1.2	insw()	19
4.4.1.3	outb()	19
4.4.1.4	outw()	20
4.5	/home/fransys/prog/C/bootable_code/src/kernel.c File Reference	20
4.5.1	Function Documentation	21
4.5.1.1	kernel_main()	21
4.5.1.2	lame_color_show()	21
4.5.1.3	print()	21
4.5.1.4	strlen()	22
4.5.1.5	term_initialize()	22
4.5.1.6	term_make_char()	22
4.5.1.7	term_putchar()	23
4.5.1.8	term_writechar()	23
4.5.2	Variable Documentation	24
4.5.2.1	term_col	24
4.5.2.2	term_row	24
4.5.2.3	video_mem	24
4.6	/home/fransys/prog/C/bootable_code/src/kernel.h File Reference	24

4.6.1 Macro Definition Documentation	25
4.6.1.1 VGA_HEIGHT	25
4.6.1.2 VGA_WIDTH	26
4.6.2 Function Documentation	26
4.6.2.1 kernel_main()	26
4.6.2.2 lame_color_show()	26
4.6.2.3 print()	26
4.6.2.4 strlen()	26
4.6.2.5 term_initialize()	27
4.6.2.6 term_make_char()	27
4.6.2.7 term_putchar()	28
4.6.2.8 term_writechar()	28
4.7 /home/fransys/prog/C/bootable_code/src/memory/heap/heap.c File Reference	28
4.7.1 Detailed Description	29
4.7.2 Function Documentation	30
4.7.2.1 heap_address_to_block()	30
4.7.2.2 heap_align_value_to_upper()	30
4.7.2.3 heap_block_to_address()	31
4.7.2.4 heap_check_alignment()	32
4.7.2.5 heap_check_table()	32
4.7.2.6 heap_create()	33
4.7.2.7 heap_free()	34
4.7.2.8 heap_get_entry_type()	34
4.7.2.9 heap_get_start_block()	36
4.7.2.10 heap_malloc()	37
4.7.2.11 heap_malloc_blocks()	37
4.7.2.12 heap_mark_blocks_free()	39
4.7.2.13 heap_mark_blocks_taken()	39
4.8 /home/fransys/prog/C/bootable_code/src/memory/heap/heap.h File Reference	40
4.8.1 Detailed Description	41
4.8.2 Macro Definition Documentation	41
4.8.2.1 HEAP_BLOCK_HAS_NEXT	41
4.8.2.2 HEAP_BLOCK_IS_FIRST	42
4.8.2.3 HEAP_BLOCK_TABLE_ENTRY_FREE	42
4.8.2.4 HEAP_BLOCK_TABLE_ENTRY_TAKEN	42
4.8.3 Typedef Documentation	42
4.8.3.1 HEAP_BLOCK_TABLE_ENTRY	42
4.8.4 Function Documentation	42
4.8.4.1 heap_create()	42
4.8.4.2 heap_free()	43
4.8.4.3 heap_malloc()	44
4.9 /home/fransys/prog/C/bootable_code/src/memory/heap/kheap.c File Reference	44

4.9.1 Detailed Description . . . . .	45
4.9.2 Function Documentation . . . . .	45
4.9.2.1 kfree() . . . . .	46
4.9.2.2 kheap_init() . . . . .	46
4.9.2.3 kmalloc() . . . . .	46
4.9.3 Variable Documentation . . . . .	47
4.9.3.1 kernel_heap . . . . .	47
4.9.3.2 kernel_heap_table . . . . .	47
4.10 /home/fransys/prog/C/bootable_code/src/memory/heap/kheap.h File Reference . . . . .	47
4.10.1 Detailed Description . . . . .	48
4.10.2 Function Documentation . . . . .	49
4.10.2.1 kfree() . . . . .	49
4.10.2.2 kheap_init() . . . . .	49
4.10.2.3 kmalloc() . . . . .	49
4.11 /home/fransys/prog/C/bootable_code/src/memory/memory.c File Reference . . . . .	50
4.11.1 Function Documentation . . . . .	50
4.11.1.1 memset() . . . . .	51
4.12 /home/fransys/prog/C/bootable_code/src/memory/memory.h File Reference . . . . .	51
4.12.1 Function Documentation . . . . .	52
4.12.1.1 memset() . . . . .	52
4.13 /home/fransys/prog/C/bootable_code/src/status.h File Reference . . . . .	52
4.13.1 Detailed Description . . . . .	53
4.13.2 Macro Definition Documentation . . . . .	53
4.13.2.1 EINVAL . . . . .	53
4.13.2.2 EIO . . . . .	54
4.13.2.3 ENOMEM . . . . .	54
4.13.2.4 LAMEOS_OK . . . . .	54
<b>Index</b>	<b>55</b>

# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">heap</a>	.....	5
<a href="#">heap_table</a>	.....	6
<a href="#">idt_desc</a>	.....	
	Interrupt Descriptor Table (IDT) Descriptor .....	6
<a href="#">idtr_desc</a>	.....	
	IDT Register (IDTR) Descriptor .....	8





## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

/home/fransys/prog/C/bootable_code/src/ <a href="#">config.h</a>	9
/home/fransys/prog/C/bootable_code/src/ <a href="#">kernel.c</a>	20
/home/fransys/prog/C/bootable_code/src/ <a href="#">kernel.h</a>	24
/home/fransys/prog/C/bootable_code/src/ <a href="#">status.h</a>	
Defines system-level status codes	52
/home/fransys/prog/C/bootable_code/src/idt/ <a href="#">idt.c</a>	11
/home/fransys/prog/C/bootable_code/src/idt/ <a href="#">idt.h</a>	15
/home/fransys/prog/C/bootable_code/src/io/ <a href="#">io.h</a>	18
/home/fransys/prog/C/bootable_code/src/memory/ <a href="#">memory.c</a>	50
/home/fransys/prog/C/bootable_code/src/memory/ <a href="#">memory.h</a>	51
/home/fransys/prog/C/bootable_code/src/memory/heap/ <a href="#">heap.c</a>	
Heap management implementation	28
/home/fransys/prog/C/bootable_code/src/memory/heap/ <a href="#">heap.h</a>	
Heap management interface	40
/home/fransys/prog/C/bootable_code/src/memory/heap/ <a href="#">kheap.c</a>	
Kernel heap management implementation	44
/home/fransys/prog/C/bootable_code/src/memory/heap/ <a href="#">kheap.h</a>	
Kernel heap management interfaces	47



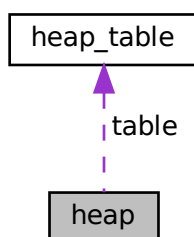
## Chapter 3

# Class Documentation

### 3.1 heap Struct Reference

```
#include <heap.h>
```

Collaboration diagram for heap:



#### Public Attributes

- struct `heap_table` \* `table`
- void \* `saddr`

#### 3.1.1 Member Data Documentation

##### 3.1.1.1 saddr

```
void* heap::saddr
```

### 3.1.1.2 table

```
struct heap_table* heap::table
```

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

- [/home/fransys/prog/C/bootable\\_code/src/memory/heap/heap.h](#)

## 3.2 heap\_table Struct Reference

```
#include <heap.h>
```

### Public Attributes

- [HEAP\\_BLOCK\\_TABLE\\_ENTRY](#) \* [entries](#)
- [size\\_t](#) [total](#)

### 3.2.1 Member Data Documentation

#### 3.2.1.1 entries

```
HEAP\_BLOCK\_TABLE\_ENTRY* heap_table::entries
```

#### 3.2.1.2 total

```
size_t heap_table::total
```

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

- [/home/fransys/prog/C/bootable\\_code/src/memory/heap/heap.h](#)

## 3.3 idt\_desc Struct Reference

Interrupt Descriptor Table (IDT) Descriptor.

```
#include <idt.h>
```

## Public Attributes

- uint16\_t [offset\\_1](#)
- uint16\_t [selector](#)
- uint8\_t [zero](#)
- uint8\_t [type\\_attr](#)
- uint16\_t [offset\\_2](#)

### 3.3.1 Detailed Description

Interrupt Descriptor Table (IDT) Descriptor.

This structure represents a single entry in the Interrupt Descriptor Table (IDT). The IDT is used by the processor to handle interrupts and exceptions. Each IDT descriptor corresponds to a specific interrupt or exception and provides the necessary information for the processor to handle them correctly.

### 3.3.2 Member Data Documentation

#### 3.3.2.1 offset\_1

```
uint16_t idt_desc::offset_1
```

#### 3.3.2.2 offset\_2

```
uint16_t idt_desc::offset_2
```

#### 3.3.2.3 selector

```
uint16_t idt_desc::selector
```

#### 3.3.2.4 type\_attr

```
uint8_t idt_desc::type_attr
```

### 3.3.2.5 zero

```
uint8_t idt_desc::zero
```

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

- [/home/fransys/prog/C/bootable\\_code/src/idt/idt.h](#)

## 3.4 idtr\_desc Struct Reference

IDT Register (IDTR) Descriptor.

```
#include <idt.h>
```

### Public Attributes

- [uint16\\_t limit](#)
- [uint32\\_t base](#)

### 3.4.1 Detailed Description

IDT Register (IDTR) Descriptor.

This structure represents the IDT Register (IDTR) descriptor, which provides the base address and limit of the Interrupt Descriptor Table (IDT). The IDTR is a control register used by the processor to locate and access the IDT.

