# libSTD Documentation

*Release 0.8.5*

**ANSSI**

**May 17, 2019**

# CONTENTS

**Contents**

CHAPTER

ONE

# OVERVIEW

EwoK standard library is the EwoK microkernel userspace small libc implementation, hosting:

- The userspace syscall part
- The various embedded-specific utility functions (such as registers manipulation helpers)
- Some various basic functions for string manipulation, etc.

CHAPTER

TWO

# API

Libstd **does not aim** to be a POSIX compliant library. Nevertheless, for functions that behave like POSIX ones, libstd try to keep the POSIX conformant API.

Each function is described bellow.

## 2.1 aprintf

Asynchronous pretty printing

### 2.1.1 Synopsys

The *aprintf* functions familly is an asynchronous implementation of the printf function, which permits to specific embedded components such as ISR handlers to pretty print content without requesting kernel scheduling and blocking execution.

The printing phase write content in a ring buffer and the flush phase is handled voluntary, using aprintf_flush() function.

**Caution:** aprintf_flush() is a blocking function and can't be executed in ISR mode

`aprintf()` is based on a ring buffer, and as is, can't print a big amount of content between two flushes.

### 2.1.2 Usage

The aprintf API respects the following prototypes:

```
#include "api/print.h"

void aprintf(const char*fmt, ...);
void aprintf_flush(void);
```

## 2.2 aprintf

Asynchronous pretty printing

### 2.2.1 Synopsys

The *aprintf* functions familly is an asynchronous implementation of the printf function, which permits to specific embedded components such as ISR handlers to pretty print content without requesting kernel scheduling and blocking execution.

The printing phase write content in a ring buffer and the flush phase is handled voluntary, using aprintf_flush() function.

> **Caution:** aprintf_flush() is a blocking function and can't be executed in ISR mode

`aprintf()` is based on a ring buffer, and as is, can't print a big amount of content between two flushes.

### 2.2.2 Usage

The aprintf API respects the following prototypes:

```
#include "api/print.h"

void aprintf(const char*fmt, ...);
void aprintf_flush(void);
```

## 2.3 get_random

Random source accessor

### 2.3.1 Synopsis

Get back some random content from the system entropy source.

> **Caution:** get_random() requires the PERM_RES_DEV_RNG permission to request random content from the KRNG entropy source

### 2.3.2 Description

get_random load random content from the system entropy source into a buffer

get_random returns:

- MBED_ERROR_NONE if the RNG source fullfill the buffer, or:
- MBED_ERROR_DENIED if the task is not authorized to request RNG source
- MBED_ERROR_BUSY if the RNG source entropy is not ready
- MBED_ERROR_INVPARAM if len is not 32bit aligned or the buffer is NULL.

get_random() has the following API:

```
#include "api/random.h"

mbed_error_t get_random(uint8_t *buffer, uint16_t len);
```

## 2.4 Bitiwse and register operations

Manipulating registers and bitfields

### 2.4.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.4.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address
- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

## 2.5 hexdump

Print binary values in hexadecimal.

## 2.6 Bitiwse operations

Manipulate endianess

**Note:** The libstd's libARPA implementation is **not** a network stack implementation but an implementation of the minimalist network bytes order encoding/decoding API.

### 2.6.1 Synopsis

When writing a protocol stack that need to interact with the external word, the information encoding (MSB, LSB) may not be the host encoding.

The host encoding varies, depending on the hardware architecture and sometimes the system configuration (e.g. ARM, which support both MSB and LSB encoding).

To guarantee the correct encoding of numerical data, the POSIX standard has defined an API historically defined for network communications.

The Libstd API implements this API and permits its usage for any encoding conformance usage.

The endianess manipulation API is the following

```
#include "api/arpa/inet.h"

uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
uint16_t ntohs(uint16_t netshort);
uint32_t ntohl(uint32_t netlong);
```

### 2.6.2 Description

`htons()` and `htonl()` convert a short integer and a long integer into a MSB encoded value.

`ntohs()` and `ntohl()` convert a short integer and a long integer from a MSB encoded value into the host endianess encoding.

## 2.7 Bitiwse operations

Manipulate endianess

**Note:** The libstd's libARPA implementation is **not** a network stack implementation but an implementation of the minimalist network bytes order encoding/decoding API.

### 2.7.1 Synopsis

When writing a protocol stack that need to interact with the external word, the information encoding (MSB, LSB) may not be the host encoding.

The host encoding varies, depending on the hardware architecture and sometimes the system configuration (e.g. ARM, which support both MSB and LSB encoding).

To guarantee the correct encoding of numerical data, the POSIX standard has defined an API historically defined for network communications.

The Libstd API implements this API and permits its usage for any encoding conformance usage.

