API

UWB QOSAL API

Qorvo

Release R12.7.0-405-gb33c5c4272

Contents

1 Overview

The Qorvo Operating System Abstraction Layer (QOSAL) is a general purpose operating system abstraction layer that implements OS specific functionalities. The objective of this layer is to provide a common API for Qorvo applications to abstract the operating system used, avoiding dependencies on the underlying operating system.

The selection of a specific implementation is made through the `CONFIG_QOSAL_IMPL_<impl>` cmake option. It can also be selected through Kconfig if using Zephyr OS.

In the scope of the UWB stack, the QOSAL is the common dependency of all its components and is only dependent on the Operating System with some exceptions. For example, there are some cyclic dependencies with the Qorvo Hardware Abstraction Layer (QHAL) which abstracts the hardware and the SDK.

1.1 Porting guide

The QOSAL can be reimplemented for an unsupported OS or adapted for an unsupported version of one of the supported OS as long as it follows the interface.

The specific code for an implementation is contained in the directories:

- `qosal/include/<impl>` for the specific headers
- `qosal/src/<impl>` for the sources

Warning: For delivered pre-compiled libraries built against the qosal, the `qosal/include/dummy` directory will be applied to avoid implementation contamination. It cannot be overridden after the fact due to how the macros work. This is a current limitation of this version of the qosal. For example the `QASSERT` macros cannot be redefined.

The behavior of each functions to define is descriped in their *api documentation*.

To modify the build, the file `qosal/CMakeLists.txt` shall be updated, either by adding a new `CONFIG_QOSAL_IMPL_<impl>` option, or by modifying the existing one.

Some parameters can be set as cmake option and as compile definition of the build to configure the qosal, as this parameters are in the interface, they shall be set during the build of the components that depends on it:

- `CONFIG_QOSAL_PROFILING_STATS` to enable system profiling (stack status, memory status, . . .)
- `CONFIG_QOSAL_PROFILING_MALLOC` to enable malloc profiling (use the log system) [zephyr only]
- `CONFIG_QOSAL_MAX_SIGNAL` to set the maximum of signals allowed (default: `1`) [zephyr only]
- `CONFIG_QOSAL_MAX_THREAD` to set the maximum of threads allowed (default: `2`) [zephyr only]
- `CONFIG_QOSAL_MAX_MSG_QUEUE` to set the maximum of message queues allowed (default: `1`) [zephyr only]
- `CONFIG_QOSAL_MAX_MUTEX` to set the maximum of mutex allowed (default: `3`) [zephyr only]
- `CONFIG_QLOG_LEVEL` to set the minimal log level (default: `3`) [zephyr only]
- `CONFIG_MEM_QUOTA_ID1` to set the default quota ID (default: `8192`)
- `CONFIG_MEM_QUOTA_ID2` to set the default quota ID (default: `4096`)
- `CONFIG_MEM_QUOTA_RANGING_REPORT` to set the quota ID used by ranging reports (default: `MEM_QUOTA_ID1`)
- `CONFIG_MEM_QUOTA_PSDU_REPORT` to set the quota ID used by psdu reports (default: `MEM_QUOTA_ID1`)
- `CONFIG_MEM_QUOTA_UCI_REPORT` to set the quota ID used by uci reports (default: `MEM_QUOTA_ID2`)

1.1.1 qmalloc

Our implementation of malloc uses a system of quota to limit the memory usage of some non-essential parts of our components, notably the reports containing the payload of the UWB frame or the statistics which can be in kilobytes.

The quota is implemented as a header behind the pointer returned by the allocation, which is typical from the different allocator implementations, however it requires to free the memory with the same allocator used, so memory allocated through `q[cm]alloc*` functions shall be freed with `qfree` and memory allocated by another allocator shall not be freed by `qfree`.

1.1.2 qlog

Due to a limitation, the qlog system is not available to pre-compiled libraries which have been built against the qosal in OS agnostic mode, even on Zephyr OS.

1.1.3 qassert

qassert default behavior is to use `assert()` from the C standard library, but it cannot be overridden by the integrator after the build in library mode. Furthermore, it follows the same behavior as `assert()` so in Release build mode, if `NDEBUG` is defined the assert is removed.

1.1.4 qirq

None of the qirq macros are used nor tested for non-Zephyr OS targets:

- `QIRQ_CONNECT`: not used, not implemented
- `QIRQ_ENABLE`: not used
- `QIRQ_DISABLE`: not used
- `QIRQ_CLEAR_PENDING`: not used
- `QIRQ_CLEAR_GET`: not used

`qirq_lock()` and `qirq_unlock()` supports nested locks only on Zephyr OS targets.

1.2 Supported OS

The QOSAL is tested on the following OS and versions:

- FreeRTOS v10.0.0
- FreeRTOS v9.0.0

The linux implementation is mostly used by some components on a host chip (external MCU or PC). It doesn't support the whole UWB software.

