QRT User Manual

Release v1.2.1

Preempt Systems Oy

Oct 10, 2026

CONTENTS:

1 Part I: Getting Started **3**
- 1.1 Prerequisites . . . 3
- 1.2 Document Conventions . . . 3
- 1.3 Supported Architectures . . . 3
- 1.4 Supported Toolchains . . . 4
- 1.5 Building the Kernel . . . 4

2 Part II: Kernel Architecture **5**
- 2.1 Architecture Overview . . . 5
- 2.2 Boot & Kernel Bring-Up . . . 6
- 2.3 Timers . . . 6
- 2.4 Scheduler . . . 8
- 2.5 Floating-Point Support . . . 9
- 2.6 Console . . . 10

3 Part III: Kernel Services **13**
- 3.1 POSIX Thread . . . 13
- 3.2 POSIX Semaphore . . . 15
- 3.3 POSIX Thread Mutex . . . 15
- 3.4 POSIX Message Queue . . . 16
- 3.5 POSIX Timer . . . 16
- 3.6 POSIX Clock . . . 17
- 3.7 System Call Extension . . . 17
- 3.8 Deferred Call . . . 17

4 Part IV: Memory Protection **19**
- 4.1 Privileged and Unprivileged Modes . . . 19
- 4.2 Stack Isolation . . . 19
- 4.3 Memory Regions . . . 19
- 4.4 Memory Fault Reporting . . . 20
- 4.5 Configuration . . . 20

5 Part V: Kernel Configuration **23**
- 5.1 Configuration Options . . . 23

6 Part VI: Memory Footprint **31**
- 6.1 How the figures are measured . . . 31
- 6.2 Minimal configuration . . . 31
- 6.3 Per-feature ROM cost . . . 31
- 6.4 Kernel object sizes . . . 32
- 6.5 Thread Control Block . . . 32

7 Part VII: Profiling **35**
7.1 Enabling instrumentation . . . 35
7.2 Reading the results . . . 35
7.3 Worst-case figures . . . 36

8 Part VIII: API Reference **37**
8.1 POSIX . . . 37
8.2 QRT . . . 105

9 Changelog **117**
9.1 Version History . . . 117

10 Glossary **123**
10.1 A . . . 123
10.2 B . . . 123
10.3 C . . . 123
10.4 D . . . 124
10.5 F . . . 124
10.6 H . . . 124
10.7 I . . . 124
10.8 J . . . 125
10.9 K . . . 125
10.10 L . . . 125
10.11 M . . . 125
10.12 N . . . 126
10.13 O . . . 126
10.14 P . . . 126
10.15 R . . . 127
10.16 S . . . 127
10.17 T . . . 128
10.18 U . . . 128
10.19 W . . . 129
10.20 Y . . . 129

Index **131**

QRT Kernel Documentation

Copyright © 2026 Preempt Systems Oy. All rights reserved.

CHAPTER

ONE

PART I: GETTING STARTED

QRT is an advanced preemptive, priority-based Real-Time Kernel designed to address the demanding needs of modern embedded applications requiring precise timing, deterministic execution, and efficient resource management. Engineered specifically for embedded platforms, QRT provides robust performance suitable for a wide variety of real-time use cases.

This manual provides technical documentation for QRT. It covers the kernel architecture, scheduling mechanisms, memory protection, API usage, and integration requirements. This document is intended for embedded software developers and system integrators using QRT in microcontroller-based systems.

1.1 Prerequisites

The reader is expected to have:

- A working knowledge of the target processor architecture (e.g., ARM Cortex-M).
- Familiarity with C programming in embedded environments.
- Experience using build tools, toolchains, and debugging software relevant to the target platform.

1.2 Document Conventions

- `Monospaced` — API calls, macros, configuration defines.
- **Bold** — Important terms or emphasis.

1.3 Supported Architectures

QRT currently supports the **ARM Cortex-M** family of microcontrollers:

- ARMv6-M (Cortex-M0+)
- ARMv7-M (Cortex-M3, M4, M7)
- ARMv8-M Baseline (Cortex-M23)
- ARMv8-M Mainline (Cortex-M33, M35P, M52, M55, M85)

All targets are supported in little-endian mode. FPU and TrustZone extensions are supported where available.

1.4 Supported Toolchains

QRT is officially supported with the **Arm GNU Toolchain** (ARM GCC 13.x or later).

Other compilers may work if they follow the ARM EABI specification, but they are not verified or maintained.

1.5 Building the Kernel

1.5.1 Build Configuration

QRT is configured using a set of compile-time configuration options defined in the header file `qrt_cfg.h`.

Default configuration options are tuned for ARM Cortex-M3.

1.5.2 Build Requirements

QRT itself is build-system agnostic.

However, to build the official demo application distributed with QRT requires the following tools:

- Arm GNU Toolchain (Arm GCC 13.x or newer)
- CMake 3.20 or newer
- Ninja build system

1.5.3 Building the Demo Application

To build the demo navigate to the root directory of the QRT source tree and run the following commands:

```
cmake -B build -G Ninja -DCMAKE_TOOLCHAIN_FILE=arm-none-eabi-gcc.cmake
cmake --build build
```

This produces the demo binary and the static library `libqrt.a`.

CHAPTER

TWO

PART II: KERNEL ARCHITECTURE

2.1 Architecture Overview

QRT is POSIX native Real-Time Kernel which implements a subset of the POSIX.1-2024 standard. In addition, QRT incorporates advanced real-time features such as High-Resolution Scheduling and Memory Protection.