The endianess manipulation API is the following

```
#include "api/arpa/inet.h"

uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
uint16_t ntohs(uint16_t netshort);
uint32_t ntohl(uint32_t netlong);
```

### 2.7.2 Description

`htons()` and `htonl()` convert a short integer and a long integer into a MSB encoded value.

`ntohs()` and `ntohl()` convert a short integer and a long integer from a MSB encoded value into the host endianess encoding.

## 2.8 memcmp

### 2.8.1 Synopsys

## 2.9 mutexes

Mutual exclusive locks

### 2.9.1 Synopsys

The mutex functions family allows to use a variable to detect a mutually exclusive lock. This permit to protect critical sections of code between two concurrent threads.

In mutexes, unlike semaphores, there is no counter notions. The mutex can be locked or free. When a function try to lock the mutex, it might fail, if a preemption happen just between the mutex load and store. In this very case, the failure is detected and the lock failure can be handled by the calling code.

The mutex API respects the following prototypes:

```
#include "api/semaphore.h"

void mutex_init(volatile uint32_t *mutex);

bool mutex_trylock(volatile uint32_t *mutex);

void mutex_lock(volatile uint32_t *mutex);

bool mutex_tryunlock(uint32_t *mutex);
```

(continues on next page)

(continued from previous page)

```
void mutex_unlock(uint32_t *mutex);
```

All mutex API but the mutex_lock() and mutex_unlock() functions are non-blocking functions. mutex_lock() and mutex_unlock() block the caller until the mutex is free to be acceded exclusively.

**Danger:** Exclusive store may fail on ARM systems, even when releasing a mutex. This behavior may happen when an exception arrise during the execution of the overall __STREX intrinsic. Retrying to unlock just after should be enough is nearly all the times

It is possible to use multiple mutexes in the same time.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using mutexes and lock mechanisms in your software

**Caution:** The mutex **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the mutexes is using specific synchronisation instructions

## 2.10 mutexes

Mutual exclusive locks

### 2.10.1 Synopsys

The mutex functions family allows to use a variable to detect a mutually exclusive lock. This permit to protect critical sections of code between two concurrent threads.

In mutexes, unlike semaphores, there is no counter notions. The mutex can be locked or free. When a function try to lock the mutex, it might fail, if a preemption happen just between the mutex load and store. In this very case, the failure is detected and the lock failure can be handled by the calling code.

The mutex API respects the following prototypes:

```
#include "api/semaphore.h"

void mutex_init(volatile uint32_t *mutex);

bool mutex_trylock(volatile uint32_t *mutex);

void mutex_lock(volatile uint32_t *mutex);

bool mutex_tryunlock(uint32_t *mutex);

void mutex_unlock(uint32_t *mutex);
```

All mutex API but the mutex_lock() and mutex_unlock() functions are non-blocking functions. mutex_lock() and mutex_unlock() block the caller until the mutex is free to be acceded exclusively.

**Danger:** Exclusive store may fail on ARM systems, even when releasing a mutex. This behavior may happen when an exception arrise during the execution of the overall __STREX intrinsic. Retrying to unlock just after should be enough is nearly all the times

It is possible to use multiple mutexes in the same time.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using mutexes and lock mechanisms in your software

**Caution:** The mutex **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the mutexes is using specific synchronisation instructions

## 2.11 mutexes

Mutual exclusive locks

### 2.11.1 Synopsys

The mutex functions family allows to use a variable to detect a mutually exclusive lock. This permit to protect critical sections of code between two concurrent threads.

In mutexes, unlike semaphores, there is no counter notions. The mutex can be locked or free. When a function try to lock the mutex, it might fail, if a preemption happen just between the mutex load and store. In this very case, the failure is detected and the lock failure can be handled by the calling code.

The mutex API respects the following prototypes:

```
#include "api/semaphore.h"

void mutex_init(volatile uint32_t *mutex);

bool mutex_trylock(volatile uint32_t *mutex);

void mutex_lock(volatile uint32_t *mutex);

bool mutex_tryunlock(uint32_t *mutex);

void mutex_unlock(uint32_t *mutex);
```

All mutex API but the mutex_lock() and mutex_unlock() functions are non-blocking functions. mutex_lock() and mutex_unlock() block the caller until the mutex is free to be acceded exclusively.

**Danger:** Exclusive store may fail on ARM systems, even when releasing a mutex. This behavior may happen when an exception arrise during the execution of the overall __STREX intrinsic. Retrying to unlock just after should be enough is nearly all the times

It is possible to use multiple mutexes in the same time.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using mutexes and lock mechanisms in your software

**Caution:** The mutex **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the mutexes is using specific synchronisation instructions

## 2.12 mutexes

Mutual exclusive locks

### 2.12.1 Synopsys

The mutex functions family allows to use a variable to detect a mutually exclusive lock. This permit to protect critical sections of code between two concurrent threads.