Each Qorvo UWB devices has a more restricted list of supported and tested OS:

OS	QM33 + NRF52XX	QM33 + NRF53XX	QM33 + ST NUCLEO
FreeRTOS v10.0.0	Supported	Not supported	Not supported
FreeRTOS v9.0.0	Not supported	Not supported	Not supported

2 QOSAL API

2.1 QASSERT

2.1.1 macro QASSERT

`QASSERT`(cond)

Assert macro to use, the implementation must manage if the assert is implemented or stubbed depending on the definition of NDEBUG in accordance to the C standard.

Parameters

- **cond** – Condition to be tested, if false, it doesn't return and displays an error message

2.2 QATOMIC

2.2.1 macro qatomic_int

qatomic_int()

See atomic_int in GCC documentation.

2.2.2 macro qatomic_bool

qatomic_bool()

See atomic_bool in GCC documentation.

2.2.3 macro qatomic_init

qatomic_init(x, value)

See atomic_init in GCC documentation.

Parameters

- **x** – The pointer to the qatomic_* variable to initialize.
- **value** – The value to initialize it to.

2.2.4 macro qatomic_load

qatomic_load(x)

See atomic_load in GCC documentation.

Parameters

- **x** – The pointer to the qatomic_* variable to load.

2.2.4.1 Return

The value of the variable.

2.2.5 macro qatomic_store

qatomic_store(x, value)

See atomic_store in GCC documentation.

Parameters

- **x** – The pointer to the qatomic_* variable to load.
- **value** – The value to store.

2.2.6 macro qatomic_exchange

`qatomic_exchange`(x, value)

See atomic_exchange in GCC documentation.

Parameters

- **`x`** – The pointer to the qatomic_* variable to exchange.
- **`value`** – The value to exchange it with.

2.2.6.1 Return

The old value.

2.2.7 macro qatomic_fetch_add

`qatomic_fetch_add`(x, value)

See atomic_fetch_add in GCC documentation.

Parameters

- **`x`** – The pointer to the qatomic_* variable to add to.
- **`value`** – The value to add.

2.2.7.1 Return

The old value.

2.2.8 macro qatomic_fetch_sub

`qatomic_fetch_sub`(x, value)

See atomic_fetch_sub in GCC documentation.

Parameters

- **`x`** – The pointer to the qatomic_* variable to substract to.
- **`value`** – The value to add.

2.2.8.1 Return

The old value.

2.3 QERR

2.3.1 enum qerr

enum **qerr**

Return values for most QOSAL functions.

2.3.1.1 Definition

```
enum qerr {
    QERR_SUCCESS,
    QERR_EADDRNOTAVAIL,
    QERR_EAFNOSUPPORT,
    QERR_EAGAIN,
    QERR_EBADF,
    QERR_EBADMSG,
    QERR_EBUSY,
    QERR_ECONNREFUSED,
    QERR_EEXIST,
    QERR_EFAULT,
    QERR_EINTR,
    QERR_EINVAL,
    QERR_EIO,
    QERR_EMSGSIZE,
    QERR_ENETDOWN,
    QERR_ENOBUFS,
    QERR_ENOENT,
    QERR_ENOMEM,
    QERR_ENOTSUP,
    QERR_EPERM,
    QERR_EPIPE,
    QERR_EPROTO,
    QERR_EPROTONOSUPPORT,
    QERR_ERANGE,
    QERR_ETIME,
    QERR_ENODEV,
    QERR_ENOSPC,
    QERR_SE_EINVAL,
    QERR_SE_ENOKEY,
    QERR_SE_ENOSUBKEY,
    QERR_SE_ERDSFETCHFAIL,
    QERR_SE_ECANCEL
};
```

2.3.1.2 Constants

QERR_SUCCESS
Operation successful.

QERR_EADDRNOTAVAIL
Address not available.

QERR_EAFNOSUPPORT
Address family not supported.

QERR_EAGAIN
Resource temporarily unavailable.

QERR_EBADF
Bad file descriptor.

QERR_EBADMSG
Bad message.

QERR_EBUSY
Device or resource busy.

QERR_ECONNREFUSED
Connection refused.

QERR_EEXIST
File exists.

QERR_EFAULT
Bad address.

QERR_EINTR
Interrupted system call.

QERR_EINVAL
Invalid argument.

QERR_EIO
I/O error.

QERR_EMSGSIZE
Message too long.

QERR_ENETDOWN
Network is down.

QERR_ENOBUFS
No buffer space available.

QERR_ENOENT
No such region or scheduler.

QERR_ENOMEM
Not enough memory.

QERR_ENOTSUP
Operation not supported.

QERR_EPERM
Permission denied.

QERR_EPIPE
Broken pipe.

QERR_EPROTO
Protocol error.

QERR_EPROTONOSUPPORT
Protocol not supported.

QERR_ERANGE
Result too large.

QERR_ETIME
Timer expired.

QERR_ENODEV
No such device.