### 3.4.2 Member Data Documentation

#### 3.4.2.1 base

```
uint32_t idtr_desc::base
```

#### 3.4.2.2 limit

```
uint16_t idtr_desc::limit
```

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

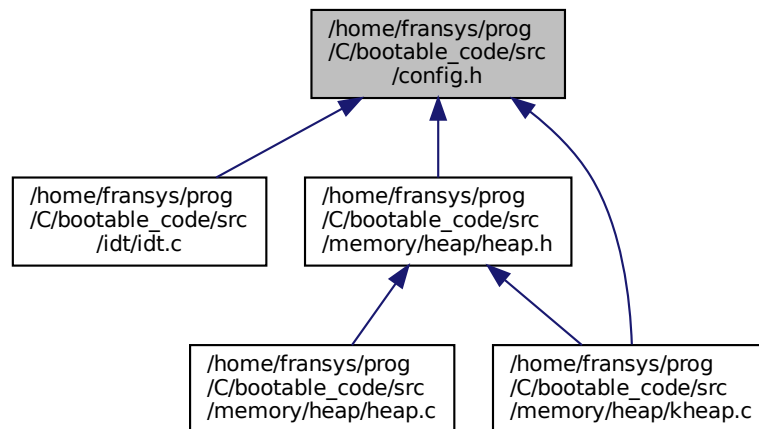
- [/home/fransys/prog/C/bootable\\_code/src/idt/idt.h](#)

## Chapter 4

# File Documentation

### 4.1 /home/fransys/prog/C/bootable\_code/src/config.h File Reference

This graph shows which files directly or indirectly include this file:



### Macros

- `#define KERNEL_CODE_SELECTOR 0X08`  
*Code Segment Selector.*
- `#define KERNEL_DATA_SELECTOR 0X10`  
*Data Segment Selector.*
- `#define LAMEOS_TOTAL_INTERRUPTS 512`  
*Macro Constant Defining Total Interrupts.*
- `#define LAMEOS_HEAP_SIZE_BYTES 104857600`
- `#define LAMEOS_HEAP_BLOCK_SIZE 4096`
- `#define LAMEOS_HEAP_ADDRESS 0x01000000`
- `#define LAMEOS_HEAP_TABLE_ADDRESS 0x00007E00`

## 4.1.1 Macro Definition Documentation

### 4.1.1.1 KERNEL\_CODE\_SELECTOR

```
#define KERNEL_CODE_SELECTOR 0x08
```

Code Segment Selector.

The offset of the code\_seg entry in the GDT is 0x08.

### 4.1.1.2 KERNEL\_DATA\_SELECTOR

```
#define KERNEL_DATA_SELECTOR 0x10
```

Data Segment Selector.

The offset of the data\_seg entry in the GDT is 0x10.

### 4.1.1.3 LAMEOS\_HEAP\_ADDRESS

```
#define LAMEOS_HEAP_ADDRESS 0x01000000
```

### 4.1.1.4 LAMEOS\_HEAP\_BLOCK\_SIZE

```
#define LAMEOS_HEAP_BLOCK_SIZE 4096
```

### 4.1.1.5 LAMEOS\_HEAP\_SIZE\_BYTES

```
#define LAMEOS_HEAP_SIZE_BYTES 104857600
```

### 4.1.1.6 LAMEOS\_HEAP\_TABLE\_ADDRESS

```
#define LAMEOS_HEAP_TABLE_ADDRESS 0x00007E00
```



#### 4.1.1.7 LAMEOS\_TOTAL\_INTERRUPTS

```
#define LAMEOS_TOTAL_INTERRUPTS 512
```

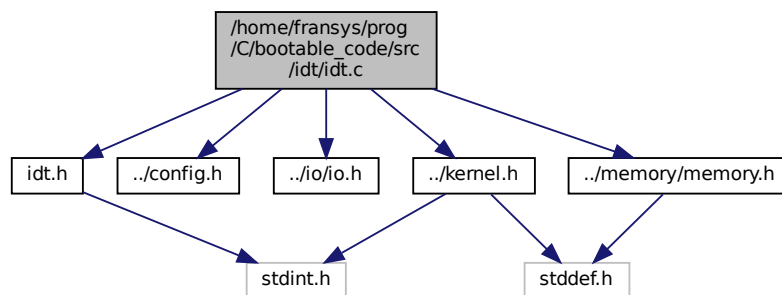
Macro Constant Defining Total Interrupts.

The IDT is an array of 512 descriptors, each 8 bytes long. Although in reality only 256 are actually available for use by programmers. The rest are reserved by the CPU for one reason or another.

## 4.2 /home/fransys/prog/C/bootable\_code/src/idt/idt.c File Reference

```
#include "idt.h"
#include "../config.h"
#include "../io/io.h"
#include "../kernel.h"
#include "../memory/memory.h"
```

Include dependency graph for idt.c:



## Functions

- void `idt_load` (struct `idtr_desc` \*ptr)  
*Wrapper function for assembly routine `idt_load`.*
- void `int21h` ()
- void `no_interrupt` ()
- void `int21h_handler` ()
- void `no_interrupt_handler` ()
- void `idt_zero` ()  
*Interrupt Zero Definition.*
- void `idt_set` (int interrupt\_no, void \*address)  
*Defines an IDT descriptor.*
- void `idt_init` ()  
*Initialize Kernel Interrupt Descriptor Table (IDT).*

## Variables

- struct [idt\\_desc](#) [idt\\_descriptors](#) [[LAMEOS\\_TOTAL\\_INTERRUPTS](#)]  
*Array of 512 IDT Descriptors.*
- struct [idtr\\_desc](#) [idtr\\_descriptor](#)  
*A struct representing the IDT register (IDTR).*

## 4.2.1 Function Documentation

### 4.2.1.1 [idt\\_init\(\)](#)

```
void idt_init ( )
```

Initialize Kernel Interrupt Descriptor Table (IDT).

Initializes kernel IDT array by zeroing every descriptor in the array, Sets the IDTR descriptor limit and base, Intended to set each IDT descriptor, but currently only sets the interrupt descriptor 0, Concludes Loads the IDTR by calling wrapper function [idt\\_load](#), for the asm function of the same name. The asm routine [idt\\_load](#) loads the IDTR with the kernel IDTR struct.

#### Note

There is a 1:1 mapping between the IDT and the CPU's interrupt numbers.

#### See also

[memset](#) in [src/memory/memory.c](#)

[idt\\_set](#) in [src/idt/idt.c](#)

[idt\\_load](#) in [src/idt/idt.asm](#)

### 4.2.1.2 [idt\\_load\(\)](#)

```
void idt_load (
    struct idtr\_desc * ptr )
```

Wrapper function for assembly routine [idt\\_load](#).

The wrapper fct is called from within [idt\\_init](#). It loads the IDTR by calling the assembly function [idt\\_load](#). By loading the kernel IDTR struct, the processor knows where the kernel IDT struct-array is located in memory.

#### Note

The assembly routine is exposed to the linker by `global idt_load` in the [idt.asm](#) file.

#### See also

[idt\\_init](#) in [src/idt/idt.c](#)

[idt\\_load](#) in [src/idt/idt.asm](#)

## Parameters

<i>ptr</i>	a void pointer to the IDTR descriptor
------------	---------------------------------------

#### 4.2.1.3 idt\_set()

```
void idt_set (
    int interrupt_no,
    void * address )
```

Defines an IDT descriptor.

Defines a descriptor by setting the offset, selector, zero, type\_attr, and offset\_2 fields of the descriptor. The offset is the address of the programmable interrupt routine. The selector is the kernel code selector. The zero field is unused and set to zero. The type\_attr field is set to 0xEE, which is the type and attributes for a 32-bit interrupt gate. The offset\_2 field is the upper 16 bits of the offset.

## See also

[idt\\_init](#) in [src/idt/idt.c](#)

## Parameters

<i>interrupt_no</i>	The CPU interrupt number to map fct address to.
<i>address</i>	The address of the programmable interrupt routine.

#### 4.2.1.4 idt\_zero()

```
void idt_zero ( )
```

Interrupt Zero Definition.

This interrupt routine is called by the CPU when a divide by zero exception occurs. It is mapped to interrupt 0 in the CPU's IDT when `idt_init` is called. The routine clears the screen and prints an error message.

## See also

[idt\\_load](#) in [src/idt/idt.asm](#)

#### 4.2.1.5 int21h()

```
void int21h ( )
```

#### 4.2.1.6 int21h\_handler()

```
void int21h_handler ( )
```

#### 4.2.1.7 no\_interrupt()

```
void no_interrupt ( )
```

#### 4.2.1.8 no\_interrupt\_handler()

```
void no_interrupt_handler ( )
```

### 4.2.2 Variable Documentation

#### 4.2.2.1 idt\_descriptors

```
struct idt_desc idt_descriptors[LAMEOS_TOTAL_INTERRUPTS]
```

Array of 512 IDT Descriptors.

The kernel maintains an array of 512 IDT descriptors. Each descriptor corresponds to a specific interrupt or exception. The array is initialized by `idt_init`.

#### 4.2.2.2 idtr\_descriptor

```
struct idtr_desc idtr_descriptor
```

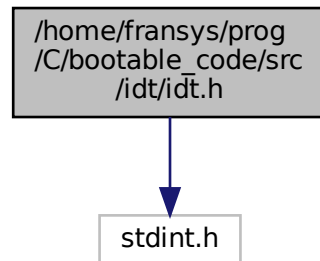
A struct representing the IDT register (IDTR).

The IDTR is a control register used by the processor to locate and access the IDT. The IDTR is initialized by `idt_init`.

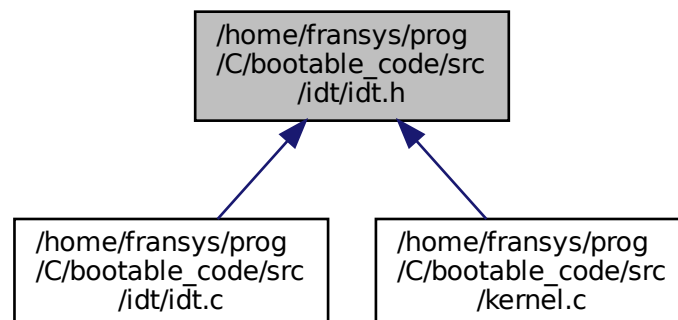
## 4.3 /home/fransys/prog/C/bootable\_code/src/idt/idt.h File Reference

```
#include <stdint.h>
```

Include dependency graph for idt.h:



This graph shows which files directly or indirectly include this file:



### Classes

- struct [idt\\_desc](#)  
*Interrupt Descriptor Table (IDT) Descriptor.*
- struct [idtr\\_desc](#)  
*IDT Register (IDTR) Descriptor.*

### Functions

- struct [idt\\_desc](#) [\\_\\_attribute\\_\\_\(\(packed\)\)](#)
- void [idt\\_init](#) ()  
*Initialize Kernel Interrupt Descriptor Table (IDT).*

## Variables

- `uint16_t` [offset\\_1](#)
- `uint16_t` [selector](#)
- `uint8_t` [zero](#)
- `uint8_t` [type\\_attr](#)
- `uint16_t` [offset\\_2](#)
- `uint16_t` [limit](#)
- `uint32_t` [base](#)

## 4.3.1 Function Documentation

### 4.3.1.1 `__attribute__()`

```
struct idtr_desc __attribute__ (  
    (packed) )
```

### 4.3.1.2 `idt_init()`

```
void idt_init ( )
```

Initialize Kernel Interrupt Descriptor Table (IDT).