In mutexes, unlike semaphores, there is no counter notions. The mutex can be locked or free. When a function try to lock the mutex, it might fail, if a preemption happen just between the mutex load and store. In this very case, the failure is detected and the lock failure can be handled by the calling code.

The mutex API respects the following prototypes:

```
#include "api/semaphore.h"

void mutex_init(volatile uint32_t *mutex);

bool mutex_trylock(volatile uint32_t *mutex);

void mutex_lock(volatile uint32_t *mutex);

bool mutex_tryunlock(uint32_t *mutex);

void mutex_unlock(uint32_t *mutex);
```

All mutex API but the mutex_lock() and mutex_unlock() functions are non-blocking functions. mutex_lock() and mutex_unlock() block the caller until the mutex is free to be acceded exclusively.

**Danger:** Exclusive store may fail on ARM systems, even when releasing a mutex. This behavior may happen when an exception arrise during the execution of the overall __STREX intrinsic. Retrying to unlock just after should be enough is nearly all the times

It is possible to use multiple mutexes in the same time.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using mutexes and lock mechanisms in your software

**Caution:** The mutex **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the mutexes is using specific synchronisation instructions

## 2.13 Bitiwse operations

Manipulate endianess

**Note:** The libstd's libARPA implementation is **not** a network stack implementation but an implementation of the minimalist network bytes order encoding/decoding API.

### 2.13.1 Synopsis

When writing a protocol stack that need to interact with the external word, the information encoding (MSB, LSB) may not be the host encoding.

The host encoding varies, depending on the hardware architecture and sometimes the system configuration (e.g. ARM, which support both MSB and LSB encoding).

To guarantee the correct encoding of numerical data, the POSIX standard has defined an API historically defined for network communications.

The Libstd API implements this API and permits its usage for any encoding conformance usage.

The endianess manipulation API is the following

```
#include "api/arpa/inet.h"

uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
uint16_t ntohs(uint16_t netshort);
uint32_t ntohl(uint32_t netlong);
```

### 2.13.2 Description

`htons()` and `htonl()` convert a short integer and a long integer into a MSB encoded value.

`ntohs()` and `ntohl()` convert a short integer and a long integer from a MSB encoded value into the host endianess encoding.

## 2.14 Bitiwse operations

Manipulate endianess

**Note:** The libstd's libARPA implementation is **not** a network stack implementation but an implementation of the minimalist network bytes order encoding/decoding API.

### 2.14.1 Synopsis

When writing a protocol stack that need to interact with the external word, the information encoding (MSB, LSB) may not be the host encoding.

The host encoding varies, depending on the hardware architecture and sometimes the system configuration (e.g. ARM, which support both MSB and LSB encoding).

To guarantee the correct encoding of numerical data, the POSIX standard has defined an API historically defined for network communications.

The Libstd API implements this API and permits its usage for any encoding conformance usage.

The endianess manipulation API is the following

```
#include "api/arpa/inet.h"

uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
uint16_t ntohs(uint16_t netshort);
uint32_t ntohl(uint32_t netlong);
```

### 2.14.2 Description

`htons()` and `htonl()` convert a short integer and a long integer into a MSB encoded value.

`ntohs()` and `ntohl()` convert a short integer and a long integer from a MSB encoded value into the host endianess encoding.

## 2.15 printf

formated output conversion

### 2.15.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.15.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.15.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.16 queue

Data queuing and enqueuing API

### 2.16.1 Synopsys

This implementation is a libstd specific complex storage data queue management. It behave as a FIFO implementation for various objects. It can be used, for exemple, as a media for complex messages passing between threads or to hold information in memory.

The libembed queue API is the following:

```
#include "api/queue.h"

mbed_error_t queue_create(uint32_t capacity, queue_t **queue);
mbed_error_t queue_enqueue(queue_t *q, void *data);
```

(continues on next page)

(continued from previous page)

```
mbed_error_t queue_dequeue(queue_t *q, void **data);
bool queue_is_empty(queue_t *q);
mbed_error_t queue_available_space(queue_t *q, uint32_t *space);
```

### 2.16.2 Description

- *queue_create()* create an empty queue of the given size
- *queue_enqueue()* add an element in the queue
- *queue_dequeue()* dequeue the next element, if it exists
- *queue_next_element()* get the next element of the queue given in argument, if it exists, without dequeuing it
- *queue_is_empty()* return the empty state of the queue
- *queue_available_space()* get the remaining free slots count of the given queue

The Queue API is thread-safe and reentrant.