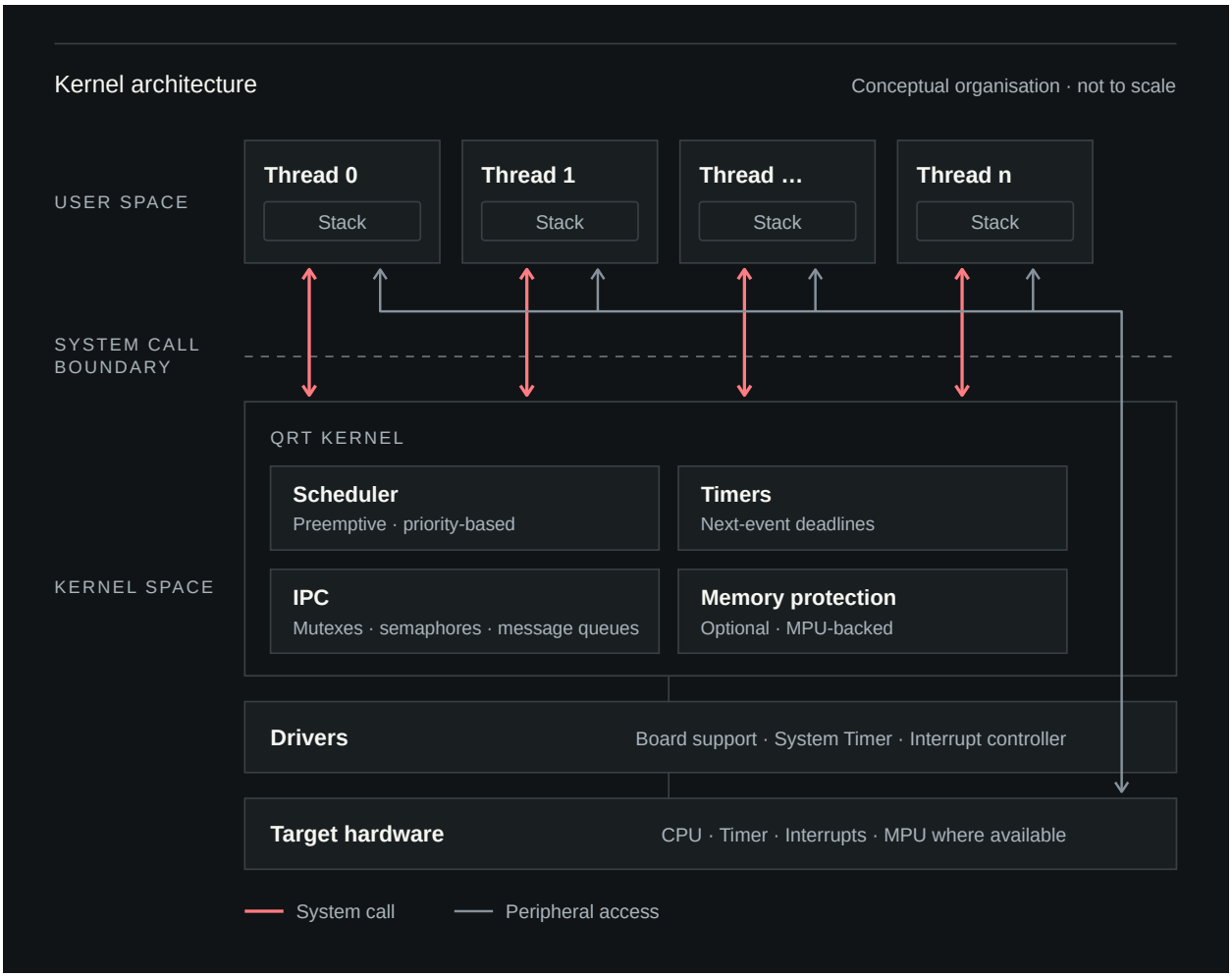


Fig. 1: Kernel Architecture diagram.

2.1.1 Execution Model

QRT operates in two execution levels: Privileged and Unprivileged. Privileged code has unrestricted access to all system resources, while unprivileged code is restricted from modifying critical kernel or hardware registers.

When Memory Protection is disabled, all code executes in privileged mode and can access the entire memory map.

When Memory Protection is enabled, QRT runs the kernel, idle thread, and all interrupt service routines in privileged mode. User threads may execute in unprivileged mode, isolating them from kernel memory, other thread stacks, and preventing direct access to protected peripherals. User threads may also run in privileged mode if required.

2.1.2 Memory Model

Each thread and the kernel has its own stack memory, isolated from each other.

ARM Cortex-M processors provide two stack pointers, the Process Stack Pointer (PSP) and the Main Stack Pointer (MSP), which the kernel uses to keep thread stacks separate from the stack used by the kernel and interrupt handlers.

2.2 Boot & Kernel Bring-Up

QRT is shipped with default BSP for ARM Cortex-M platforms which handles necessary low-level initialization for the kernel operation. BSP must be initialized with `bsp_init()`.

Before booting the kernel, the System Timer must be configured by calling *qrt_systimer_configure()*.

The kernel is initialized and started by calling *qrt_kernel_boot()*, which takes thread attributes for the startup thread as parameters. After the kernel has started, the scheduler begins managing thread execution and the startup thread begins running. More threads can be created after the kernel has started.

The kernel handles Memory Protection Unit (MPU) configuration if Memory Protection is enabled (*QRT_CFG_MPU_ENABLE*).

2.3 Timers

2.3.1 System Timer

The kernel requires a hardware timer to serve as the primary timekeeping mechanism. The System Timer provides time-based services, ensuring accurate delays, timeouts, and scheduling. The resolution of timing services is set by the frequency of the hardware System Timer.

The System Timer interrupt is configured with the highest priority to maintain accurate timing and ensure no ticks are lost during interrupt handling. To satisfy this requirement, the System Timer interrupt must be the only interrupt assigned the highest priority level.