QERR_ENOSPC
No space left.

QERR_SE_EINVAL
Invalid arguments given to SE.

QERR_SE_ENOKEY
No session key found for given id.

QERR_SE_ENOSUBKEY
No sub-session key found for given id.

QERR_SE_ERDSFETCHFAIL
Unexpected failure in SE while fetching keys.

QERR_SE_ECANCEL
SE acknowledges cancellation of a request.

2.3.2 qerr_convert_os_to_qerr

enum *qerr* **`qerr_convert_os_to_qerr`**(int error)

Convert error from OS specific error to *qerr*.

Parameters

- **`error`** (`int`) – Error to be converted.

2.3.2.1 Return

qerr converted from implementation-defined error.

2.3.3 qerr_convert_qerr_to_os

int **`qerr_convert_qerr_to_os`**(enum *qerr* error)

Convert error from *qerr* to OS specific error.

Parameters

- **`error`** (`enum qerr`) – Error to be converted.

2.3.3.1 Return

OS error converted from qerr error.

2.3.4 qerr_to_str

const char ***qerr_to_str**(enum *qerr* error)

Convert qerr to string. Used for printing errors.

Parameters

- **error** (enum *qerr*) – Error to be converted.

2.3.4.1 Return

qerr converted to a string.

2.4 QIRQ

2.4.1 macro QIRQ_CONNECT

QIRQ_CONNECT(irqn, prio, handler)

Configure IRQ priority and handler.

Parameters

- **irqn** – IRQ number.
- **prio** – IRQ priority.
- **handler** – IRQ handler.

2.4.2 macro QIRQ_ENABLE

QIRQ_ENABLE(irqn)

Enable IRQ.

Parameters

- **irqn** – IRQ number.

2.4.3 macro QIRQ_DISABLE

QIRQ_DISABLE(irqn)

Enable IRQ.

Parameters

- **irqn** – IRQ number.

2.4.4 macro QIRQ_CLEAR_PENDING

QIRQ_CLEAR_PENDING(irqn)

Clear pending IRQ.

Parameters

- **irqn** – IRQ number.

2.4.5 macro QIRQ_GET

QIRQ_GET(irqn)

Get IRQ.

Parameters

- **irqn** – IRQ number.

2.4.6 qirq_lock

unsigned int **qirq_lock**(void)

Disable all interrupts.

Parameters

- **void** – no arguments

2.4.6.1 Description

The use of the lock-out key to be able to nest lock/unlock is not supported depending on the implementation. It is advisable to not use this feature.

2.4.6.2 Return

A lock-out key, its representation is implementation-defined.

2.4.7 qirq_unlock

void **qirq_unlock**(unsigned int key)

Enable interrupts previously disabled by *qirq_lock()*.

Parameters

- **key** (unsigned int) – The lock-out key returned by the previous call to *qirq_lock()*.

2.4.7.1 Description

The use of the lock-out key to be able to nest lock/unlock is not supported depending on the implementation. It is advisable to not use this feature.

2.5 QLOG

2.5.1 macro QLOG_CURRENT_LEVEL

`QLOG_CURRENT_LEVEL()`

Log level to be defined by the user. Possible values are: QLOG_LEVEL_NONE, QLOG_LEVEL_ERR, QLOG_LEVEL_WARN, QLOG_LEVEL_INFO or QLOG_LEVEL_DEBUG.

2.5.2 macro LOG_TAG

`LOG_TAG()`

Log tag to be defined by the user.

2.5.3 macro QLOGD

`QLOGD(...)`

Print a debug log.

Parameters

- **`ellipsis`** (`ellipsis`) – variable arguments

2.5.4 macro QLOGE

`QLOGE(...)`

Print an error log.

Parameters

- **`ellipsis`** (`ellipsis`) – variable arguments

2.5.5 macro QLOGI

`QLOGI(...)`

Print an information log.

Parameters

- **`ellipsis`** (`ellipsis`) – variable arguments

2.5.6 macro QLOGW

`QLOGW(...)`

Print a warning log.

Parameters

- **`ellipsis`** (`ellipsis`) – variable arguments

2.6 QMALLOC

2.6.1 qmalloc_internal

void ***`qmalloc_internal`**(size_t size)

Allocate memory.

Parameters

- **`size`** (`size_t`) – Number of bytes to allocate.

2.6.1.1 Return

Pointer to the allocated memory.

2.6.2 qrealloc_internal

void ***`qrealloc_internal`**(void *ptr, size_t new_size)

Resize the already allocated memory.

Parameters

- **`ptr`** (`void*`) – Pointer to previously allocated memory.
- **`new_size`** (`size_t`) – New size for the memory, in bytes.

2.6.2.1 Description

A size of zero will free the memory. The memory won't be modified if the allocation fails.

2.6.2.2 Return

Pointer to the reallocated memory, NULL in case of failure or if new_size is zero.