The Queue API is using the libstd allocator to manipulate the storage backend. All data are stored in the task's heap.

for function returning mbed_error_t return type, queue API may returns:

- MBED_ERROR_NONE: function completed successfully
- MBED_ERROR_NOMEM: the allocator failed to allocate the requested memory on the head
- MBED_ERROR_NOSTORAGE: the requested element is not in the queue
- MBED_ERROR_BUSY: the queue is locked by another thread

## 2.17 queue

Data queuing and enqueuing API

### 2.17.1 Synopsys

This implementation is a libstd specific complex storage data queue management. It behave as a FIFO implementation for various objects. It can be used, for exemple, as a media for complex messages passing between threads or to hold information in memory.

The libembed queue API is the following:

```
#include "api/queue.h"

mbed_error_t queue_create(uint32_t capacity, queue_t **queue);
mbed_error_t queue_enqueue(queue_t *q, void *data);
mbed_error_t queue_dequeue(queue_t *q, void **data);
bool queue_is_empty(queue_t *q);
mbed_error_t queue_available_space(queue_t *q, uint32_t *space);
```

### 2.17.2 Description

- *queue_create()* create an empty queue of the given size
- *queue_enqueue()* add an element in the queue
- *queue_dequeue()* dequeue the next element, if it exists
- *queue_next_element()* get the next element of the queue given in argument, if it exists, without dequeuing it
- *queue_is_empty()* return the empty state of the queue
- *queue_available_space()* get the remaining free slots count of the given queue

The Queue API is thread-safe and reentrant.

The Queue API is using the libstd allocator to manipulate the storage backend. All data are stored in the task's heap.

for function returning mbed_error_t return type, queue API may returns:

- MBED_ERROR_NONE: function completed successfully
- MBED_ERROR_NOMEM: the allocator failed to allocate the requested memory on the head
- MBED_ERROR_NOSTORAGE: the requested element is not in the queue
- MBED_ERROR_BUSY: the queue is locked by another thread

## 2.18 queue

Data queuing and enqueuing API

### 2.18.1 Synopsys

This implementation is a libstd specific complex storage data queue management. It behave as a FIFO implementation for various objects. It can be used, for exemple, as a media for complex messages passing between threads or to hold information in memory.

The libembed queue API is the following:

```
#include "api/queue.h"

mbed_error_t queue_create(uint32_t capacity, queue_t **queue);
mbed_error_t queue_enqueue(queue_t *q, void *data);
mbed_error_t queue_dequeue(queue_t *q, void **data);
bool queue_is_empty(queue_t *q);
mbed_error_t queue_available_space(queue_t *q, uint32_t *space);
```

### 2.18.2 Description

- *queue_create()* create an empty queue of the given size
- *queue_enqueue()* add an element in the queue
- *queue_dequeue()* dequeue the next element, if it exists
- *queue_next_element()* get the next element of the queue given in argument, if it exists, without dequeuing it
- *queue_is_empty()* return the empty state of the queue

- *queue_available_space()* get the remaining free slots count of the given queue

The Queue API is thread-safe and reentrant.

The Queue API is using the libstd allocator to manipulate the storage backend. All data are stored in the task's heap.

for function returning mbed_error_t return type, queue API may returns:

- MBED_ERROR_NONE: function completed successfully
- MBED_ERROR_NOMEM: the allocator failed to allocate the requested memory on the head
- MBED_ERROR_NOSTORAGE: the requested element is not in the queue
- MBED_ERROR_BUSY: the queue is locked by another thread

## 2.19 queue

Data queuing and enqueuing API

### 2.19.1 Synopsys

This implementation is a libstd specific complex storage data queue management. It behave as a FIFO implementation for various objects. It can be used, for exemple, as a media for complex messages passing between threads or to hold information in memory.

The libembed queue API is the following:

```
#include "api/queue.h"

mbed_error_t queue_create(uint32_t capacity, queue_t **queue);
mbed_error_t queue_enqueue(queue_t *q, void *data);
mbed_error_t queue_dequeue(queue_t *q, void **data);
bool queue_is_empty(queue_t *q);
mbed_error_t queue_available_space(queue_t *q, uint32_t *space);
```

### 2.19.2 Description

- *queue_create()* create an empty queue of the given size
- *queue_enqueue()* add an element in the queue
- *queue_dequeue()* dequeue the next element, if it exists
- *queue_next_element()* get the next element of the queue given in argument, if it exists, without dequeuing it
- *queue_is_empty()* return the empty state of the queue
- *queue_available_space()* get the remaining free slots count of the given queue

The Queue API is thread-safe and reentrant.