Configuration

The kernel requires integration with a hardware timer. The user must configure the hardware timer to trigger interrupts on overflow and provide *qrt_systimer_configure()* with the following parameters:

- A function for reloading the hardware timer
- A function for reading the hardware timer
- The maximum period of the hardware timer in ticks
- The frequency of the hardware timer in Hertz (Hz)
- The hardware timer's interrupt request number

The hardware timer's interrupt handler must invoke `qrt_systimer_handler()`. The kernel manages the interrupt controller configuration for the hardware timer.

2.3.2 High-Resolution Scheduling

The kernel implements High-Resolution Scheduling: it has no periodic tick. The System Timer is programmed for the next sleep, timeout, timer expiry or round-robin time slice end that falls due, and interrupts when that event is due rather than at a fixed rate. If nothing falls due within the timer's maximum period, it interrupts at the end of that period so that the kernel can keep system time. Timed services therefore resolve to one count of the hardware System Timer instead of a tick period, and there is no tick rate to configure. `clock_getres()` reports this resolution in whole nanoseconds.

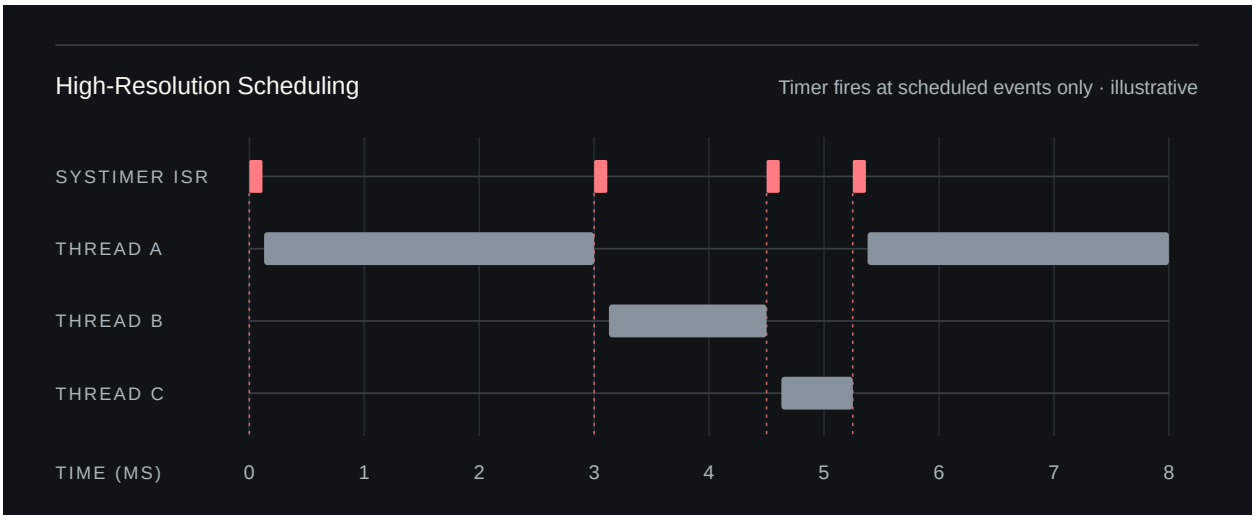


Fig. 2: High-Resolution Scheduling timing diagram.

Comparison with Tick-Resolution Scheduling

A kernel that uses Tick-Resolution Scheduling interrupts at a fixed tick rate, typically every millisecond, whether or not an event is due. Sleeps, timeouts and time slices resolve to whole tick periods, so the tick rate trades timing resolution against interrupt overhead. High-Resolution Scheduling removes that trade-off.

	Tick-Resolution Scheduling	High-Resolution Scheduling
Timer interrupts	Every tick, whether or not an event is due	When a sleep, timeout, timer or time slice falls due; otherwise only for kernel timekeeping, at the latest when the System Timer reaches its maximum period
Resolution of sleeps, timeouts and timers	One tick period	One System Timer count
Time slice length	A whole number of tick periods	Any number of System Timer counts
Tick rate	A compromise between resolution and overhead	None to configure

System Time in Ticks

For compatibility with code written for tick-based kernels, QRT provides an API for querying the system time in ticks. The system time can be retrieved with `qrt_systimer_getmonotonic()`, which returns `EINVAL` until `qrt_kernel_boot()` has started the kernel. The value it reports is the elapsed system time expressed in ticks of the hardware System Timer, at its configured frequency, not in periods of a fixed-rate tick interrupt.

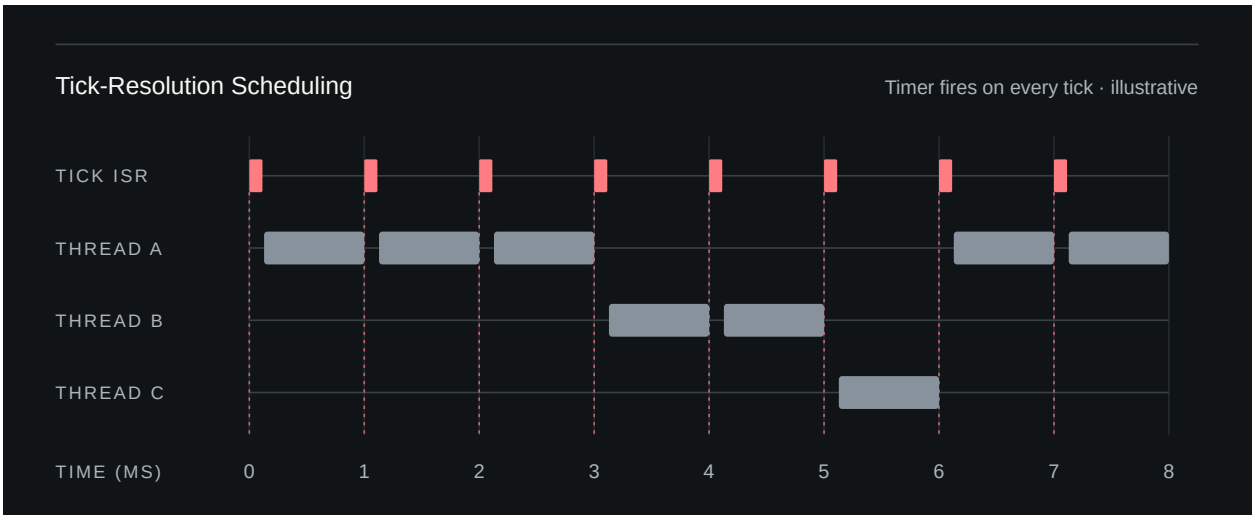


Fig. 3: Tick-Resolution Scheduling timing diagram.