2.6.3 qfree_internal

void **`qfree_internal`**(void *ptr)

Free memory allocated with *`qmalloc()`*.

Parameters

- **`ptr`** (`void*`) – Pointer to previously allocated memory. NULL will do nothing.

2.6.4 qmalloc

void ***qmalloc**(size_t size)

Allocate memory.

Parameters

- **size** (size_t) – Number of bytes to allocate.

2.6.4.1 Description

The internal quota will be set to the infinite quota.

2.6.4.2 Return

Pointer to the allocated memory or NULL in case of failure.

2.6.5 qcalloc

void ***qcalloc**(size_t nb_items, size_t item_size)

Allocate memory and set it to 0.

Parameters

- **nb_items** (size_t) – Number of items to allocate.
- **item_size** (size_t) – Size of an item.

2.6.5.1 Description

The internal quota will be set to the infinite quota.

2.6.5.2 Return

Pointer to the allocated memory or NULL in case of failure.

2.6.6 qfree

void **qfree**(void *ptr)

Free memory allocated with *qmalloc()*.

Parameters

- **ptr** (void*) – Pointer to previously allocated memory, NULL will do nothing.

2.6.6.1 Description

The memory freed must be allocated with *qmalloc()* or *qcalloc()* (or the _quota versions) as the quota header is taken into account.

2.6.7 qrealloc

void ***qrealloc**(void *ptr, size_t new_size)

Resize the already allocated memory.

Parameters

- **ptr** (void*) – Pointer to previously allocated memory.
- **new_size** (size_t) – New size for the memory, in bytes.

2.6.7.1 Description

A size of zero will free the memory. The memory won't be modified if the allocation fails. The memory reallocated must be allocated with *qmalloc()* or *qcalloc()* (or the _quota versions) as the quota header is taken into account.

2.6.7.2 Return

Pointer to the reallocated memory, NULL in case of failure or if new_size is zero.

2.6.8 qmalloc_quota

void ***qmalloc_quota**(size_t size, enum mem_quota_id qid)

Allocate memory.

Parameters

- **size** (size_t) – Number of bytes to allocate.
- **qid** (enum mem_quota_id) – Quota to use for this allocation.

2.6.8.1 Description

The internal quota will be set to qid.

2.6.8.2 Return

Pointer to the allocated memory or NULL in case of failure.

2.6.9 qcalloc_quota

void ***qcalloc_quota**(size_t nb_items, size_t item_size, enum mem_quota_id qid)

Allocate memory and set it to 0.

Parameters

- **nb_items** (size_t) – Number of items to allocate.
- **item_size** (size_t) – Size of an item.
- **qid** (enum mem_quota_id) – Quota to use for this allocation.

2.6.9.1 Description

The internal quota will be set to qid.

2.6.9.2 Return

Pointer to the allocated memory or NULL in case of failure.

2.7 QMSG_QUEUE

2.7.1 qmsg_queue_init

struct qmsg_queue ***qmsg_queue_init**(char *msg_queue_buffer, uint32_t item_size, uint32_t max_item)

Initialize a message queue.

Parameters

- **msg_queue_buffer** (char*) – Message queue buffer.
- **item_size** (uint32_t) – Size of the items in the message queue.
- **max_item** (uint32_t) – Maximum number of items in the message queue.

2.7.1.1 NOTE

1. If msg_queue_buffer is NULL, it will be automatically allocated of (max_item * item_size). 2. Message queue thread-safety is not guaranteed and is implementation-defined.

2.7.1.2 Return

Pointer to the initialized message queue.

2.7.2 qmsg_queue_deinit

void **qmsg_queue_deinit**(struct qmsg_queue *msg_queue)

De-initialize a message queue.

Parameters

- **msg_queue** (struct qmsg_queue*) – Pointer to the message queue initialized by *qmsg_queue_init()*.

2.7.3 qmsg_queue_put

enum *qerr* **qmsg_queue_put**(struct qmsg_queue *msg_queue, const void *item)

Push an item in the message queue.

Parameters

- **msg_queue** (struct qmsg_queue*) – Pointer to the message queue initialized by *qmsg_queue_init()*.
- **item** (const void*) – Item to push into the queue.

2.7.3.1 Return

QERR_SUCCESS on success, else another enum qerr value.

2.7.4 qmsg_queue_get

enum *qerr* **qmsg_queue_get**(struct qmsg_queue *msg_queue, void *item, uint32_t timeout_ms)

Get an item from the message queue.

Parameters

- **msg_queue** (struct qmsg_queue*) – Pointer to the message queue initialized by *qmsg_queue_init()*.
- **item** (void*) – Pointer to a buffer that will get the first item of the queue or NULL if the queue is empty.
- **timeout_ms** (uint32_t) – Delay until timeout in ms. Use *QOSAL_WAIT_FOREVER* to wait indefinitely.

2.7.4.1 Return

QERR_SUCCESS or error.