Initializes the Interrupt Descriptor Table (IDT) by: Zeroing out the user-IDT array, Setting the IDT Register (IDTR) descriptor limit and base, Setting the IDT descriptors for each programmed interrupt, and Loading the IDTR by calling the assembly function `idt_load`.

#### See also

[idt\\_init](#) in [src/idt/idt.c](#)

Initializes kernel IDT array by zeroing every descriptor in the array, Sets the IDTR descriptor limit and base, Intended to set each IDT descriptor, but currently only sets the interrupt descriptor 0, Concludes Loads the IDTR by calling wrapper function `idt_load`, for the asm function of the same name. The asm routine `idt_load` loads the IDTR with the kernel IDTR struct.

#### Note

There is a 1:1 mapping between the IDT and the CPU's interrupt numbers.

#### See also

[memset](#) in [src/memory/memory.c](#)

[idt\\_set](#) in [src/idt/idt.c](#)

[idt\\_load](#) in [src/idt/idt.asm](#)

## 4.3.2 Variable Documentation

### 4.3.2.1 base

`uint32_t base`

### 4.3.2.2 limit

`uint16_t limit`

### 4.3.2.3 offset\_1

`uint16_t offset_1`

### 4.3.2.4 offset\_2

`uint16_t offset_2`

### 4.3.2.5 selector

`uint16_t selector`

### 4.3.2.6 type\_attr

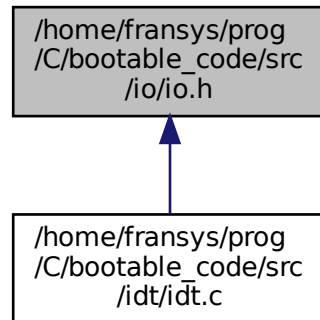
`uint8_t type_attr`

### 4.3.2.7 zero

`uint8_t zero`

## 4.4 /home/fransys/prog/C/bootable\_code/src/io/io.h File Reference

This graph shows which files directly or indirectly include this file:



### Functions

- unsigned char `insb` (unsigned short port)  
*C wrapper of BIOS `insb` instruction Reads a byte in from a PIO port.*
- unsigned short `insw` (unsigned short port)  
*C wrapper of BIOS `insw` instruction Reads a word in from a PIO port.*
- void `outb` (unsigned short port, unsigned char val)  
*C wrapper of BIOS `outb` instruction Writes a byte out to a PIO port.*
- void `outw` (unsigned short port, unsigned short val)  
*C wrapper of BIOS `outw` instruction Writes a word out to a PIO port.*

### 4.4.1 Function Documentation

#### 4.4.1.1 `insb()`

```
unsigned char insb (
    unsigned short port )
```

C wrapper of BIOS `insb` instruction Reads a byte in from a PIO port.

See also

`/src/io/io.asm`



#### Parameters

<i>port</i>	The PIO port to read from, range 0x0000 - 0xFFFF (0-65535).
-------------	---

#### Returns

unsigned char, the byte read in from the port.

#### Note

This function is implemented in assembly. A char is 1 byte.

#### 4.4.1.2 insw()

```
unsigned short insw (  
    unsigned short port )
```

C wrapper of BIOS `insw` instruction Reads a word in from a PIO port.

#### Parameters

<i>port</i>	The PIO port to read from, range 0x0000 - 0xFFFF (0-65535).
-------------	---

#### Returns

unsigned short, the word read in from the port.

#### Note

This function is implemented in assembly. A short is 2 bytes.

#### 4.4.1.3 outb()

```
void outb (  
    unsigned short port,  
    unsigned char val )
```

C wrapper of BIOS `outb` instruction Writes a byte out to a PIO port.

#### Parameters

<i>port</i>	The PIO port to write to, range 0x0000 - 0xFFFF (0-65535).
<i>val</i>	The byte to write out to the port.

**Note**

This function is implemented in assembly. A char is 1 byte.

**4.4.1.4 outw()**

```
void outw (
    unsigned short port,
    unsigned short val )
```

C wrapper of BIOS `outw` instruction Writes a word out to a PIO port.

**Parameters**

<i>port</i>	The PIO port to write to, range 0x0000 - 0xFFFF (0-65535).
<i>val</i>	The word to write out to the port.

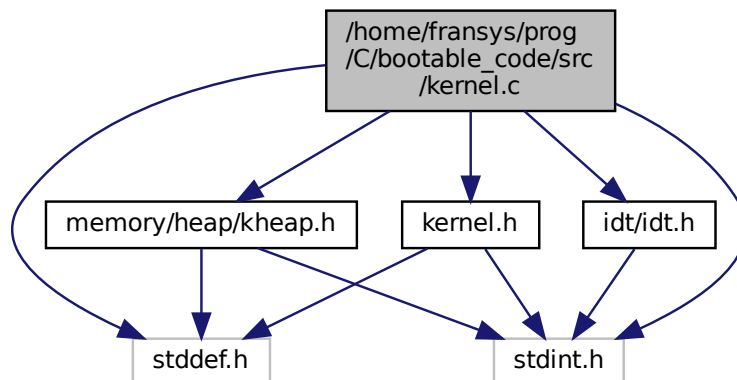
**Note**

This function is implemented in assembly. A short is 2 bytes.

**4.5 /home/fransys/prog/C/bootable\_code/src/kernel.c File Reference**

```
#include "kernel.h"
#include <stddef.h>
#include <stdint.h>
#include "idt/idt.h"
#include "memory/heap/kheap.h"
```

Include dependency graph for kernel.c:



## Functions

- uint16\_t `term_make_char` (char c, char color)  
*Decodes a character and color into a uint16\_t.*
- void `term_putchar` (int x, int y, char c, char color)  
*Writes a character to the VGA framebuffer.*
- void `term_initialize` ()  
*Initializes the VGA framebuffer.*
- size\_t `strlen` (const char \*str)  
*Returns the length of a string.*
- void `term_writechar` (char c, char color)  
*Writes a character, advancing cursor, newline if necessary.*
- void `print` (const char \*str)  
*Writes a string using term\_writechar.*
- void `lame_color_show` ()  
*This is what LameOS is all about.*
- void `kernel_main` ()

## Variables

- uint16\_t \* `video_mem` = 0  
*Pointer to VGA Framebuffer.*
- uint16\_t `term_row` = 0  
*VGA Framebuffer Width.*
- uint16\_t `term_col` = 0  
*VGA Framebuffer Height.*

### 4.5.1 Function Documentation

#### 4.5.1.1 kernel\_main()

```
void kernel_main ( )
```

#### 4.5.1.2 lame\_color\_show()

```
void lame_color_show ( )
```

This is what LameOS is all about.

This function iterates kaleidoscopically through all characters and colors in the VGA framebuffer. It does this forever.  
EPILEPSY WARNING!

#### 4.5.1.3 print()

```
void print (
    const char * str )
```

Writes a string using term\_writechar.

This function writes a string by iterating through the string and writing each character using term\_writechar to the VGA framebuffer.

**Parameters**

<i>str</i>	The string to write.
------------	----------------------

**4.5.1.4 strlen()**

```
size_t strlen (
    const char * str )
```

Returns the length of a string.

This function returns the length of a string by iterating through the string until it reaches a null terminator, maintaining a count as it goes.

**Parameters**

<i>str</i>	The string to get the length of.
------------	----------------------------------

**Returns**

size\_t The length of the string.

**4.5.1.5 term\_initialize()**

```
void term_initialize ( )
```

Initializes the VGA framebuffer.

This function initializes the VGA framebuffer by clearing the screen and setting the video\_mem pointer to 0xB8000. The screen is cleared by calling term\_putchar with space characters and a black background on position in the framebuffer.

**Note**

sets term\_row and term\_col to 0. Useful for related functions.

**4.5.1.6 term\_make\_char()**

```
uint16_t term_make_char (
    char c,
    char color )
```

Decodes a character and color into a uint16\_t.

The VGA framebuffer is a 2D array of uint16\_t. Each uint16\_t represents a character and its color. The first 8 bits of the uint16\_t are the character and the last 8 bits are the color.

**Parameters**

<i>c</i>	The character to display.
<i>color</i>	The color of the character.

**Returns**

uint16\_t The character and color encoded into a uint16\_t.

**4.5.1.7 term\_putchar()**

```
void term_putchar (
    int x,
    int y,
    char c,
    char color )
```

Writes a character to the VGA framebuffer.

This function writes a character and color, given by *c* and *color*, to the VGA framebuffer at the specified location, given by *x* and *y*. The function first converts the *x* and *y* to a 1D index, then writes the character and color to the framebuffer at that index.

**Parameters**

<i>x</i>	The x coordinate, column, range 0-79.
<i>y</i>	The y coordinate, row, range 0-24.
<i>c</i>	The character to display, range 0-255.
<i>color</i>	The color of the character, range 0-15.

**4.5.1.8 term\_writechar()**

```
void term_writechar (
    char c,
    char color )
```

Writes a character, advancing cursor, newline if necessary.

Writes a character, advancing the cursor. If the cursor is at the end of the line, the cursor is moved to the next line.

**Parameters**

<i>c</i>	The character to write.
<i>color</i>	The color of the character.

## 4.5.2 Variable Documentation

### 4.5.2.1 term\_col

```
uint16_t term_col = 0
```

VGA Framebuffer Height.

The VGA framebuffer is 25 characters high.

### 4.5.2.2 term\_row

```
uint16_t term_row = 0
```

VGA Framebuffer Width.

The VGA framebuffer is 80 characters wide.

### 4.5.2.3 video\_mem

```
uint16_t* video_mem = 0
```

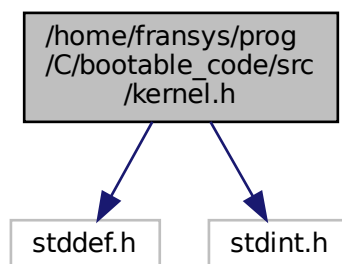
Pointer to VGA Framebuffer.

The kernel uses the VGA framebuffer to display text on the screen. The framebuffer is located at 0xB8000. The kernel writes to the framebuffer using the `term_putchar` function.