The Queue API is using the libstd allocator to manipulate the storage backend. All data are stored in the task's heap.

for function returning mbed_error_t return type, queue API may returns:

- MBED_ERROR_NONE: function completed successfully
- MBED_ERROR_NOMEM: the allocator failed to allocate the requested memory on the head

- MBED_ERROR_NOSTORAGE: the requested element is not in the queue
- MBED_ERROR_BUSY: the queue is locked by another thread

## 2.20 queue

Data queuing and enqueuing API

### 2.20.1 Synopsys

This implementation is a libstd specific complex storage data queue management. It behave as a FIFO implementation for various objects. It can be used, for exemple, as a media for complex messages passing between threads or to hold information in memory.

The libembed queue API is the following:

```
#include "api/queue.h"

mbed_error_t queue_create(uint32_t capacity, queue_t **queue);
mbed_error_t queue_enqueue(queue_t *q, void *data);
mbed_error_t queue_dequeue(queue_t *q, void **data);
bool queue_is_empty(queue_t *q);
mbed_error_t queue_available_space(queue_t *q, uint32_t *space);
```

### 2.20.2 Description

- *queue_create()* create an empty queue of the given size
- *queue_enqueue()* add an element in the queue
- *queue_dequeue()* dequeue the next element, if it exists
- *queue_next_element()* get the next element of the queue given in argument, if it exists, without dequeuing it
- *queue_is_empty()* return the empty state of the queue
- *queue_available_space()* get the remaining free slots count of the given queue

The Queue API is thread-safe and reentrant.

The Queue API is using the libstd allocator to manipulate the storage backend. All data are stored in the task's heap.

for function returning mbed_error_t return type, queue API may returns:

- MBED_ERROR_NONE: function completed successfully
- MBED_ERROR_NOMEM: the allocator failed to allocate the requested memory on the head
- MBED_ERROR_NOSTORAGE: the requested element is not in the queue
- MBED_ERROR_BUSY: the queue is locked by another thread

## 2.21 Bitiwse and register operations

Manipulating registers and bitfields

### 2.21.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.21.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address
- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

## 2.22 Bitiwse and register operations

Manipulating registers and bitfields

### 2.22.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.22.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address
- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

## 2.23 semaphore

Synchronization primitives with counters

### 2.23.1 Synopsis

The semaphore functions family allows to use an integer variable to detect a lock. The variable holds a counter that is set when calling the semaphore_init() function.

If the counter is greater than 0, the semaphore_trylock() will try to decrease the counter using the arch-specific exclusive atomic write on the counter. If the counter is correctly updated, the function returns true. If the counter is already set to 0 or the update fails due to a potential race condition with another thread during the update, the function returns false.

The semaphore_release() function should be called only if the semaphore_trylock() function has previously been called and has returned true. This function tries to increment the lock counter and returns true if the counter has been correctly increased, or false otherwise.

The semaphore API respects the following prototypes

```
#include "api/semaphore.h"

void semaphore_init(uint8_t value, volatile uint32_t* semaphore);
bool semaphore_trylock(volatile uint32_t* semaphore);
void semaphore_lock(volatile uint32_t* semaphore);
bool semaphore_release(volatile uint32_t* semaphore);
```

It is possible to use multiple semaphores at the same time.

It is possible to create mutually exclusive access (also known as mutex) to a variable or a specific function when setting the semaphore to 1 at initialization time. Although, you should use the mutex API instead for this last case.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using semaphore and lock mechanisms in your software

**Caution:** The semaphore **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the semaphore is using specific synchronisation instructions

**Hint:** When a lock is required between a thread and an ISR for which a treatment can't be postponed (i.e. the application whishes to avoid the lock to happend instead to detect it). The kernel sys_lock() API is a better choice

semaphore_trylock is nonblocking and try to lock the semaphore, returning false if the lock fails.

semaphore_lock is a blocking function, waiting for the samaphore to be free to be locked before returning.

## 2.24 semaphore

Synchronization primitives with counters

### 2.24.1 Synopsis

The semaphore functions family allows to use an integer variable to detect a lock. The variable holds a counter that is set when calling the semaphore_init() function.

If the counter is greater than 0, the semaphore_trylock() will try to decrease the counter using the arch-specific exclusive atomic write on the counter. If the counter is correctly updated, the function returns true. If the counter is already set to 0 or the update fails due to a potential race condition with another thread during the update, the function returns false.

The semaphore_release() function should be called only if the semaphore_trylock() function has previously been called and has returned true. This function tries to increment the lock counter and returns true if the counter has been correctly increased, or false otherwise.