2.8 QMUTEX

2.8.1 qmutex_init

struct qmutex ***qmutex_init**(void)

Initialize a mutex.

Parameters

- **void** – no arguments

2.8.1.1 Return

Pointer to the initialized mutex.

2.8.2 qmutex_deinit

void **qmutex_deinit**(struct qmutex *mutex)

De-initialize a mutex.

Parameters

- **mutex** (struct qmutex*) – Pointer to the mutex initialized by *qmutex_init()*.

2.8.3 qmutex_lock

enum *qerr* **qmutex_lock**(struct qmutex *mutex, uint32_t timeout_ms)

Lock a mutex.

Parameters

- **mutex** (struct qmutex*) – Pointer to the mutex initialized by *qmutex_init()*.
- **timeout_ms** (uint32_t) – Delay until timeout in ms. Use *QOSAL_WAIT_FOREVER* to wait indefinitely.

2.8.3.1 Return

QERR_SUCCESS or error.

2.8.4 qmutex_unlock

enum *qerr* **qmutex_unlock**(struct qmutex *mutex)

Unlock a mutex.

Parameters

- **mutex** (struct qmutex*) – Pointer to the mutex initialized by *qmutex_init()*.

2.8.4.1 Return

QERR_SUCCESS or error.

2.9 QPM

2.9.1 enum qpm_sleep_state

enum **qpm_sleep_state**
QOSAL Power sleep state, following ACPI standard.

2.9.1.1 Definition

```
enum qpm_sleep_state {
    QPM_STATE_S0,
    QPM_STATE_S0ix,
    QPM_STATE_S1,
    QPM_STATE_S2,
    QPM_STATE_S3,
    QPM_STATE_S4,
    QPM_STATE_S5
};
```

2.9.1.2 Constants

QPM_STATE_S0
ACPI Sleep State S0.

QPM_STATE_S0ix
ACPI Sleep State S0ix.

QPM_STATE_S1
ACPI Sleep State S1.

QPM_STATE_S2
ACPI Sleep State S2.

QPM_STATE_S3
ACPI Sleep State S3.

QPM_STATE_S4
ACPI Sleep State S4.

QPM_STATE_S5
ACPI Sleep State S5.

2.9.2 macro QPM_ALL_SUBSTATES

`QPM_ALL_SUBSTATES()`

Select all sub-states of a given power sleep state.

2.9.3 qpm_sleep_state_lock

void **`qpm_sleep_state_lock`**(enum *qpm_sleep_state* state, uint8_t substate_id)

Disallow a power sleep state by increasing a lock counter.

Parameters

- **`state`** (`enum` *`qpm_sleep_state`*) – Power sleep state.
- **`substate_id`** (`uint8_t`) – ID of sub-state.

2.9.3.1 Description

Lock power sleep states higher or equal the given sleep state. This means that only the sleep states below the specified one will be accessible.

For example `qpm_mgmt_state_lock(QPM_STATE_S1, QPM_ALL_SUBSTATES)` will allow the states QPM_STATE_S0 and QPM_STATE_S0ix only.

To allow again QPM_STATE_S1 and higher, the sleep state must be unlocked the same number of times it has been locked, so that the lock counter equals 0.

2.9.4 qpm_sleep_state_unlock

void **`qpm_sleep_state_unlock`**(enum *qpm_sleep_state* state, uint8_t substate_id)

Allow a power sleep state by decreasing a lock counter.

Parameters

- **`state`** (`enum` *`qpm_sleep_state`*) – Power sleep state.
- **`substate_id`** (`uint8_t`) – ID of sub-state.

2.9.5 qpm_sleep_state_is_active

bool **`qpm_sleep_state_is_active`**(enum *qpm_sleep_state* state, uint8_t substate_id)

Check if a power sleep state is active (unlocked) or not.

Parameters

- **`state`** (`enum` *`qpm_sleep_state`*) – Power sleep state.
- **`substate_id`** (`uint8_t`) – ID of sub-state.

2.9.5.1 Description

In order to be active, the lock counter of a power sleep state must equals 0.

2.9.5.2 Return

True if power sleep state is unlocked, false otherwise.

2.9.6 qpm_set_low_power_mode

enum *qerr* **`qpm_set_low_power_mode`**(bool enabled)

Set low power mode state.

Parameters

- **`enabled`** (`bool`) – Enable or disable low power mode state.

2.9.6.1 Description

With current implementation, *`qpm_set_low_power_mode()`* must be called at least once before using *`qpm_get_low_power_mode()`*. Is ensures persited low power config matches actual setting.

2.9.6.2 Return

QERR_SUCCESS or error.

2.9.7 qpm_get_low_power_mode

bool **`qpm_get_low_power_mode`**(void)

get low power mode state.

Parameters

- **`void`** – no arguments

2.9.7.1 Return

True if low power mode is set, otherwise false.