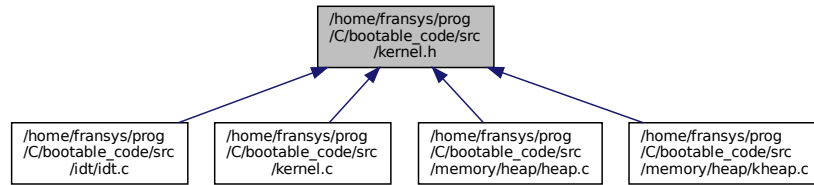
## 4.6 /home/fransys/prog/C/bootable\_code/src/kernel.h File Reference

```
#include <stddef.h>
#include <stdint.h>
```

Include dependency graph for kernel.h:



This graph shows which files directly or indirectly include this file:



## Macros

- `#define VGA_WIDTH 80`  
*Macro Constant for VGA framebuffer width.*
- `#define VGA_HEIGHT 25`  
*Macro Constant for VGA framebuffer height.*

## Functions

- `void kernel_main ()`
- `void term_initialize ()`  
*Initializes the VGA framebuffer.*
- `size_t strlen (const char *str)`  
*Returns the length of a string.*
- `uint16_t term_make_char (char c, char color)`  
*Decodes a character and color into a uint16\_t.*
- `void term_putchar (int x, int y, char c, char color)`  
*Writes a character to the VGA framebuffer.*
- `void term_writechar (char c, char color)`  
*Writes a character, advancing cursor, newline if necessary.*
- `void print (const char *str)`  
*Writes a string using term\_writechar.*
- `void lame_color_show ()`  
*This is what LameOS is all about.*

### 4.6.1 Macro Definition Documentation

#### 4.6.1.1 VGA\_HEIGHT

```
#define VGA_HEIGHT 25
```

Macro Constant for VGA framebuffer height.

#### 4.6.1.2 VGA\_WIDTH

```
#define VGA_WIDTH 80
```

Macro Constant for VGA framebuffer width.

### 4.6.2 Function Documentation

#### 4.6.2.1 kernel\_main()

```
void kernel_main ( )
```

#### 4.6.2.2 lame\_color\_show()

```
void lame_color_show ( )
```

This is what LameOS is all about.

This function iterates kaleidoscopically through all characters and colors in the VGA framebuffer. It does this forever.  
EPILEPSY WARNING!

#### 4.6.2.3 print()

```
void print (
    const char * str )
```

Writes a string using term\_writechar.

This function writes a string by iterating through the string and writing each character using term\_writechar to the VGA framebuffer.

##### Parameters

<i>str</i>	The string to write.
------------	----------------------

#### 4.6.2.4 strlen()

```
size_t strlen (
    const char * str )
```

Returns the length of a string.



This function returns the length of a string by iterating through the string until it reaches a null terminator, maintaining a count as it goes.

#### Parameters

<i>str</i>	The string to get the length of.
------------	----------------------------------

#### Returns

`size_t` The length of the string.

### 4.6.2.5 `term_initialize()`

```
void term_initialize ( )
```

Initializes the VGA framebuffer.

This function initializes the VGA framebuffer by clearing the screen and setting the `video_mem` pointer to 0xB8000. The screen is cleared by calling `term_putchar` with space characters and a black background on position in the framebuffer.

#### Note

sets `term_row` and `term_col` to 0. Useful for related functions.

### 4.6.2.6 `term_make_char()`

```
uint16_t term_make_char (
    char c,
    char color )
```

Decodes a character and color into a `uint16_t`.

The VGA framebuffer is a 2D array of `uint16_t`. Each `uint16_t` represents a character and its color. The first 8 bits of the `uint16_t` are the character and the last 8 bits are the color.

#### Parameters

<i>c</i>	The character to display.
<i>color</i>	The color of the character.

#### Returns

`uint16_t` The character and color encoded into a `uint16_t`.

#### 4.6.2.7 term\_putchar()

```
void term_putchar (
    int x,
    int y,
    char c,
    char color )
```

Writes a character to the VGA framebuffer.

This function writes a character and color, given by *c* and *color*, to the VGA framebuffer at the specified location, given by *x* and *y*. The function first converts the *x* and *y* to a 1D index, then writes the character and color to the framebuffer at that index.

##### Parameters

<i>x</i>	The x coordinate, column, range 0-79.
<i>y</i>	The y coordinate, row, range 0-24.
<i>c</i>	The character to display, range 0-255.
<i>color</i>	The color of the character, range 0-15.

#### 4.6.2.8 term\_writechar()

```
void term_writechar (
    char c,
    char color )
```

Writes a character, advancing cursor, newline if necessary.

Writes a character, advancing the cursor. If the cursor is at the end of the line, the cursor is moved to the next line.

##### Parameters

<i>c</i>	The character to write.
<i>color</i>	The color of the character.

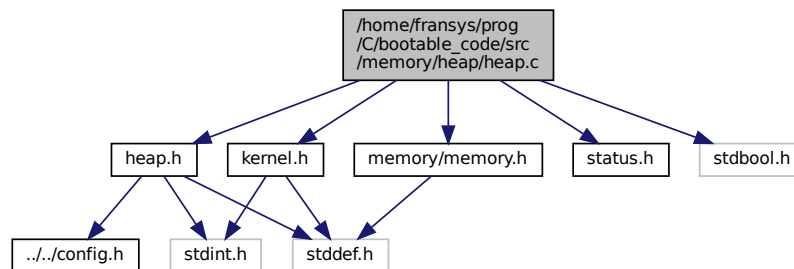
## 4.7 /home/fransys/prog/C/bootable\_code/src/memory/heap/heap.c File Reference

Heap management implementation.

```
#include "heap.h"
#include "kernel.h"
#include "memory/memory.h"
#include "status.h"
```

```
#include <stdbool.h>
```

Include dependency graph for heap.c:



## Functions

- static int [heap\\_check\\_table](#) (void \*ptr, void \*end, struct [heap\\_table](#) \*table)  
*Checks if the heap table has a valid block count.*
- static bool [heap\\_check\\_alignment](#) (void \*ptr)  
*Validates if a given pointer is correctly aligned to the heap block size.*
- int [heap\\_create](#) (struct [heap](#) \*heap, void \*ptr, void \*end, struct [heap\\_table](#) \*table)  
*Initializes a heap object and its corresponding heap table.*
- static uint32\_t [heap\\_align\\_value\\_to\\_upper](#) (uint32\_t val)  
*Adjusts a provided value to align with the next upper heap block boundary.*
- static int [heap\\_get\\_entry\\_type](#) (HEAP\_BLOCK\_TABLE\_ENTRY entry)  
*Retrieves the status of a heap block table entry.*
- int [heap\\_get\\_start\\_block](#) (struct [heap](#) \*heap, uint32\_t total\_blocks)  
*Finds a contiguous sequence of free blocks in the heap.*
- void \* [heap\\_block\\_to\\_address](#) (struct [heap](#) \*heap, uint32\_t block)  
*Converts a heap block index into its corresponding memory address.*
- void [heap\\_mark\\_blocks\\_taken](#) (struct [heap](#) \*heap, int start\_block, int total\_blocks)  
*Marks a range of blocks in the heap as taken.*
- void \* [heap\\_malloc\\_blocks](#) (struct [heap](#) \*heap, uint32\_t total\_blocks)  
*Allocates a specified number of blocks in the heap.*
- void [heap\\_mark\\_blocks\\_free](#) (struct [heap](#) \*heap, int start\_block)
- int [heap\\_address\\_to\\_block](#) (struct [heap](#) \*heap, void \*address)  
*Marks a sequence of blocks in the heap as free.*
- void \* [heap\\_malloc](#) (struct [heap](#) \*heap, size\_t size)  
*Allocates a block of memory from the heap.*
- void [heap\\_free](#) (struct [heap](#) \*heap, void \*ptr)  
*Deallocates a block of memory from the heap.*

### 4.7.1 Detailed Description

Heap management implementation.

This file implements the interface defined in [heap.h](#). It provides functionality for heap management, including memory allocation, memory deallocation, heap initialization, block address calculation, and so on. The heap management strategies are used throughout the operating system's kernel, including memory management and task scheduling.

## 4.7.2 Function Documentation

### 4.7.2.1 heap\_address\_to\_block()

```
int heap_address_to_block (
    struct heap * heap,
    void * address )
```

Marks a sequence of blocks in the heap as free.

This function is responsible for marking a sequence of blocks in the heap as free. It works with a start block index and continues marking subsequent blocks as free until it hits a block that does not have the 'HEAP\_BLOCK\_HAS\_NEXT' flag set. This indicates the end of a previously allocated sequence of blocks.

The function iterates over the heap block table entries starting from the given 'start\_block' index. For each block, it retrieves the corresponding table entry and sets it to 'HEAP\_BLOCK\_TABLE\_ENTRY\_FREE' indicating that the block is now free. It also checks if the current block was part of a multi-block allocation by examining the 'HEAP\_BLOCK\_HAS\_NEXT' flag. If this flag is not set, it means the end of the sequence has been reached, and the function stops marking blocks as free.

#### Parameters

<i>heap</i>	Pointer to the heap object in which the blocks are to be freed.
<i>start_block</i>	The index of the first block in the sequence to be freed.

### 4.7.2.2 heap\_align\_value\_to\_upper()

```
static uint32_t heap_align_value_to_upper (
    uint32_t val ) [static]
```

Adjusts a provided value to align with the next upper heap block boundary.

Invoked by [heap\\_malloc\(\)](#), this function ensures that the provided value (val), which represents an end pointer within the heap, is aligned with a heap block boundary. This is critical for maintaining consistency within the heap structure and enabling efficient memory allocation.

The alignment process involves checking if the provided value is already a multiple of the heap block size (LAMEOS\_HEAP\_BLOCK\_SIZE). If it is, no adjustment is necessary, and the function simply returns the original value.

However, if the value is not a multiple of the block size (i.e., it falls within a block), the function needs to adjust it. It first subtracts the remainder of the value divided by the block size from the value itself. This effectively 'rounds down' the value to the start of the current heap block. Then, it adds the size of a full heap block to this result. The final value is thus rounded up to the start of the next heap block, ensuring alignment with the block boundary.

#### Parameters

<i>val</i>	The end pointer value to be rounded up to the next heap block boundary. This value should be within the heap memory range.
------------	--

#### Returns

`uint32_t` Returns the end pointer value adjusted to align with the heap block size. This will be the original value if it's already a multiple of the block size, or the next upper block boundary otherwise.

#### See also

[heap\\_malloc\(\)](#)

#### 4.7.2.3 heap\_block\_to\_address()

```
void* heap_block_to_address (
    struct heap * heap,
    uint32_t block )
```

Converts a heap block index into its corresponding memory address.

This function assists in the conversion of a relative block index within the heap into an absolute memory address. The function achieves this by taking the start address of the heap and adding the product of the block index and the predefined size of each heap block (`LAMEOS_HEAP_BLOCK_SIZE`).