The semaphore API respects the following prototypes

```
#include "api/semaphore.h"

void semaphore_init(uint8_t value, volatile uint32_t* semaphore);
bool semaphore_trylock(volatile uint32_t* semaphore);
```

(continues on next page)

(continued from previous page)

```
void semaphore_lock(volatile uint32_t* semaphore);
bool semaphore_release(volatile uint32_t* semaphore);
```

It is possible to use multiple semaphores at the same time.

It is possible to create mutually exclusive access (also known as mutex) to a variable or a specific function when setting the semaphore to 1 at initialization time. Although, you should use the mutex API instead for this last case.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using semaphore and lock mechanisms in your software

**Caution:** The semaphore **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the semaphore is using specific synchronisation instructions

**Hint:** When a lock is required between a thread and an ISR for which a treatment can't be postponed (i.e. the application whishes to avoid the lock to happend instead to detect it). The kernel sys_lock() API is a better choice

semaphore_trylock is nonblocking and try to lock the semaphore, returning false if the lock fails.

semaphore_lock is a blocking function, waiting for the samaphore to be free to be locked before returning.

## 2.25 semaphore

Synchronization primitives with counters

### 2.25.1 Synopsis

The semaphore functions family allows to use an integer variable to detect a lock. The variable holds a counter that is set when calling the semaphore_init() function.

If the counter is greater than 0, the semaphore_trylock() will try to decrease the counter using the arch-specific exclusive atomic write on the counter. If the counter is correctly updated, the function returns true. If the counter is already set to 0 or the update fails due to a potential race condition with another thread during the update, the function returns false.

The semaphore_release() function should be called only if the semaphore_trylock() function has previously been called and has returned true. This function tries to increment the lock counter and returns true if the counter has been correctly increased, or false otherwise.

The semaphore API respects the following prototypes

```
#include "api/semaphore.h"

void semaphore_init(uint8_t value, volatile uint32_t* semaphore);
bool semaphore_trylock(volatile uint32_t* semaphore);
void semaphore_lock(volatile uint32_t* semaphore);
bool semaphore_release(volatile uint32_t* semaphore);
```

It is possible to use multiple semaphores at the same time.

It is possible to create mutually exclusive access (also known as mutex) to a variable or a specific function when setting the semaphore to 1 at initialization time. Although, you should use the mutex API instead for this last case.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using semaphore and lock mechanisms in your software

**Caution:** The semaphore **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the semaphore is using specific synchronisation instructions

**Hint:** When a lock is required between a thread and an ISR for which a treatment can't be postponed (i.e. the application whishes to avoid the lock to happend instead to detect it). The kernel sys_lock() API is a better choice

semaphore_trylock is nonblocking and try to lock the semaphore, returning false if the lock fails.

semaphore_lock is a blocking function, waiting for the samaphore to be free to be locked before returning.

## 2.26 semaphore

Synchronization primitives with counters

### 2.26.1 Synopsis

The semaphore functions family allows to use an integer variable to detect a lock. The variable holds a counter that is set when calling the semaphore_init() function.

If the counter is greater than 0, the semaphore_trylock() will try to decrease the counter using the arch-specific exclusive atomic write on the counter. If the counter is correctly updated, the function returns true. If the counter is already set to 0 or the update fails due to a potential race condition with another thread during the update, the function returns false.

The semaphore_release() function should be called only if the semaphore_trylock() function has previously been called and has returned true. This function tries to increment the lock counter and returns true if the counter has been correctly increased, or false otherwise.

The semaphore API respects the following prototypes

```
#include "api/semaphore.h"

void semaphore_init(uint8_t value, volatile uint32_t* semaphore);
bool semaphore_trylock(volatile uint32_t* semaphore);
void semaphore_lock(volatile uint32_t* semaphore);
bool semaphore_release(volatile uint32_t* semaphore);
```

It is possible to use multiple semaphores at the same time.

It is possible to create mutually exclusive access (also known as mutex) to a variable or a specific function when setting the semaphore to 1 at initialization time. Although, you should use the mutex API instead for this last case.

**Caution:** There is no protection against dead-lock, you must be aware of the impact of using semaphore and lock mechanisms in your software

**Caution:** The semaphore **must** be declared as volatile to avoid any border effect associated to the compilation process. The assembler backend manipulates the semaphore is using specific synchronisation instructions

**Hint:** When a lock is required between a thread and an ISR for which a treatment can't be postponed (i.e. the application whishes to avoid the lock to happend instead to detect it). The kernel sys_lock() API is a better choice

semaphore_trylock is nonblocking and try to lock the semaphore, returning false if the lock fails.

semaphore_lock is a blocking function, waiting for the samaphore to be free to be locked before returning.