2.4 Scheduler

The scheduler manages thread execution, ensuring fair resource distribution, responsiveness, and prioritized handling of critical threads. The kernel uses preemptive scheduling, allowing it to interrupt a running thread whenever a higher priority thread becomes ready to execute.

2.4.1 Priority-Based Scheduling

The kernel uses preemptive priority-based scheduling, a scheduling algorithm designed to allocate processing time among threads based on their priority level, where a higher value represents higher priority.

To enable Priority-Based scheduling for a thread, pthread attribute must be set with *SCHED_FIFO* with *pthread_attr_setschedpolicy()*.

A thread preempted by a higher-priority thread resumes before the other ready threads of its priority. A thread whose priority is changed with *pthread_setschedparam()* goes behind the ready threads of its new priority. *pthread_setschedprio()* keeps the thread's place when the priority is unchanged, places a raised thread behind the ready threads of its new priority and a lowered thread ahead of them.

2.4.2 Round-Robin Scheduling

The kernel provides Round-Robin scheduling for threads of equal priority, ensuring fair distribution of CPU time among threads without strict real-time constraints, such as background tasks. Threads execute in rotation, each receiving a fixed execution time slice. When its time slice expires, the thread goes behind the other ready threads of its priority, and the next of them executes. A thread preempted by a higher-priority thread resumes first and runs the rest of its time slice.

To enable Round-Robin scheduling for a thread, pthread attribute must be set with *SCHED_RR* with *pthread_attr_setschedpolicy()*.

Round-Robin scheduling is a superset of Priority-Based scheduling, meaning that Round-Robin scheduling also considers thread priority levels. Round-Robin scheduling is the default scheduling policy.

2.4.3 Interrupts and Scheduling

Interrupt Service Routines (ISRs) respond to hardware events by temporarily interrupting thread execution.

QRT does not perform scheduling directly within ISR context. Instead, QRT provides deferred system calls *qrt_defer_call()* for ISRs to request kernel services after the ISR has completed. This approach minimizes interrupt latency and ensures predictable execution.

2.4.4 Yielding

Threads do not need to explicitly yield, but the scheduler automatically preempts lower-priority threads and manages execution order based on priority and scheduling policy. Threads can voluntarily yield the processor using *sched_yield()*, allowing other threads of equal priority to execute. Yielding is particularly useful when a thread has completed its immediate work and wants to allow other threads to run.

2.4.5 Waiting Threads

Threads waiting on a semaphore, mutex or message queue are woken highest priority first, and in the order they started waiting among threads of equal priority. A waiting thread whose priority changes, through *pthread_setschedparam()*, *pthread_setschedprio()* or priority inheritance, is placed after the threads already waiting at its new priority, also when *pthread_setschedprio()* lowers it; only a ready thread goes ahead of its peers then.

2.4.6 Priority Inversion

Priority inversion occurs when a lower-priority thread indirectly prevents a higher-priority thread from executing, effectively reversing the intended priority order. Priority inversion can be bounded or unbounded:

- **Bounded priority inversion** occurs when a high-priority thread is waiting for a resource held by a low-priority thread. The duration of inversion is bounded because it ends as soon as the low-priority thread releases the resource.
- **Unbounded priority inversion** occurs if the duration of inversion becomes unpredictable due to intermediate-priority threads preempting the low-priority thread holding a critical resource needed by a high-priority thread.

QRT mitigates priority inversion through Priority Inheritance (PI) and Priority Ceiling (PC) protocols when using mutexes. See *POSIX Thread Mutex* for details on configuring these protocols.

2.4.7 Idle Thread

The kernel creates a idle thread that runs when no other threads are in the ready state. The idle thread always runs at the lowest priority level and System Calls cannot be invoked from it. Idle thread executes in a minimal infinite loop unless power-saving mode is enabled.

If power-saving mode is enabled, the idle thread places the processor into a low-power state when no other threads are ready. Power-saving mode can be enabled with configuration option *QRT_CFG_IDLE_DEEP_SLEEP_ENABLE*.

An idle thread hook can be registered and unregistered with *qrt_kernel_idlehookregister()* to execute custom code during idle periods. Care must be taken to ensure the hook function does not compromise the system security as it executes in kernel context.

2.5 Floating-Point Support

Some supported processor cores include a FPU (Floating-Point Unit), introducing additional registers that must be managed during context switching. Stacking floating-point registers increases thread stack size, adds interrupt latency, and extends context switching time because the registers must be saved and restored.

The kernel supports stacking floating-point registers and it can be enabled per thread with a call to *pthread_setfpcontext_np()* passing *PTHREAD_FPCONTEXT_ENABLE_NP*, and can be disabled by passing *PTHREAD_FPCONTEXT_DISABLE_NP*.

FPU support can be enabled with configuration option *QRT_CFG_FPU_ENABLE*, and floating-point stacking can be enabled with *QRT_CFG_FPU_STACK_ENABLE*.

2.6 Console

QRT ships a small, freestanding (libc-free) `printf` implementation used for boot and diagnostic output. Formatted text is buffered internally and emitted as whole strings through `qrt_console_puts()`, a weak hook that the platform overrides to target its console (UART, semihosting, ITM).