This form of address calculation is central to the functioning of a heap memory manager, enabling the translation from an abstract block index to a physical memory address that can be utilized for storing and retrieving data. This function is typically invoked during the memory allocation process, where specific blocks within the heap are allocated to meet a requested memory size.

It's important to note that this function doesn't check if the block index is within the valid range of the heap or whether the block at the given index is free or allocated.

#### Parameters

<i>heap</i>	Pointer to the heap object. The base address for the heap memory resides in this structure.
<i>block</i>	The block index within the heap to be translated into a memory address.

#### Returns

`void*` The absolute memory address that corresponds to the given block index within the heap.

#### See also

[heap\\_malloc\\_blocks\(\)](#)

#### 4.7.2.4 heap\_check\_alignment()

```
static bool heap_check_alignment (
    void * ptr ) [static]
```

Validates if a given pointer is correctly aligned to the heap block size.

Invoked by [heap\\_create\(\)](#), this function checks whether the supplied pointer adheres to the alignment requirements of the heap block size. In a correctly functioning heap, each block of memory must start at an address that is a multiple of the heap block size (LAMEOS\_HEAP\_BLOCK\_SIZE). To determine alignment, the function calculates the modulus of the pointer's value and the heap block size. In memory arithmetic, a pointer that is correctly aligned to a particular block size will have a modulus of zero when its value is divided by the block size. Therefore, if the modulus is zero, the function deems the pointer correctly aligned and returns true. Conversely, if the modulus is non-zero, it denotes misalignment and the function returns false.

##### Parameters

<i>ptr</i>	The pointer whose alignment is to be verified. It could point to any arbitrary location within the heap.
------------	--

##### Returns

true if the pointer is aligned correctly with respect to the heap block size, ensuring that it points to the start of a block.

false if the pointer is not aligned, indicating it may point to the middle of a block or some other misaligned location.

##### See also

[heap\\_create\(\)](#)

#### 4.7.2.5 heap\_check\_table()

```
static int heap_check_table (
    void * ptr,
    void * end,
    struct heap_table * table ) [static]
```

Checks if the heap table has a valid block count.

Called by [heap\\_create\(\)](#). It compares the number of blocks physically present in the heap with the block count as indicated by the heap table structure. Is a 'sanity check' to ensure that the heap table is valid at the point of heap creation.

The function works by first calculating the total size of the heap. This is accomplished by performing pointer arithmetic between the end and start pointers of the heap. It then translates this size into the number of blocks by dividing the total heap size by the size of a single heap block (LAMEOS\_HEAP\_BLOCK\_SIZE). The result of this division yields the total number of blocks in the heap.

Next, the function compares the computed total number of blocks with the total number of blocks as stated in the heap table. If the two totals do not match, the function sets the result to -EINVAL, signalling an invalid argument error. Otherwise, the function returns 0, indicating a valid heap table.

## Parameters

<i>ptr</i>	Pointer to the start of the heap. It provides the base address for the heap memory.
<i>end</i>	Pointer to the end of the heap. It serves as the boundary of the heap memory.
<i>table</i>	Pointer to the heap table. The heap table keeps track of the total number of blocks in the heap.

## Returns

int Returns 0 if the heap table is valid, -EINVAL otherwise. -EINVAL is a flag indicating that an invalid argument was encountered.

## See also

[heap\\_create\(\)](#)

## 4.7.2.6 heap\_create()

```
int heap_create (
    struct heap * heap,
    void * ptr,
    void * end,
    struct heap_table * table )
```

Initializes a heap object and its corresponding heap table.

This function is invoked by [kheap\\_init\(\)](#) to setup a heap object. It conducts a series of validation checks and initializations to ensure the heap is ready for use.

The function begins by verifying the alignment of the start (*ptr*) and end pointers of the heap. If either pointer is not correctly aligned to the heap block size, the function aborts the heap creation process and returns -EINVAL to signal the alignment error.

If both pointers are correctly aligned, the function proceeds to initialize the heap object. It first wipes the heap object's memory using `memset`, setting all bytes to zero. This ensures a clean, predictable state for the new heap object. Then, it sets the start address of the heap (*saddr*) and associates the heap object with its heap table.

After initializing the heap object, the function validates the heap table by calling [heap\\_check\\_table\(\)](#). If this function reports an error (by returning a value less than 0), the function halts the creation process and returns the error code.

If the heap table is valid, the function then initializes the heap table. It calculates the size of the table in bytes and sets all entries in the heap table to indicate they're free.

## Parameters

<i>heap</i>	The heap object to initialize. This will house all the essential data about the heap.
<i>ptr</i>	The start address of the heap. It must be aligned to the heap block size.
<i>end</i>	The end address of the heap. It also must be aligned to the heap block size.
<i>table</i>	The heap table associated with the heap. It keeps track of the state of each block in the heap.

**Returns**

int Returns 0 if the heap object and table are successfully initialized. Returns -EINVAL if an alignment or heap table check fails.

**See also**

[kheap\\_init\(\)](#)

**4.7.2.7 heap\_free()**

```
void heap_free (
    struct heap * heap,
    void * ptr )
```

Deallocates a block of memory from the heap.

The function frees up the previously allocated block of memory by marking it as free in the heap's block table. The address of the block to be freed is passed to the function as 'ptr'.

The process begins by converting the memory address 'ptr' to a block index within the heap using the helper function '[heap\\_address\\_to\\_block\(\)](#)'. The resulting block index represents the start of the block(s) that were previously allocated.

After obtaining the start block index, '[heap\\_mark\\_blocks\\_free\(\)](#)' is called to mark the associated block(s) in the heap's block table as free. This effectively deallocates the block of memory and makes it available for future allocation requests.

**Parameters**

<i>heap</i>	Pointer to the heap object from which the memory is to be deallocated.
<i>ptr</i>	Pointer to the start of the block of memory to be deallocated.

**See also**

[heap\\_mark\\_blocks\\_free\(\)](#).

**4.7.2.8 heap\_get\_entry\_type()**

```
static int heap_get_entry_type (
    HEAP_BLOCK_TABLE_ENTRY entry ) [static]
```

Retrieves the status of a heap block table entry.

This utility function extracts the type (status) of a heap block table entry. It performs a bitwise AND operation with the hexadecimal value 0x0F on the provided entry, effectively isolating the lower 4 bits. These bits represent the status of the block.



The usage of the lower 4 bits for block status enables efficient storage and retrieval of this information. Possible status values include 0x00 for a free block and 0x01 for a taken (allocated) block.

It's important to note that this function interprets the status of a single heap block, not an array or sequence of blocks. It aids in the process of finding, allocating, and freeing blocks within the heap.

**Parameters**

<i>entry</i>	The heap block table entry to examine. This value corresponds to a single block within the heap.
--------------	--

**Returns**

int Returns 0 if the heap block is free, or 1 if the block is taken.

**See also**

[heap\\_get\\_start\\_block\(\)](#)

**4.7.2.9 heap\_get\_start\_block()**

```
int heap_get_start_block (
    struct heap * heap,
    uint32_t total_blocks )
```

Finds a contiguous sequence of free blocks in the heap.

This function is invoked when trying to allocate a chunk of memory from the heap. It scans the heap's block table for a contiguous sequence of free blocks that can accommodate the requested memory size. The total size of the memory request is represented in terms of the number of blocks (*total\_blocks*).

The search process starts from the beginning of the heap block table, iterating through each block entry. It maintains a count of consecutive free blocks (*bc*) and the start index of the first block in this free sequence (*bs*).

For each block, it uses the [heap\\_get\\_entry\\_type\(\)](#) function to check if the block is free. If a block is not free, it resets the free block count and start index to start the search anew from the next block. If a block is free and it's the first in a new sequence, the function records its index as the start index.

The function keeps incrementing the free block count until it reaches the total required block count or until it encounters a taken block. If it successfully finds a sufficient sequence of free blocks, it returns the start index of this sequence. Otherwise, it returns an -ENOMEM error to indicate insufficient memory in the heap.

**Parameters**

<i>heap</i>	Pointer to the heap object. This heap contains the block table to search.
<i>total_blocks</i>	The total number of contiguous blocks needed.

**Returns**

int Returns the start index of the free block sequence if successful. If unable to find sufficient contiguous free blocks, it returns -ENOMEM.

#### 4.7.2.10 heap\_malloc()

```
void* heap_malloc (
    struct heap * heap,
    size_t size )
```

Allocates a block of memory from the heap.

The function begins by aligning the requested size to the heap block size. This is done using the helper function ['heap\\_align\\_value\\_to\\_upper\(\)'](#). The alignment ensures that the allocated block of memory will start at an address that is a multiple of 'LAMEOS\_HEAP\_BLOCK\_SIZE', thereby respecting the architecture's memory alignment restrictions. This aligned size is then divided by the block size to determine the total number of blocks needed to satisfy the request.

After the total number of blocks is calculated, ['heap\\_malloc\\_blocks\(\)'](#) is called to allocate these blocks from the heap. If successful, ['heap\\_malloc\\_blocks\(\)'](#) returns a pointer to the start of the allocated memory.

##### Parameters

<i>heap</i>	Pointer to the heap object from which the memory is to be allocated.
<i>size</i>	The number of bytes to allocate.

##### Returns

void\* If the allocation is successful, a pointer to the allocated memory is returned. If the allocation fails, the return value is NULL.

##### See also

[heap\\_malloc\\_blocks\(\)](#).

#### 4.7.2.11 heap\_malloc\_blocks()

```
void* heap_malloc_blocks (
    struct heap * heap,
    uint32_t total_blocks )
```

Allocates a specified number of blocks in the heap.

This function is tasked with finding and allocating a contiguous sequence of free blocks in the heap. The number of blocks required is specified as an input parameter. It follows a three-step process:

1. Locate the start of a sufficient sequence of free blocks: It does so by calling the helper function [heap\\_get\\_start\\_block\(\)](#). This function returns the index of the first block in a sufficient sequence of free blocks. If no such sequence exists, the function returns an error code, and [heap\\_malloc\\_blocks\(\)](#) immediately returns, indicating a failed allocation.
2. Compute the memory address corresponding to the first block: This step involves calling the helper function [heap\\_block\\_to\\_address\(\)](#) with the heap object and the first block index obtained in the previous step. This function calculates the memory address corresponding to a given block index by offsetting the heap's start address with the product of the block index and the block size.