2.9.8 qpm_set_min_inactivity_s4

enum *qerr* **`qpm_set_min_inactivity_s4`**(uint32_t time_ms)

Set the minimum inactivity time to enter S4.

Parameters

- **`time_ms`** (`uint32_t`) – minimum inactivity time to get in S4, in ms.

2.9.8.1 Return

QERR_SUCCESS or error.

2.9.9 qpm_get_min_inactivity_s4

enum *qerr* **qpm_get_min_inactivity_s4**(uint32_t *time_ms)

Get the minimum inactivity time to enter S4.

Parameters

- **time_ms** (uint32_t*) – minimum inactivity time to get in S4, in ms.

2.9.9.1 Return

QERR_SUCCESS or error.

2.10 QPROFILING

2.10.1 macro QOSAL_THREAD_MAX_NAME_LEN

QOSAL_THREAD_MAX_NAME_LEN()

Maximum length of thread name. For debug and logging purposes.

2.10.2 struct qmemstats

struct **qmemstats**

Global memory statistics.

2.10.2.1 Definition

```
struct qmemstats {
    int32_t static_size;
    int32_t heap_used;
    int32_t heap_peak;
    int32_t heap_size;
}
```

2.10.2.2 Members

static_size
Size of statically allocated memory.

heap_used
Current amount of dynamically allocated memory.

heap_peak
Peak amount of dynamically allocated memory.

heap_size
Size of heap for dynamic memory allocation.

2.10.2.3 Description

A negative value indicates that the data could not be retrieved.

2.10.3 struct qstackstats

struct **qstackstats**

Per stack memory statistics.

2.10.3.1 Definition

```
struct qstackstats {
#if QOSAL_THREAD_MAX_NAME_LEN > 0
    char thread_name[QOSAL_THREAD_MAX_NAME_LEN];
#endif
    int32_t stack_used;
    int32_t stack_peak;
    int32_t stack_size;
}
```

2.10.3.2 Members

thread_name
Name of the thread using the stack (if available).

stack_used
Current stack usage.

stack_peak
Peak stack usage.

stack_size
Size of stack.

2.10.3.3 Description

A negative value indicates that the data could not be retrieved.

2.10.4 qmemstat_get

void **qmemstat_get**(struct *qmemstats* *stats)

Get global memory statistics.

Parameters

- **stats** (struct *qmemstats**) – Pointer to the struct to fill with memory statistics.

2.10.5 qmemstat

void **qmemstat**(void)

Display peak memory allocated with *qmalloc()*.

Parameters

- **void** – no arguments

2.10.6 qstackstat_count_get

int **qstackstat_count_get**(void)

Get number of stacks.

Parameters

- **void** – no arguments

2.10.6.1 Return

Number of stacks handled by the OS.

2.10.7 qstackstat_get

int **qstackstat_get**(struct *qstackstats* *stats, int stack_count)

Get per stack memory statistics.

Parameters

- **stats** (struct *qstackstats**) – Pointer to the array of structs to fill with memory statistics.
- **stack_count** (int) – number of allocated structs in the array.

2.10.7.1 Return

Number of structs actually filled by the function.

2.10.8 qstackstat

void **qstackstat**(void)

Display peak stack usage per thread.

Parameters

- **void** – no arguments

2.10.9 qprofstat

void **qprofstat**(void)

Call others qprofiling functions. Include *qmemstat()*, *qstackstat()*.

Parameters

- **void** – no arguments

2.11 QRAND

2.11.1 qrand_seed

void **qrand_seed**(uint32_t seed)

Initialize the seed for rand generator.

Parameters

- **seed** (uint32_t) – Initialization value.

2.11.2 qrand_rand

uint32_t **qrand_rand**(void)

Return a random number.

Parameters

- **void** – no arguments

2.11.2.1 Return

random. Max value is 65536.

2.12 QSEMAPHORE

2.12.1 qsemaphore_init

struct qsemaphore ***qsemaphore_init**(uint32_t init_count, uint32_t max_count)

Initialize a semaphore.

Parameters

- **init_count** (uint32_t) – Initial semaphore count.
- **max_count** (uint32_t) – Maximum semaphore count.

2.12.1.1 Return

Pointer to the initialized semaphore on NULL on error.

2.12.2 qsemaphore_deinit

void **qsemaphore_deinit**(struct qsemaphore *sem)

De-initialize a semaphore.

Parameters

- **sem** (struct qsemaphore*) – Pointer to the semaphore initialized by *qsemaphore_init()*.

2.12.3 qsemaphore_take

enum *qerr* **qsemaphore_take**(struct qsemaphore *sem, uint32_t timeout_ms)

Take a semaphore.

Parameters

- **sem** (struct qsemaphore*) – Pointer to the semaphore initialized by *qsemaphore_init()*.
- **timeout_ms** (uint32_t) – Delay until timeout in ms. Use *QOSAL_WAIT_FOREVER* to wait indefinitely.

2.12.3.1 Return

QERR_SUCCESS or error.