2.6.1 Configuration

Console output is gated by the *QRT_CFG_CONSOLE_ENABLE* option in `qrt_cfg.h`.

2.6.2 Output hook and linkage

`qrt_console_puts()` is defined **weakly** inside the kernel as a no-op, so the kernel links standalone with no console dependency. The platform supplies a strong definition with external linkage that routes characters to the actual output device:

```
/* Platform-side strong override. */
void qrt_console_puts(const char *str)
{
   while (*str != '\0')
   {
      uart_putc(*str);
      str++;
   }
}
```

2.6.3 Example boot output

With *QRT_CFG_CONSOLE_ENABLE* set to 1, the kernel prints a banner followed by a trace of each boot stage:

```
  ___  ____ _____
 / _ \|  _ \_   _|
| | | | |_) || |
| |_| |  _ < | |
 \__\_\_| \_\|_|

    v1.2.1 (yyyy-mm-dd)
    Copyright (c) Preempt Systems Oy

   Booting...
   Initializing CPU core
   Initializing heap allocators
     Kernel heap: 32768 B
     Thread stack pool: 65536 B
   Initializing software timer system
   Initializing threading subsystem
```

(continues on next page)

(continued from previous page)

```
    Initializing scheduler
    Creating idle thread
    Creating startup thread
  Configuring thread stack
  Initializing Thread-Local Storage
  Starting System Timer
    Programming initial timer period of 48000000

  Boot sequence complete
  Dispatching startup thread at 0x000001f4
```

CHAPTER

THREE

PART III: KERNEL SERVICES

3.1 POSIX Thread

POSIX Threads (*pthread_t*) represent a schedulable execution unit. Each thread executes independently with its own stack and priority, managed by the scheduler. Threads can be used by including `pthread.h`.

Threads are always enabled and the kernel must be configured with configuration option *QRT_CFG_THREAD_LIMIT* with non-zero value.

3.1.1 Thread Attributes

Threads are created with the default attributes if no attributes are specified. The default attributes are defined as:

- Detach state: *PTHREAD_CREATE_JOINABLE*
- Cancellation state: *PTHREAD_CANCEL_ENABLE*
- Cancellation type: *PTHREAD_CANCEL_DEFERRED*
- Stack size: *QRT_CFG_THREAD_STACK_DEFAULT*
- Scheduling policy: *SCHED_RR*
- Scheduling priority: lowest priority (1)
- Timeslice: *QRT_CFG_SCHED_TIME_SLICE*
- Privilege level: *PTHREAD_UNPRIVILEGED_NP*

To customize thread attributes before passing it to *pthread_create()* and a *pthread_attr_t* object must be initialized with *pthread_attr_init()*. The *pthread_attr_t* object can then be modified with:

- Stack: *pthread_attr_setstack()* / *pthread_attr_getstack()*
- Stack size (if stack not set): *pthread_attr_setstacksize()* / *pthread_attr_getstacksize()*
- Scheduling policy: *pthread_attr_setschedpolicy()* / *pthread_attr_getschedpolicy()*
- Scheduling priority: *pthread_attr_setschedparam()* / *pthread_attr_getschedparam()*
- Detach state: *pthread_attr_setdetachstate()* / *pthread_attr_getdetachstate()*
- Scheduling attribute inheritance: *pthread_attr_setinheritsched()* / *pthread_attr_getinheritsched()*
- Uninitialize the attribute object: *pthread_attr_destroy()*

QRT also provides extensions to configure aditional thread attributes:

- Timeslice: *pthread_attr_settimeslice_np()* / *pthread_attr_gettimeslice_np()*
- Privilege level: *pthread_attr_setprivilege_np()* / *pthread_attr_getprivilege_np()*

- Thread name: *pthread_attr_setname_np()* / *pthread_attr_getname_np()*

3.1.2 Thread Stack Allocation

Threads require a dedicated stack for execution. The stack can be allocated either:

- **Statically** with *pthread_attr_setstack()*
- **Dynamically** with *pthread_attr_setstacksize()*.

If the stack size is not explicitly set, a default size, which must be configured with *QRT_CFG_THREAD_STACK_DEFAULT*, is used. If dynamic memory allocation is used, thread stack pool must be configured with sufficient memory with *QRT_CFG_STACK_POOL_SIZE*.

Note

- ARMv6-M and ARMv7-M Stack base address must be stack size aligned.
- ARMv8-M Stack base address must be 32-byte aligned if memory protection is enabled.

Note

- ARMv6-M and ARMv7-M Stack size must be power of two if memory protection is enabled.
- ARMv8-M Stack size must be multiple of 32 bytes if memory protection is enabled.

3.1.3 Thread Lifecycle

Threads are created using *pthread_create()* and are exited with *pthread_exit()*.

Cancelling a thread with *pthread_cancel()* allows it to be terminated from another thread. Both deferred and asynchronous cancellation are supported:

- Deferred cancellation allows a thread to specify cancellation points where it can be safely cancelled. Cancel points include blocking system calls and explicit cancellation points with *pthread_testcancel()*.
- Asynchronous cancellation cancels a thread immediately.

Threads can be joined with (*pthread_join()*) allowing another thread to wait for its termination and retrieve its exit status. Joined threads resources are automatically released upon termination. Only deferred cancelability type threads can be joined.

Detaching a thread with *pthread_detach()* allows it to run independently. Detached threads resources are automatically released upon termination.

Cleanup handlers are used to ensure resources are released when a thread is cancelled or exits. Cleanup handlers can be registered with *pthread_cleanup_push()* and *pthread_cleanup_pop()*. *QRT_CFG_CLEANUP_LIMIT* is used to configure the number of cleanup handlers per thread.