## 2.27 Bitiwse and register operations

Manipulating registers and bitfields

### 2.27.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.27.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address

- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

## 2.28 printf

formated output conversion

### 2.28.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.28.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.28.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.29 printf

formated output conversion

### 2.29.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.29.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.29.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.30 strcmp

String comparison

### 2.30.1 Synopsys

strcmp compare two (potentially NULL) strings and return a negative, null or positive value depending on the comparison.

The comparison algorithm is the following:

- If both string are NULL, strcmp returns 0
- If the s1 is NULL, strcmp returns -1
- If the s2 is NULL, strcmp returns 1
- If both string are non-NULL, the result depends on the first differenciated character:
  - If s1 char ASCII value is smaller than the second string one, strcmp returns a negative value
  - If s2 char ASCII value is smaller than the first string one, strcmp returns a positive value
  - If strings are equal (no difference found upto the leading 0), strcmp return 0

strncmp behaves like strcmp unless it compares up to len characters between the two string parameters.

strcmp respects the following prototype:

```
#include "string.h"

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, uint32_t len);
```

### 2.30.2 Caution

strcmp compare C (null-terminated) strings. strcmp arguments can be NULL but if not must finish with '0'.

### 2.30.3 Conforming to

`strcmp()` and `strncmp()` are conform to POSIX-1-2001, C09, C99, SVr4 and 4.3BSD

The behavior of these functions in case of invalid parameters (i.e. one or more in s1 and s2 is null) is considered as unpredictable in the POSIX implementation. In the libstd API, we have decided to sanitize correctly the input values.

## 2.31 strncpy

Copy a substring from one string to another

### 2.31.1 Synopsys

strcpy - copy a string

strcpy function copy a string from src to the buffer pointed by dest, including the final null byte.

strncpy function is similar to strcpy, unless it copies at most n characters (including the final null byte) from src to dest. If the length of src is less than n, strncpy() complete dest with null bytes up to n chars.

strcpy respects the following prototype:

```
#include "string.h"

char *strncpy(char *dest, const char *src, uint32_t n);
char *strcpy(char *dest, const char *src);
```

### 2.31.2 Conforming to

strcpy() and strncpy() are conform to POSIX-1-2001, C09, C99, SVr4 and 4.3BSD

The behavior of these functions in case of invalid parameters (i.e. one or more in dest and src is null) is considered as unpredictable in the POSIX implementation. In the libstd API, we have decided to sanitize correctly the input values.

## 2.32 strlen

Calculate the length of a string

### 2.32.1 Synopsys

strlen respects the following prototype:

```
#include "string.h"

uint32_t strlen(const char *str);
```

## 2.33 strcmp

String comparison

### 2.33.1 Synopsys

strcmp compare two (potentially NULL) strings and return a negative, null or positive value depending on the comparison.

The comparison algorithm is the following:

- If both string are NULL, strcmp returns 0
- If the s1 is NULL, strcmp returns -1
- If the s2 is NULL, strcmp returns 1
- If both string are non-NULL, the result depends on the first differenciated character:
  - If s1 char ASCII value is smaller than the second string one, strcmp returns a negative value
  - If s2 char ASCII value is smaller than the first string one, strcmp returns a positive value
  - If strings are equal (no difference found upto the leading 0), strcmp return 0

strncmp behaves like strcmp unless it compares up to len characters between the two string parameters.

strcmp respects the following prototype:

```
#include "string.h"

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, uint32_t len);
```

### 2.33.2 Caution

strcmp compare C (null-terminated) strings. strcmp arguments can be NULL but if not must finish with '0'.

### 2.33.3 Conforming to

`strcmp()` and `strncmp()` are conform to POSIX-1-2001, C09, C99, SVr4 and 4.3BSD

The behavior of these functions in case of invalid parameters (i.e. one or more in s1 and s2 is null) is considered as unpredictable in the POSIX implementation. In the libstd API, we have decided to sanitize correctly the input values.

## 2.34 strncpy

Copy a substring from one string to another

### 2.34.1 Synopsys

strcpy - copy a string

strcpy function copy a string from src to the buffer pointed by dest, including the final null byte.

strncpy function is similar to strcpy, unless it copies at most n characters (including the final null byte) from src to dest. If the length of src is less than n, strncpy() complete dest with null bytes up to n chars.

strcpy respects the following prototype:

```
#include "string.h"

char *strncpy(char *dest, const char *src, uint32_t n);
char *strcpy(char *dest, const char *src);
```

### 2.34.2 Conforming to

strcpy() and strncpy() are conform to POSIX-1-2001, C09, C99, SVr4 and 4.3BSD

The behavior of these functions in case of invalid parameters (i.e. one or more in dest and src is null) is considered as unpredictable in the POSIX implementation. In the libstd API, we have decided to sanitize correctly the input values.