2.12.4 qsemaphore_give

enum *qerr* **qsemaphore_give**(struct qsemaphore *sem)

Give a semaphore.

Parameters

- **sem** (struct qsemaphore*) – Pointer to the semaphore initialized by *qsemaphore_init()*.

2.12.4.1 Return

QERR_SUCCESS or error.

2.13 QSIGNAL

2.13.1 qsignal_init

struct qsignal ***qsignal_init**(void)

Initialize a signal.

Parameters

- **void** – no arguments

2.13.1.1 NOTE

Signal thread-safety is not guaranteed and is implementation-defined.

2.13.1.2 Return

Pointer to the initialized signal or NULL on error.

2.13.2 qsignal_deinit

void **qsignal_deinit**(struct qsignal *signal)

De-initialize a signal.

Parameters

- **signal** (struct qsignal*) – Pointer to the signal initialized by *qsignal_init()*.

2.13.3 qsignal_raise

enum *qerr* **qsignal_raise**(struct qsignal *signal, int value)

Raise a signal.

Parameters

- **signal** (struct qsignal*) – Pointer to the signal initialized by *qsignal_init()*.
- **value** (int) – The value sent by the signal.

2.13.3.1 Return

QERR_SUCCESS or error.

2.13.4 qsignal_wait

enum *qerr* **qsignal_wait**(struct qsignal *signal, int *value, uint32_t timeout_ms)

Wait for a signal.

Parameters

- **signal** (struct qsignal*) – Pointer to the signal initialized by *qsignal_init()*.
- **value** (int*) – Pointer that will be filled with the value of the signal.
- **timeout_ms** (uint32_t) – Delay until timeout in ms.

2.13.4.1 Return

QERR_SUCCESS or error.

2.14 QTHREAD

2.14.1 macro QTHREAD_STACK_DEFINE

`QTHREAD_STACK_DEFINE`(name, stack_size)

Statically allocate a thread stack.

Parameters

- **`name`** – Name of the stack
- **`stack_size`** – Stack size

2.14.2 macro QALIGN

`QALIGN`(size, byte)

Perform a byte alignment.

Parameters

- **`size`** – Initial size.
- **`byte`** – Number of byte boundary for the alignment.

2.14.3 enum qthread_priority

enum **`qthread_priority`**

QOSAL Thread priority.

2.14.3.1 Definition

```
enum qthread_priority {
    QTHREAD_PRIORITY_CRITICAL,
    QTHREAD_PRIORITY_HIGH,
    QTHREAD_PRIORITY_ABOVE_NORMAL,
    QTHREAD_PRIORITY_NORMAL,
    QTHREAD_PRIORITY_BELOW_NORMAL,
    QTHREAD_PRIORITY_LOW,
    QTHREAD_PRIORITY_IDLE,
    QTHREAD_PRIORITY_MAX
};
```

2.14.3.2 Constants

QTHREAD_PRIORITY_CRITICAL
Critical priority (maximum).

QTHREAD_PRIORITY_HIGH
High priority.

QTHREAD_PRIORITY_ABOVE_NORMAL
Above normal priority.

QTHREAD_PRIORITY_NORMAL
Normal priority.

QTHREAD_PRIORITY_BELOW_NORMAL
Below normal priority.

QTHREAD_PRIORITY_LOW
Low priority.

QTHREAD_PRIORITY_IDLE
Idle priority (minimum).

QTHREAD_PRIORITY_MAX
Internal use.

2.14.4 typedef qthread_func

void **qthread_func**(void *arg)

Pointer to a thread entry point.

Parameters

- **arg** (void*) – private data of the thread.

2.14.4.1 Return

nothing.

2.14.5 qthread_create

struct qthread ***qthread_create**(*qthread_func* thread, void *arg, const char *name, void *stack, uint32_t stack_size, enum *qthread_priority* prio)

Create a new thread.

Parameters

- **thread** (*qthread_func*) – Entry point of the thread.
- **arg** (void*) – Private data of the thread.
- **name** (const char*) – Name of the thread.
- **stack** (void*) – Pointer to the stack of the thread.
- **stack_size** (uint32_t) – Size of the stack of the thread.
- **prio** (enum *qthread_priority*) – Priority of the thread.

2.14.5.1 NOTE

If stack is NULL, it will be automatically allocated.

2.14.5.2 Return

Pointer to the created thread.

2.14.6 qthread_join

enum *qerr* **qthread_join**(struct qthread *thread)

Wait for the thread to exit.

Parameters

- **thread** (`struct qthread*`) – Pointer to the thread initialized by *qthread_create()*.

2.14.6.1 Return

QERR_SUCCESS or error.

2.14.7 qthread_delete

enum *qerr* **qthread_delete**(struct qthread *thread)

Delete a running thread.

Parameters

- **thread** (`struct qthread*`) – Pointer to the thread initialized by *qthread_create()*.

2.14.7.1 Return

QERR_SUCCESS or error.