3.1.4 Thread-Local Storage

POSIX Thread-Local Storage (TLS) provides a mechanism for threads to maintain their own private data.

TLS keys can be created using *pthread_key_create()* and deleted with *pthread_key_delete()*. Threads can set and get their own specific data using *pthread_setspecific()* and *pthread_getspecific()*.

3.2 POSIX Semaphore

POSIX semaphores provide a mechanism for signaling between threads and managing access to shared resources. Semaphores can be used by including `semaphore.h`.

Semaphores can be created using *sem_init()* and destroyed with *sem_destroy()*. Threads can wait on a semaphore using *sem_trywait()* / *sem_wait()* / *sem_timedwait()* / *sem_clockwait()* and signal a semaphore using *sem_post()*. The current value of a semaphore can be queried with *sem_getvalue()*

Part of the semaphore API is available from deferred call context.

Note

Named semaphores are not supported.

Semaphores can be enabled by setting configuration option *QRT_CFG_SEMAPHORE_ENABLE*.

3.3 POSIX Thread Mutex

POSIX Thread Mutexes provide a mechanism for mutual exclusion, allowing threads to safely access shared resources without data corruption. Mutexes can be used by including `pthread.h`.

Mutexes can be created using *pthread_mutex_init()* and destroyed with *pthread_mutex_destroy()*.

Mutexes are initialized with default attributes if no attributes are specified. The default attributes are defined as:

- Protocol: *PTHREAD_PRIO_NONE*
- Priority ceiling: 0
- Type: *PTHREAD_MUTEX_DEFAULT*

To customize mutex attributes before passing it to *pthread_mutex_init()* and a *pthread_mutexattr_t* object must be initialized with *pthread_mutexattr_init()*. The *pthread_mutexattr_t* object can then be modified with:

- Protocol: *pthread_mutexattr_setprotocol()* / *pthread_mutexattr_getprotocol()*
- Priority ceiling: *pthread_mutexattr_setprioceiling()* / *pthread_mutexattr_getprioceiling()*
- Type: *pthread_mutexattr_settype()* / *pthread_mutexattr_gettype()*, with one of the following values: - *PTHREAD_MUTEX_NORMAL* - *PTHREAD_MUTEX_ERRORCHECK* - *PTHREAD_MUTEX_RECURSIVE* (recursion depth of 255) - *PTHREAD_MUTEX_DEFAULT*
- Uninitialize the attribute object: *pthread_mutexattr_destroy()*

Threads can lock a mutex using *pthread_mutex_trylock()* / *pthread_mutex_lock()* / *pthread_mutex_timedlock()* / *pthread_mutex_clocklock()* and unlock a mutex using *pthread_mutex_unlock()*.

Mutexes can be enabled by setting configuration option *QRT_CFG_MUTEX_ENABLE*. A thread can hold any number of mutexes simultaneously.

3.3.1 Priority Inversion

POSIX Thread Mutexes support priority inheritance and priority ceiling protocols to mitigate priority inversion issues. These protocols can be configured on mutex creation using *pthread_mutexattr_setprotocol()* with one of the following values:

- *PTHREAD_PRIO_INHERIT*

- *PTHREAD_PRIO_PROTECT*
- *PTHREAD_PRIO_NONE*

3.3.2 Deadlocks

A deadlock occurs when multiple threads hold each other's locks and these threads attempt to acquire each other's locks with infinite timeout.

Recommended Practices to Avoid Deadlocks:

- Always acquire multiple mutexes in a consistent global order across all threads.
- Use try-lock mechanisms (*pthread_mutex_trylock()*) to attempt acquiring locks without blocking indefinitely.
- Use timed lock mechanisms (*pthread_mutex_timedlock()*) to limit the wait time for acquiring a lock.

Note

The kernel does not provide built-in deadlock detection or prevention.

3.4 POSIX Message Queue

POSIX Message Queues provide a mechanism for threads to communicate by sending and receiving messages in priority order, and in FIFO order among messages of equal priority. Message queues support multiple producers and consumers, allowing threads to exchange data safely and efficiently. Message queues can be used by including `mqueue.h`.

A message queue is created using *mq_open()* and destroyed using *mq_close()* and *mq_unlink()*. Messages are sent to the queue using *mq_send()* / *mq_timedsend()* and received using *mq_receive()* / *mq_timedreceive()*.

The queue's size attributes are set when it is created and cannot be changed afterwards. *mq_setattr()* switches only the calling descriptor between blocking and non-blocking mode, and *mq_getattr()* retrieves the attributes, including that mode. Default attributes are used if no attributes are specified during creation. The default attributes are defined as:

- *mq_maxmsg*: 10
- *mq_msgsize*: 256 bytes

Message queues dynamically allocate memory from the kernel heap for messages. The kernel heap must be configured with sufficient memory with *QRT_CFG_KERNEL_HEAP_SIZE*.

Message queues can be enabled by setting configuration option *QRT_CFG_MESSAGE_QUEUE_ENABLE*. The number of message queues is not fixed; queues are allocated from the kernel heap and are limited only by available memory.

3.5 POSIX Timer

POSIX Timers provide a mechanism for executing callback functions at specified intervals or after a certain delay. Timers can be used by including `time.h`.

Timers can be created using *timer_create()* and deleted with *timer_delete()*. Timers can be started with *timer_settime()* and stopped with *timer_settime()* by setting zero time. When a timer expires, it invokes a user-defined callback function. With memory protection enabled, only a privileged thread can create a timer with a callback; *timer_create()* returns `EPERM` to an unprivileged thread.

Software timers can be enabled by setting configuration option *QRT_CFG_TIMER_ENABLE*. Each timer is allocated from the kernel heap when created, so the number of concurrent timers is bounded only by *QRT_CFG_KERNEL_HEAP_SIZE* rather than a fixed limit.