3. Mark the found blocks as taken: In the final step, the function calls [heap\\_mark\\_blocks\\_taken\(\)](#) to mark the blocks as taken in the heap's block table. This function updates the entries in the block table corresponding to the allocated blocks, setting their status as taken and updating the linking flags accordingly.

The function then returns the memory address computed in step 2. This address points to the start of the allocated memory block in the heap.

## Parameters

<i>heap</i>	Pointer to the heap object from which the memory is to be allocated.
<i>total_blocks</i>	The number of contiguous blocks to be allocated.

## Returns

void\* If the allocation was successful, a pointer to the start of the allocated memory. If not, the function returns NULL.

**4.7.2.12 heap\_mark\_blocks\_free()**

```
void heap_mark_blocks_free (
    struct heap * heap,
    int start_block )
```

## Parameters

<i>heap</i>	
<i>start_block</i>	

**4.7.2.13 heap\_mark\_blocks\_taken()**

```
void heap_mark_blocks_taken (
    struct heap * heap,
    int start_block,
    int total_blocks )
```

Marks a range of blocks in the heap as taken.

This function is used when allocating memory from the heap. It receives the index of the first block and the total number of blocks to be marked as taken. It then proceeds to mark these blocks in the heap's block table as taken.

The marking process involves setting the status of each block entry in the heap's block table. The first block is marked with the HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN and HEAP\_BLOCK\_IS\_FIRST flags. If there are multiple blocks, the first block also receives the HEAP\_BLOCK\_HAS\_NEXT flag to indicate that the allocated sequence of blocks continues in the subsequent block.

For the following blocks, the function marks them with the HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN flag. If a block isn't the last in the sequence, it also receives the HEAP\_BLOCK\_HAS\_NEXT flag.

## Parameters

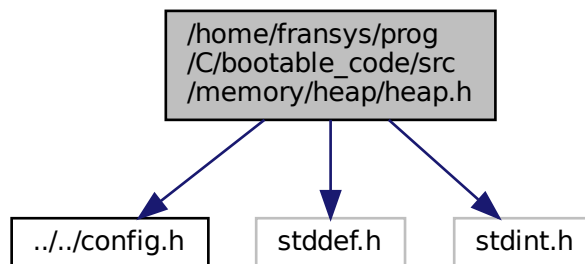
<i>heap</i>	Pointer to the heap object. This heap contains the block table to be updated.
<i>start_block</i>	The index of the first block to be marked as taken.
<i>total_blocks</i>	The total number of contiguous blocks to be marked as taken.

## 4.8 /home/fransys/prog/C/bootable\_code/src/memory/heap/heap.h File Reference

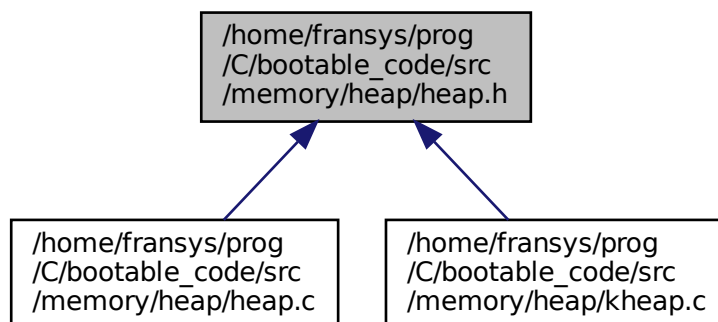
Heap management interface.

```
#include "../..//config.h"
#include <stddef.h>
#include <stdint.h>
```

Include dependency graph for heap.h:



This graph shows which files directly or indirectly include this file:



## Classes

- struct [heap\\_table](#)
- struct [heap](#)

## Macros

- #define `HEAP_BLOCK_TABLE_ENTRY_TAKEN` 0x01  
*Represents a block entry that is currently occupied.*
- #define `HEAP_BLOCK_TABLE_ENTRY_FREE` 0x00  
*Represents a block entry that is currently free.*
- #define `HEAP_BLOCK_HAS_NEXT` 0b10000000  
*Bitmask indicating that the current block has a subsequent block.*
- #define `HEAP_BLOCK_IS_FIRST` 0b01000000  
*Bitmask indicating that the block is the first in a series of blocks.*

## Typedefs

- typedef unsigned char `HEAP_BLOCK_TABLE_ENTRY`  
*Defines a type for heap block table entries.*

## Functions

- int `heap_create` (struct `heap` \*`heap`, void \*`ptr`, void \*`end`, struct `heap_table` \*`table`)  
*Initializes a heap object and its corresponding heap table.*
- void \* `heap_malloc` (struct `heap` \*`heap`, size\_t `size`)  
*Allocates a block of memory from the heap.*
- void `heap_free` (struct `heap` \*`heap`, void \*`ptr`)  
*Deallocates a block of memory from the heap.*

### 4.8.1 Detailed Description

Heap management interface.

This file declares the interface for the heap management functions. These functions include memory allocation, memory deallocation, heap initialization, and block address calculation, etc. This header file is intended to be used by kernel modules that need direct control over heap management.

### 4.8.2 Macro Definition Documentation

#### 4.8.2.1 `HEAP_BLOCK_HAS_NEXT`

```
#define HEAP_BLOCK_HAS_NEXT 0b10000000
```

Bitmask indicating that the current block has a subsequent block.

#### 4.8.2.2 HEAP\_BLOCK\_IS\_FIRST

```
#define HEAP_BLOCK_IS_FIRST 0b01000000
```

Bitmask indicating that the block is the first in a series of blocks.

#### 4.8.2.3 HEAP\_BLOCK\_TABLE\_ENTRY\_FREE

```
#define HEAP_BLOCK_TABLE_ENTRY_FREE 0x00
```

Represents a block entry that is currently free.

#### 4.8.2.4 HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN

```
#define HEAP_BLOCK_TABLE_ENTRY_TAKEN 0x01
```

Represents a block entry that is currently occupied.

### 4.8.3 Typedef Documentation

#### 4.8.3.1 HEAP\_BLOCK\_TABLE\_ENTRY

```
HEAP_BLOCK_TABLE_ENTRY
```

Defines a type for heap block table entries.

### 4.8.4 Function Documentation

#### 4.8.4.1 heap\_create()

```
int heap_create (
    struct heap * heap,
    void * ptr,
    void * end,
    struct heap_table * table )
```

Initializes a heap object and its corresponding heap table.

This function is invoked by [kheap\\_init\(\)](#) to setup a heap object. It conducts a series of validation checks and initializations to ensure the heap is ready for use.

The function begins by verifying the alignment of the start (*ptr*) and end pointers of the heap. If either pointer is not correctly aligned to the heap block size, the function aborts the heap creation process and returns `-EINVAL` to signal the alignment error.

If both pointers are correctly aligned, the function proceeds to initialize the heap object. It first wipes the heap object's memory using `memset`, setting all bytes to zero. This ensures a clean, predictable state for the new heap object. Then, it sets the start address of the heap (*saddr*) and associates the heap object with its heap table.

After initializing the heap object, the function validates the heap table by calling [heap\\_check\\_table\(\)](#). If this function reports an error (by returning a value less than 0), the function halts the creation process and returns the error code.

If the heap table is valid, the function then initializes the heap table. It calculates the size of the table in bytes and sets all entries in the heap table to indicate they're free.



**Parameters**

<i>heap</i>	The heap object to initialize. This will house all the essential data about the heap.
<i>ptr</i>	The start address of the heap. It must be aligned to the heap block size.
<i>end</i>	The end address of the heap. It also must be aligned to the heap block size.
<i>table</i>	The heap table associated with the heap. It keeps track of the state of each block in the heap.

**Returns**

int Returns 0 if the heap object and table are successfully initialized. Returns -EINVAL if an alignment or heap table check fails.

**See also**

[kheap\\_init\(\)](#)

**4.8.4.2 heap\_free()**

```
void heap_free (
    struct heap * heap,
    void * ptr )
```

Deallocates a block of memory from the heap.

The function frees up the previously allocated block of memory by marking it as free in the heap's block table. The address of the block to be freed is passed to the function as 'ptr'.

The process begins by converting the memory address 'ptr' to a block index within the heap using the helper function '[heap\\_address\\_to\\_block\(\)](#)'. The resulting block index represents the start of the block(s) that were previously allocated.

After obtaining the start block index, '[heap\\_mark\\_blocks\\_free\(\)](#)' is called to mark the associated block(s) in the heap's block table as free. This effectively deallocates the block of memory and makes it available for future allocation requests.

**Parameters**

<i>heap</i>	Pointer to the heap object from which the memory is to be deallocated.
<i>ptr</i>	Pointer to the start of the block of memory to be deallocated.

**See also**

[heap\\_mark\\_blocks\\_free\(\)](#).

#### 4.8.4.3 heap\_malloc()

```
void* heap_malloc (
    struct heap * heap,
    size_t size )
```

Allocates a block of memory from the heap.

The function begins by aligning the requested size to the heap block size. This is done using the helper function '[heap\\_align\\_value\\_to\\_upper\(\)](#)'. The alignment ensures that the allocated block of memory will start at an address that is a multiple of 'LAMEOS\_HEAP\_BLOCK\_SIZE', thereby respecting the architecture's memory alignment restrictions. This aligned size is then divided by the block size to determine the total number of blocks needed to satisfy the request.

After the total number of blocks is calculated, '[heap\\_malloc\\_blocks\(\)](#)' is called to allocate these blocks from the heap. If successful, '[heap\\_malloc\\_blocks\(\)](#)' returns a pointer to the start of the allocated memory.

##### Parameters

<i>heap</i>	Pointer to the heap object from which the memory is to be allocated.
<i>size</i>	The number of bytes to allocate.

##### Returns

void\* If the allocation is successful, a pointer to the allocated memory is returned. If the allocation fails, the return value is NULL.

##### See also

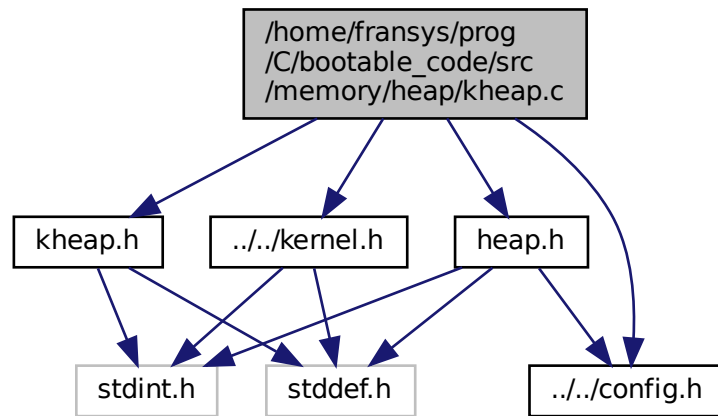
[heap\\_malloc\\_blocks\(\)](#).