2.14.8 qthread_yield

enum *qerr* **qthread_yield**(void)

Yield the current thread to allow other threads to run.

Parameters

- **void** – no arguments

2.14.8.1 Return

QERR_SUCCESS or error.

2.15 QTIME

2.15.1 macro QOSAL_WAIT_FOREVER

`QOSAL_WAIT_FOREVER()`

Timeout value to wait forever.

2.15.2 qtime_get_string_ticks_per_s_default

const char ***qtime_get_string_ticks_per_s_default**(void)

Get tick per second (Hz) in string.

Parameters

- **void** – no arguments

2.15.2.1 Return

ticks per s in string.

2.15.3 qtime_get_uptime_ticks_default

int64_t **qtime_get_uptime_ticks_default**(void)

Get uptime in ticks.

Parameters

- **void** – no arguments

2.15.3.1 Return

uptime in ticks.

2.15.4 qtime_get_uptime_us

int64_t **qtime_get_uptime_us**(void)

Get uptime in us.

Parameters

- **void** – no arguments

2.15.4.1 Return

uptime in us.

2.15.5 qtime_msleep

void **qtime_msleep**(int ms)

Sleep milliseconds.

Parameters

- **ms** (int) – Number of ms to sleep.

2.15.6 qtime_usleep

void **qtime_usleep**(int us)

Sleep microseconds.

Parameters

- **us** (int) – Number of us to sleep.

2.15.7 qtime_msleep_yield

void **qtime_msleep_yield**(int ms)

Sleep milliseconds with yielding.

Parameters

- **ms** (int) – Number of ms to sleep.

2.15.8 qtime_usleep_yield

void **qtime_usleep_yield**(int us)

Sleep microseconds with yielding.

Parameters

- **us** (int) – Number of us to sleep.

2.15.9 qtime_get_sys_freq_hz

uint32_t **qtime_get_sys_freq_hz**(void)

Get system frequency in Hz.

Parameters

- **void** – no arguments

2.15.9.1 Return

system frequency in Hz.

2.16 QTOOLCHAIN

2.16.1 macro QFFS

`QFFS(x)`

See __builtin_ffs in GCC documentation.

Parameters

- **x** – The value to test.

2.16.1.1 Return

one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

2.16.2 macro LIKELY

`LIKELY(x)`

See __builtin_expect in GCC documentation. Help the compiler to understand the condition is likely to be true to optimize the branch.

Parameters

- **x** – The value to test.

2.16.2.1 Return

0 if x is false, 1 if x is true.

2.16.3 macro UNLIKELY

`UNLIKELY(x)`

See __builtin_expect in GCC documentation. Help the compiler to understand the condition is unlikely to be true to optimize the branch.

Parameters

- **x** – The value to test.

2.16.3.1 Return

0 if x is false, 1 if x is true.

2.16.4 macro typeof

`typeof``(x)`

See __builtin_types_compatible_p in GCC documentation.

Parameters

- **`x`** – The value to test.

2.16.4.1 Return

the type of x.

2.17 QTRACING

2.17.1 typedef qtracing_cb_t

void **`qtracing_cb_t`**(const char *fmt, ...)

Define a tracing callback.

Parameters

- **`fmt`** (`const char*`) – string format of the trace.
- **`ellipsis`** (`ellipsis`) – variable list of arguments.

2.17.1.1 Return

nothing.

2.17.2 qtracing_init

enum *qerr* **`qtracing_init`**(void)

Initialize tracing.

Parameters

- **`void`** – no arguments

2.17.2.1 Return

QERR_SUCCESS or error.

2.18 QWORKQUEUE

2.18.1 typedef qwork_func

void **qwork_func**(void *arg)

Pointer to a work task entry point.

Parameters

- **arg** (void*) – private data of the workqueue.

2.18.1.1 Return

nothing.

2.18.2 qworkqueue_init

struct qworkqueue ***qworkqueue_init**(*qwork_func* handler, void *priv)

Initialize a workqueue.

Parameters

- **handler** (*qwork_func*) – Entry point of the work task.
- **priv** (void*) – Private data of the work task.

2.18.2.1 Return

Pointer to the initialized workqueue.

2.18.3 qworkqueue_schedule_work

enum *qerr* **qworkqueue_schedule_work**(struct qworkqueue *workqueue)

Schedule work task in a workqueue.

Parameters

- **workqueue** (struct qworkqueue*) – Pointer to the workqueue initialized by *qworkqueue_init()*.

2.18.3.1 Return

QERR_SUCCESS or error.

2.18.4 qworkqueue_cancel_work

enum *qerr* **qworkqueue_cancel_work**(struct qworkqueue *workqueue)

Cancel work task in a workqueue and free the queue.

Parameters

- **workqueue** (struct qworkqueue*) – Pointer to the workqueue initialized by *qworkqueue_init()*.

2.18.4.1 Return

QERR_SUCCESS or error.

Index