3.6 POSIX Clock

POSIX Clocks provide a mechanism for retrieving and manipulating time values. Clocks can be used by including `time.h`.

The kernel supports multiple clock sources, including:

- *CLOCK_REALTIME*: System-wide real-time clock.
- *CLOCK_MONOTONIC*: Monotonic clock that cannot be set and represents elapsed time since an unspecified starting point.

Clock API provides functions for retrieving the current time with *clock_gettime()*, setting the time with *clock_settime()*, and querying the clock resolution with *clock_getres()*.

Threads can suspend execution for a relative interval with *nanosleep()*, or against a specific clock with a relative or absolute expiry with *clock_nanosleep()*.

3.7 System Call Extension

System Call Extensions are designed for invoking custom system calls in kernel context when memory protection is enabled. They enable unprivileged threads to access peripheral devices.

Privileged threads can register System Call Extensions with *qrt_syscall_register()*. System Call Extensions are invoked with *qrt_syscall_invoke()*.

System Call Extensions can be enabled by setting configuration option *QRT_CFG_SYSCALL_EXT_ENABLE* and setting configuration option *QRT_CFG_SYSCALL_EXT_LIMIT* with non-zero value.

3.8 Deferred Call

QRT supports deferred calls, allowing ISRs and callbacks to split workload between urgent and less urgent work. Deferred calls are registered with *qrt_defer_call()*. Deferred calls are executed in order they were pushed and only after all interrupt handlers have completed.

System calls are not directly available from ISR context. Instead, ISRs must use the deferred call to invoke system calls. Deferred call can safely invoke the following system calls:

- *pthread_cancel()* and *pthread_setschedprio()*
- *sem_post()*, *sem_trywait()* and *sem_getvalue()*
- *mq_send()* and *mq_timedsend()*, on a descriptor with `O_NONBLOCK` set
- *clock_gettime()*, *timer_gettime()* and *timer_settime()*
- *qrt_systimer_getmonotonic()*

Deferred Calls can be enabled by enabling *QRT_CFG_DEFER_ENABLE* and by setting configuration option *QRT_CFG_DEFER_LIMIT* with non-zero value.

CHAPTER

FOUR

PART IV: MEMORY PROTECTION

QRT integrates robust Memory Protection capabilities designed to enhance system stability, security, and reliability by isolating threads and system resources. Utilizing the hardware MPU available on ARM Cortex processors, QRT provides key capabilities described in this chapter.

4.1 Privileged and Unprivileged Modes

QRT operates using two execution levels:

- **Privileged Mode** - Full access to system memory, peripherals, and kernel functions
- **Unprivileged Mode** - Restricted access enforced by the MPU

Threads in QRT can run in either mode, while the kernel, idle thread, and ISRs execute in privileged mode. This ensures user threads cannot directly modify critical system resources, significantly enhancing system stability and security.

4.2 Stack Isolation

QRT isolates individual thread stacks, preventing unprivileged threads from inadvertently accessing or corrupting each other's memory. Unprivileged threads have no access to kernel memory.

4.3 Memory Regions

QRT defines a fixed set of memory regions configured at thread creation. These regions isolate ROM, RAM, and threads. Total of six memory regions are defined as follows:

Table 1: Memory Map

Region Type	Memory Type	Share-able	Exe-cute	Privileged Access	Unprivileged Access
ROM	Normal, Write-Through	No	Yes	Read-Only	Read-Only
RAM	Normal, Write-Back Write-Allocate[1]	Yes	No	Read/Write	Read/Write
QRT ROM	Normal, Write-Through	No	Yes	Read-Only	No Access
QRT RAM	Normal, Write-Back Write-Allocate[1]	No	No	Read/Write	No Access
Thread Metadata	Device / Device-nGnRE	No	No	Read/Write[2], Read-Only[3]	Read-Only
Thread Stack	Normal, Write-Back[1]	No	No	Read/Write	Read/Write

Note

[1] Read-Allocate is supported only on ARMv8-M. ARMv7-M supports Write-Allocate as part of fixed Write-Back behavior but does not allow explicit control over allocation policy.
[2] ARMv6-M and ARMv7-M.
[3] ARMv8-M.

Privileged threads have the architectures default memory map with the regions overridden as defined in the table *Memory Map* above. Unprivileged threads have no peripheral access by default.

Any unprivileged peripheral access must be performed with the use of System Call Extension (see *qrt_syscall_register()* and *qrt_syscall_invoke()*) or the thread memory region must be explicitly added.

Additional thread memory regions can be added with *qrt_mpu_threadregionadd()* and removed with *qrt_mpu_threadregionremove()* as needed. Adding and removing regions is only permitted from privileged threads.

4.4 Memory Fault Reporting

QRT provides detailed memory fault reporting to assist developers in diagnosing and resolving MPU violations efficiently. When an MPU fault occurs, QRT captures and reports the following critical details:

- **Faulting Thread** - The identifier of the thread that caused the violation
- **Fault Address** - The memory address that triggered the fault
- **Faulting Instruction** - The Program Counter (PC) at the time of the fault
- **Fault Status Code** - The raw Configurable Fault Status Register (CFSR) value for full debugging insight

The most recent fault details are passed to the custom fault callback if set using *qrt_mpu_faultcallbackset()*. The callback runs in privileged mode, so only privileged threads can set it; an unprivileged thread gets `EPERM`. The same holds for the other functions the kernel calls on behalf of a thread: a timer callback (*timer_create()* with `SIGEV_THREAD`), a *pthread_once()* routine and a key destructor (*pthread_key_create()*) can only be set by privileged threads. Upon detection, the violating thread is terminated, and all mutexes it holds are unlocked to prevent deadlocks.

Note

Memory Fault reporting is not available on ARMv6-M and ARMv8-M Baseline.