## 4.9 /home/fransys/prog/C/bootable\_code/src/memory/heap/kheap.c File Reference

Kernel heap management implementation.

```
#include "kheap.h"
#include "../config.h"
#include "../kernel.h"
#include "heap.h"
```

Include dependency graph for kheap.c:



## Functions

- void [kheap\\_init](#) ()  
*Initializes the kernel heap.*
- void \* [kmalloc](#) (size\_t size)  
*Allocates memory from the kernel heap.*
- void [kfree](#) (void \*ptr)  
*Frees memory on the kernel heap.*

## Variables

- struct [heap](#) [kernel\\_heap](#)  
*Global heap object used by the kernel.*
- struct [heap\\_table](#) [kernel\\_heap\\_table](#)  
*Global heap table used by the kernel.*

### 4.9.1 Detailed Description

Kernel heap management implementation.

This file contains the implementations for the kernel heap management functions declared in '[kheap.h](#)'. These functions include memory allocation, memory deallocation, and heap initialization functions for the kernel heap. It uses the heap management interfaces provided in '[heap.h](#)'.

### 4.9.2 Function Documentation

#### 4.9.2.1 kfree()

```
void kfree (
    void * ptr )
```

Frees memory on the kernel heap.

This function wraps the `heap_free` function, providing an interface for kernel-level memory deallocation. It will free the block of memory that the provided pointer points to, making it available again for future allocations.

##### Parameters

<i>ptr</i>	A pointer to the memory block on the heap to be freed.
------------	--

##### See also

[heap\\_free\(\)](#)

#### 4.9.2.2 kheap\_init()

```
void kheap_init ( )
```

Initializes the kernel heap.

Initializes the kernel heap and the heap table with pre-defined memory size and table addresses. If the heap creation fails, it logs a message indicating the failure.

The kernel heap size and table address are defined by constants `LAMEOS_HEAP_SIZE_BYTES`, `LAMEOS_HEAP_BLOCK_SIZE`, and `LAMEOS_HEAP_TABLE_ADDRESS`. The heap creation is done using [heap\\_create\(\)](#) function, which checks the heap alignment, heap block counts and initializes the heap table.

##### See also

[heap\\_create\(\)](#)

#### 4.9.2.3 kmalloc()

```
void* kmalloc (
    size_t size )
```

Allocates memory from the kernel heap.

This function wraps the `heap_malloc` function, providing an interface for kernel-level memory allocation. The requested size is passed to the heap manager, which will return a pointer to a block of memory of at least the requested size.

#### Parameters

size	The amount of memory, in bytes, to allocate from the heap.
------	--

#### Returns

void\* A pointer to the allocated memory on the heap. If the heap cannot fulfill the request, this will be a NULL pointer.

#### See also

[heap\\_malloc\(\)](#)

### 4.9.3 Variable Documentation

#### 4.9.3.1 kernel\_heap

```
struct heap kernel_heap
```

Global heap object used by the kernel.

This is the heap object that the kernel uses to allocate and deallocate memory.

#### 4.9.3.2 kernel\_heap\_table

```
struct heap_table kernel_heap_table
```

Global heap table used by the kernel.

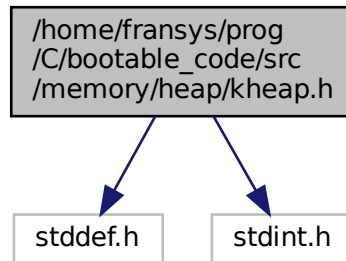
This is the heap table object that keeps track of the state of each block in the kernel heap.

## 4.10 /home/fransys/prog/C/bootable\_code/src/memory/heap/kheap.h File Reference

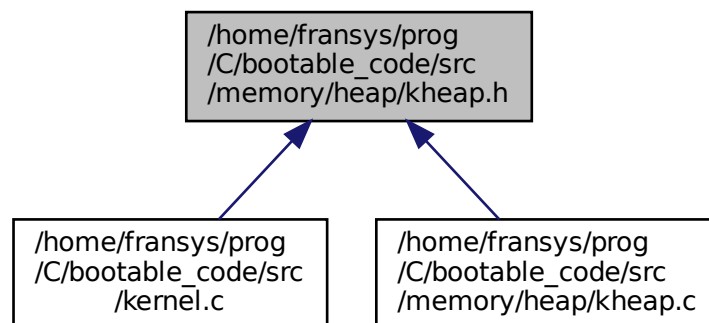
Kernel heap management interfaces.

```
#include <stddef.h>
#include <stdint.h>
```

Include dependency graph for kheap.h:



This graph shows which files directly or indirectly include this file:



## Functions

- void \* `kmalloc` (size\_t size)  
*Allocates memory from the kernel heap.*
- void `kfree` (void \*ptr)  
*Frees memory on the kernel heap.*
- void `kheap_init` ()  
*Initializes the kernel heap.*

### 4.10.1 Detailed Description

Kernel heap management interfaces.

This file declares the functions used for managing the kernel heap. This includes initialization of the heap, as well as memory allocation and deallocation. The `kmalloc` function is used to allocate memory, and `kfree` is used to free previously allocated memory. The `kheap_init` function is used to initialize the heap.

## 4.10.2 Function Documentation

### 4.10.2.1 kfree()

```
void kfree (
    void * ptr )
```

Frees memory on the kernel heap.

This function wraps the `heap_free` function, providing an interface for kernel-level memory deallocation. It will free the block of memory that the provided pointer points to, making it available again for future allocations.

#### Parameters

<i>ptr</i>	A pointer to the memory block on the heap to be freed.
------------	--

#### See also

[heap\\_free\(\)](#)

### 4.10.2.2 kheap\_init()

```
void kheap_init ( )
```

Initializes the kernel heap.

Initializes the kernel heap and the heap table with pre-defined memory size and table addresses. If the heap creation fails, it logs a message indicating the failure.

The kernel heap size and table address are defined by constants `LAMEOS_HEAP_SIZE_BYTES`, `LAMEOS_HEAP_BLOCK_SIZE`, and `LAMEOS_HEAP_TABLE_ADDRESS`. The heap creation is done using [heap\\_create\(\)](#) function, which checks the heap alignment, heap block counts and initializes the heap table.

#### See also

[heap\\_create\(\)](#)

### 4.10.2.3 kmalloc()

```
void* kmalloc (
    size_t size )
```

Allocates memory from the kernel heap.

This function wraps the `heap_malloc` function, providing an interface for kernel-level memory allocation. The requested size is passed to the heap manager, which will return a pointer to a block of memory of at least the requested size.

#### Parameters

<i>size</i>	The amount of memory, in bytes, to allocate from the heap.
-------------	--

#### Returns

void\* A pointer to the allocated memory on the heap. If the heap cannot fulfill the request, this will be a NULL pointer.

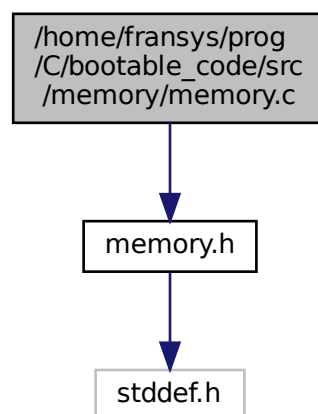
#### See also

[heap\\_malloc\(\)](#)

## 4.11 /home/fransys/prog/C/bootable\_code/src/memory/memory.c File Reference

```
#include "memory.h"
```

Include dependency graph for memory.c:



### Functions

- void \* [memset](#) (void \*ptr, int c, size\_t size)  
*Generic memset implementation.*

#### 4.11.1 Function Documentation



#### 4.11.1.1 memset()

```
void* memset (
    void * ptr,
    int c,
    size_t size )
```

Generic memset implementation.

Takes a void pointer ptr to a memory location, an int c to fill each byte with, and a size\_t size to fill to. (size\_t is the loop parameter).

##### Parameters

<i>ptr</i>	
<i>c</i>	
<i>size</i>	

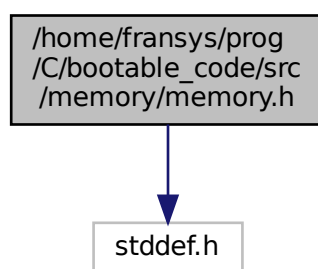
##### Returns

void\*

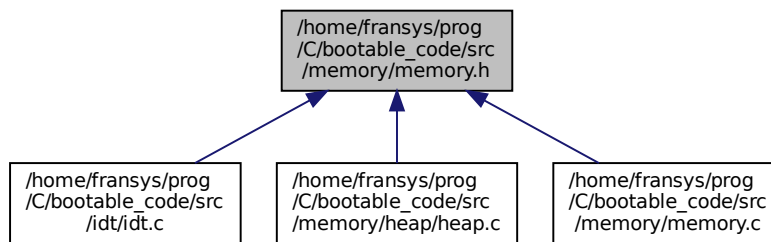
## 4.12 /home/fransys/prog/C/bootable\_code/src/memory/memory.h File Reference

```
#include <stddef.h>
```

Include dependency graph for memory.h:



This graph shows which files directly or indirectly include this file:



## Functions

- void \* [memset](#) (void \*ptr, int c, size\_t size)

*Generic memset implementation.*

### 4.12.1 Function Documentation

#### 4.12.1.1 memset()

```
void* memset (
    void * ptr,
    int c,
    size_t size )
```

Generic memset implementation.

Takes a void pointer ptr to a memory location, an int c to fill each byte with, and a size\_t size to fill to. (size\_t is the loop parameter).

#### Parameters

<i>ptr</i>	
<i>c</i>	
<i>size</i>	

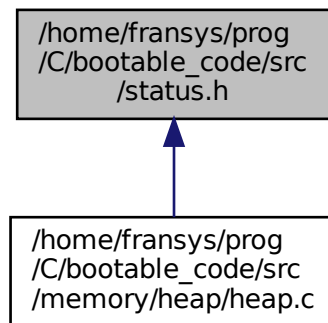
#### Returns

void\*

## 4.13 /home/fransys/prog/C/bootable\_code/src/status.h File Reference

Defines system-level status codes.