## 2.35 printf

formated output conversion

### 2.35.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.35.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.35.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.36 printf

formated output conversion

### 2.36.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
```

(continues on next page)

(continued from previous page)

```
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.36.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.36.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.37 printf

formated output conversion

### 2.37.1 Synopsys

This implementation of printf implement a subset of the POSIX.1-2001 and C99 standard API (as we are in an embedded system).

the stdio printf API uses variatic arguments. stdard printf API uses va_list arg for dynamic arguments list.

The printf familly respects the following prototype:

```
#include "api/stdio.h"

void printf(const char *fmt, ...);
void sprintf(char *dest, const char *fmt, ...);
void snprintf(char *dest, size_t len, const char *fmt, ...);

#include "api/stdarg.h"

void vprintf(const char *fmt, va_list args);
void vsprintf(char *dest, const char *fmt, va_list args);
void vsnprintf(char *dest, size_t len, const char *fmt, va_list args);
```

### 2.37.2 Description

Even if all the printf() fmt formating is not supported, the behavior of the functions for the supported flags chars and lenght modifiers is conform to the standard. The following is supported:

| flag characters | |
|---|---|
| '0' | the value should be zero-padded. This flag is allowed for any numerical value, including pointer (p). This flag *must* be followed by a numerical, decimal value specifying the size to which the value must be padded. |

| length modifiers | |
|---|---|
| 'l' | long int conversion |
| 'll' | long long int conversion |
| 'd','i' | integer signed value conversion |
| 'u' | unsigned integer value conversion |
| 'h' | short int conversion |
| 'hh' | unsigned char conversion |
| 'x' | hexadecimal value, mapped on a long integer conversion |
| 'o' | octal value, mapped on a long integer conversion |
| 'p' | pointer, word-sized unsigned integer, starting with 0x and padded with 0 up to word-length size |
| 'c' | character value conversion |
| 's' | string conversion |

Other flags characters and length modifiers are not supported, generating an immediate stop of the fmt parsing.

### 2.37.3 Conforming to

POSIX.1-2001, POSIX.1-2008, C89, C99

## 2.38 wmalloc_init

### 2.38.1 Synopsys

wmalloc_init respects the following prototype:

```
#include "malloc.h"

void todo(unsigned long long value, uint8_t base);
```

## 2.39 wmalloc_init

### 2.39.1 Synopsys

wmalloc_init respects the following prototype:

```
#include "malloc.h"
void todo(unsigned long long value, uint8_t base);
```

## 2.40 wmalloc_init

### 2.40.1 Synopsys

wmalloc_init respects the following prototype:

```
#include "malloc.h"
void todo(unsigned long long value, uint8_t base);
```

## 2.41 Bitiwse and register operations

Manipulating registers and bitfields

### 2.41.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.41.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address
- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

## 2.42 Bitiwse and register operations

Manipulating registers and bitfields

### 2.42.1 Synopsis

When writing a driver, a part of the implementation consists in manipulating device's registers. The C language is not a pleasant language for such manipulation and mostly rely on preprocessing.

The libstd embed part implement various generic helpers to manipulate register fields without requiring explicit bitwise operation.

The register manipulation helpers respect the following API:

```
#include "api/regutils.h"

uint32_t read_reg_value(volatile uint32_t * reg);
void     write_reg_value(volatile uint32_t * reg, uint32_t value);

uint32_t get_reg_value(volatile const uint32_t * reg, uint32_t mask, uint8_t pos);
```

(continues on next page)

(continued from previous page)

```
uint8_t  set_reg_value(volatile uint32_t * reg, uint32_t value, uint32_t mask, uint8_
↪t pos);

uint16_t read_reg16_value(volatile uint16_t * reg);
void     write_reg16_value(volatile uint16_t * reg, uint16_t value);
```

### 2.42.2 Description

read_reg_value() and write_reg_value() read from and write into a 32 bits register directly, with no consideration for its potential fields. They can be used at initialize or device reset time. For half-word (16 bits) registers, read_reg16_value() and write_reg16_value() behave like read_reg_value() and write_reg_value().

get_reg_value() and set_reg_value() can get or set a given field of a register, without modifying the other fields. They use the following arguments:

- reg: the register address
- value: for set_reg_bit(), the value to set in the field (position independent)
- mask: the field mask (starting at bit 0). E.g. 0x3 means a field of 2 bits.
- pos: the field position (starting with the least significant bit, used to
- move the mask and the value to the correct position

get_reg_value() return the field value independently of its position.

These helpers are not made to be used directly on registers (volatile pointers) but on basic variables.

CHAPTER

# THREE

# FAQ

- **Are there helper functions to manipulate registers in userspace?**

Helper functions and macros have been written to access registers. This API is in the libstd `regutils.h` header. Applications can include this header directly in order to use it.