This graph shows which files directly or indirectly include this file:



## Macros

- `#define LAMEOS_OK 0`  
*A status code representing successful completion.*
- `#define EIO 1`  
*A status code representing an input/output error.*
- `#define EINVAL 2`  
*A status code representing an invalid argument error.*
- `#define ENOMEM 3`  
*A status code representing an out of memory error.*

### 4.13.1 Detailed Description

Defines system-level status codes.

This file contains definitions for various system-level status codes that can be returned by various parts of the operating system, such as the kernel or device drivers.

### 4.13.2 Macro Definition Documentation

#### 4.13.2.1 EINVAL

```
#define EINVAL 2
```

A status code representing an invalid argument error.

#### 4.13.2.2 EIO

```
#define EIO 1
```

A status code representing an input/output error.

#### 4.13.2.3 ENOMEM

```
#define ENOMEM 3
```

A status code representing an out of memory error.

#### 4.13.2.4 LAMEOS\_OK

```
#define LAMEOS_OK 0
```

A status code representing successful completion.

# Index

/home/fransys/prog/C/bootable\_code/src/config.h, 9  
/home/fransys/prog/C/bootable\_code/src/idt/idt.c, 11  
/home/fransys/prog/C/bootable\_code/src/idt/idt.h, 15  
/home/fransys/prog/C/bootable\_code/src/io/io.h, 18  
/home/fransys/prog/C/bootable\_code/src/kernel.c, 20  
/home/fransys/prog/C/bootable\_code/src/kernel.h, 24  
/home/fransys/prog/C/bootable\_code/src/memory/heap/heap.c, 28  
/home/fransys/prog/C/bootable\_code/src/memory/heap/heap.h, 40  
/home/fransys/prog/C/bootable\_code/src/memory/heap/kheap.c, 44  
/home/fransys/prog/C/bootable\_code/src/memory/heap/kheap.h, 47  
/home/fransys/prog/C/bootable\_code/src/memory/memory.c, 50  
/home/fransys/prog/C/bootable\_code/src/memory/memory.h, 51  
/home/fransys/prog/C/bootable\_code/src/status.h, 52  
\_\_attribute\_\_  
    idt.h, 16  
base  
    idt.h, 17  
    idtr\_desc, 8  
config.h  
    KERNEL\_CODE\_SELECTOR, 10  
    KERNEL\_DATA\_SELECTOR, 10  
    LAMEOS\_HEAP\_ADDRESS, 10  
    LAMEOS\_HEAP\_BLOCK\_SIZE, 10  
    LAMEOS\_HEAP\_SIZE\_BYTES, 10  
    LAMEOS\_HEAP\_TABLE\_ADDRESS, 10  
    LAMEOS\_TOTAL\_INTERRUPTS, 10  
EINVARC  
    status.h, 53  
EIO  
    status.h, 53  
ENOMEM  
    status.h, 54  
entries  
    heap\_table, 6  
heap, 5  
    saddr, 5  
    table, 5  
heap.c  
    heap\_address\_to\_block, 30  
    heap\_align\_value\_to\_upper, 30  
    heap\_block\_to\_address, 31  
    heap\_check\_alignment, 31  
    heap\_check\_table, 32  
    heap\_create, 33  
    heap\_free, 34  
    heap\_get\_entry\_type, 34  
    heap\_get\_start\_block, 36  
    heap\_malloc, 36  
    heap\_mark\_blocks\_free, 39  
    heap\_mark\_blocks\_taken, 39  
    heap.h  
        HEAP\_BLOCK\_HAS\_NEXT, 41  
        HEAP\_BLOCK\_IS\_FIRST, 41  
        HEAP\_BLOCK\_TABLE\_ENTRY, 42  
        HEAP\_BLOCK\_TABLE\_ENTRY\_FREE, 42  
        HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN, 42  
        heap\_create, 42  
        heap\_free, 43  
        heap\_malloc, 43  
    heap\_address\_to\_block  
        heap.c, 30  
    heap\_align\_value\_to\_upper  
        heap.c, 30  
    HEAP\_BLOCK\_HAS\_NEXT  
        heap.h, 41  
    HEAP\_BLOCK\_IS\_FIRST  
        heap.h, 41  
    HEAP\_BLOCK\_TABLE\_ENTRY  
        heap.h, 42  
    HEAP\_BLOCK\_TABLE\_ENTRY\_FREE  
        heap.h, 42  
    HEAP\_BLOCK\_TABLE\_ENTRY\_TAKEN  
        heap.h, 42  
    heap\_block\_to\_address  
        heap.c, 31  
    heap\_check\_alignment  
        heap.c, 31  
    heap\_check\_table  
        heap.c, 32  
    heap\_create  
        heap.c, 33  
        heap.h, 42  
    heap\_free  
        heap.c, 34  
        heap.h, 43  
    heap\_get\_entry\_type  
        heap.c, 34  
    heap\_get\_start\_block

- heap.c, [36](#)
- heap\_malloc
  - heap.c, [36](#)
  - heap.h, [43](#)
- heap\_malloc\_blocks
  - heap.c, [37](#)
- heap\_mark\_blocks\_free
  - heap.c, [39](#)
- heap\_mark\_blocks\_taken
  - heap.c, [39](#)
- heap\_table, [6](#)
  - entries, [6](#)
  - total, [6](#)
- idt.c
  - idt\_descriptors, [14](#)
  - idt\_init, [12](#)
  - idt\_load, [12](#)
  - idt\_set, [13](#)
  - idt\_zero, [13](#)
  - idtr\_descriptor, [14](#)
  - int21h, [13](#)
  - int21h\_handler, [13](#)
  - no\_interrupt, [14](#)
  - no\_interrupt\_handler, [14](#)
- idt.h
  - \_\_attribute\_\_, [16](#)
  - base, [17](#)
  - idt\_init, [16](#)
  - limit, [17](#)
  - offset\_1, [17](#)
  - offset\_2, [17](#)
  - selector, [17](#)
  - type\_attr, [17](#)
  - zero, [17](#)
- idt\_desc, [6](#)
  - offset\_1, [7](#)
  - offset\_2, [7](#)
  - selector, [7](#)
  - type\_attr, [7](#)
  - zero, [7](#)
- idt\_descriptors
  - idt.c, [14](#)
- idt\_init
  - idt.c, [12](#)
  - idt.h, [16](#)
- idt\_load
  - idt.c, [12](#)
- idt\_set
  - idt.c, [13](#)
- idt\_zero
  - idt.c, [13](#)
- idtr\_desc, [8](#)
  - base, [8](#)
  - limit, [8](#)
- idtr\_descriptor
  - idt.c, [14](#)
- insb
  - io.h, [18](#)
- insw
  - io.h, [19](#)
- int21h
  - idt.c, [13](#)
- int21h\_handler
  - idt.c, [13](#)
- io.h
  - insb, [18](#)
  - insw, [19](#)
  - outb, [19](#)
  - outw, [20](#)
- kernel.c
  - kernel\_main, [21](#)
  - lame\_color\_show, [21](#)
  - print, [21](#)
  - strlen, [22](#)
  - term\_col, [24](#)
  - term\_initialize, [22](#)
  - term\_make\_char, [22](#)
  - term\_putchar, [23](#)
  - term\_row, [24](#)
  - term\_writechar, [23](#)
  - video\_mem, [24](#)
- kernel.h
  - kernel\_main, [26](#)
  - lame\_color\_show, [26](#)
  - print, [26](#)
  - strlen, [26](#)
  - term\_initialize, [27](#)
  - term\_make\_char, [27](#)
  - term\_putchar, [27](#)
  - term\_writechar, [28](#)
  - VGA\_HEIGHT, [25](#)
  - VGA\_WIDTH, [25](#)
- KERNEL\_CODE\_SELECTOR
  - config.h, [10](#)
- KERNEL\_DATA\_SELECTOR
  - config.h, [10](#)
- kernel\_heap
  - kheap.c, [47](#)
- kernel\_heap\_table
  - kheap.c, [47](#)
- kernel\_main
  - kernel.c, [21](#)
  - kernel.h, [26](#)
- kfree
  - kheap.c, [45](#)
  - kheap.h, [49](#)
- kheap.c
  - kernel\_heap, [47](#)
  - kernel\_heap\_table, [47](#)
  - kfree, [45](#)
  - kheap\_init, [46](#)
  - kmalloc, [46](#)
- kheap.h
  - kfree, [49](#)
  - kheap\_init, [49](#)
  - kmalloc, [49](#)

kheap\_init  
  kheap.c, 46  
  kheap.h, 49

kmalloc  
  kheap.c, 46  
  kheap.h, 49

lame\_color\_show  
  kernel.c, 21  
  kernel.h, 26

LAMEOS\_HEAP\_ADDRESS  
  config.h, 10

LAMEOS\_HEAP\_BLOCK\_SIZE  
  config.h, 10

LAMEOS\_HEAP\_SIZE\_BYTES  
  config.h, 10

LAMEOS\_HEAP\_TABLE\_ADDRESS  
  config.h, 10

LAMEOS\_OK  
  status.h, 54

LAMEOS\_TOTAL\_INTERRUPTS  
  config.h, 10

limit  
  idt.h, 17  
  idt\_desc, 8

memory.c  
  memset, 50

memory.h  
  memset, 52

memset  
  memory.c, 50  
  memory.h, 52

no\_interrupt  
  idt.c, 14

no\_interrupt\_handler  
  idt.c, 14

offset\_1  
  idt.h, 17  
  idt\_desc, 7

offset\_2  
  idt.h, 17  
  idt\_desc, 7

outb  
  io.h, 19

outw  
  io.h, 20

print  
  kernel.c, 21  
  kernel.h, 26

saddr  
  heap, 5

selector  
  idt.h, 17  
  idt\_desc, 7

status.h

EINVAR, 53

EIO, 53

ENOMEM, 54

LAMEOS\_OK, 54

strlen  
  kernel.c, 22  
  kernel.h, 26

table  
  heap, 5

term\_col  
  kernel.c, 24

term\_initialize  
  kernel.c, 22  
  kernel.h, 27

term\_make\_char  
  kernel.c, 22  
  kernel.h, 27

term\_putchar  
  kernel.c, 23  
  kernel.h, 27

term\_row  
  kernel.c, 24

term\_writechar  
  kernel.c, 23  
  kernel.h, 28

total  
  heap\_table, 6

type\_attr  
  idt.h, 17  
  idt\_desc, 7

VGA\_HEIGHT  
  kernel.h, 25

VGA\_WIDTH  
  kernel.h, 25

video\_mem  
  kernel.c, 24

zero  
  idt.h, 17  
  idt\_desc, 7