4.5 Configuration

Memory Protection can be enabled or disabled using the configuration option *QRT_CFG_MPU_ENABLE*. The number of regions is defined with the configuration option *QRT_CFG_MPU_ENTRIES*.

QRT requires a correctly configured linker script to define memory regions and ensure proper task and kernel placement. The following symbols must be defined in the linker script:

- `__ROM_BASE` - Start address of the ROM region
- `__ROM_SIZE` - Size of the ROM region

- `__RAM_BASE` - Start address of the RAM region
- `__RAM_SIZE` - Size of the RAM region
- `__QRT_ROM_BASE` - Start address of the QRT kernel ROM region
- `__QRT_ROM_SIZE` - Size of the QRT kernel ROM region
- `__QRT_RAM_BASE` - Start address of the QRT kernel RAM region
- `__QRT_RAM_SIZE` - Size of the QRT kernel RAM region

Due to hardware requirements memory region alignment and sizes must conform to MPU specifications:

- Region sizes must be a power of two (e.g., 32B, 64B, 128B, etc.)
- Regions must be aligned to their size (e.g., a 4KB region must be aligned to a 4KB boundary) for ARMv6 and ARMv7
- Regions must be aligned to 32 bytes for ARMv8

These requirements apply to the symbols introduced above as well.

Users are encouraged to refer to the demo application for a working example of a linker script configuration. The linker script should be modified as needed to match the hardware requirements.

CHAPTER

FIVE

PART V: KERNEL CONFIGURATION

This chapter details the available configuration options, covering system parameters, scheduling settings, memory protection, interrupt handling, and enabled services. QRT settings are defined at compile-time and must be adjusted before building the kernel. Developers can:

- Enable or disable kernel features (e.g., FPU support, caching).
- Adjust system parameters (e.g., thread limits, total memory allocated for thread stacks).
- Define memory protection settings (e.g., MPU regions).

5.1 Configuration Options

Configuration options for QRT Real-Time Kernel.

Allows configuring processor core, MPU, FPU, System Timer, scheduler, interrupt handling, IPC and kernel.

Defines

QRT_CPU_RV32E 12

RISC-V 32-bit embedded

QRT_CPU_RV32I 13

RISC-V 32-bit

QRT_CPU_CORTEX_A15 20

ARM Cortex-A15 (AArch32)

QRT_CPU_CORTEX_R5 21

ARM Cortex-R5

QRT_CFG_CPU_CORE

Options:

- QRT_CPU_CORTEX_M0P
- QRT_CPU_CORTEX_M3
- QRT_CPU_CORTEX_M4
- QRT_CPU_CORTEX_M7

- QRT_CPU_CORTEX_M23
- QRT_CPU_CORTEX_M33
- QRT_CPU_CORTEX_M35P
- QRT_CPU_CORTEX_M52
- QRT_CPU_CORTEX_M55
- QRT_CPU_CORTEX_M85
- QRT_CPU_RV32E
- QRT_CPU_RV32I

QRT_CFG_ICACHE_ENABLE (0)

Enable instruction cache (if supported by CPU core)

Options:

- 0: Instruction cache disabled
- 1: Instruction cache enabled

QRT_CFG_DCACHE_ENABLE (0)

Enable data cache (if supported by CPU core)

Options:

- 0: Data cache disabled
- 1: Data cache enabled

QRT_CFG_THREAD_LIMIT (256)

Maximum number of threads.

QRT_CFG_CLEANUP_LIMIT (4u)

Maximum cleanup handlers per thread.

QRT_CFG_STACK_POOL_SIZE (65536u)

Total stack pool size in bytes.

QRT_CFG_THREAD_STACK_DEFAULT (2048u)

Default thread stack size in bytes.

QRT_CFG_THREAD_STACK_LIMIT_CHECK (0)

Enable stack limit checking for threads (ARMv8-M only)

Options:

- 0: Stack limit checking disabled
- 1: Stack limit checking enabled

QRT_CFG_THREAD_NAME_ENABLE (1)

Enable thread naming support.

Options:

- 0: Thread naming disabled
- 1: Thread naming enabled

QRT_CFG_SCHED_TIME_SLICE (10u)

Default time slice in milliseconds.

QRT_CFG_SCHED_TIME_SLICE_MIN_US (100u)

Minimum round-robin time slice in microseconds.

Must stay well above the time one System Timer re-arm pass takes on this core, or a thread's slice ends before it can run.

QRT_CFG_IDLE_STACK_SIZE (128u)

Idle thread stack size in bytes.

QRT_CFG_IDLE_SLEEP_ENABLE (0)

Enable sleep mode in idle thread.

Options:

- 0: Sleep disabled (most clock signals stopped)
- 1: Sleep enabled (CPU enters sleep mode when idle)

QRT_CFG_IDLE_DEEP_SLEEP_ENABLE (0)

Enable deep sleep mode in idle thread. Overrides configuration option QRT_CFG_IDLE_SLEEP_ENABLE.

Options:

- 0: Deep sleep disabled (all clock signals stopped)
- 1: Deep sleep enabled (CPU enters deep sleep mode when idle)

QRT_CFG_ATOMIC_DISABLE_IRQ

Atomic operation method.

Options:

- 0: Use LDREX/STREX (ARM) or Load-Reserved/Store-Conditional (RISC-V) instructions for atomic operations
- 1: Disable interrupts for atomic operations

QRT_CFG_MPU_ENABLE (0)

Enable Memory Protection Unit support.

Options:

- 0: MPU disabled (all memory accessible)
- 1: MPU enabled (enforces memory access restrictions)

QRT_CFG_MPU_ENTRIES (8u)

Number of MPU regions to configure.

QRT_CFG_TZ_SECURE_WORLD (0u)

TrustZone configuration (ARMv8-M only)

Options:

- 0: Running in Non-Secure World
- 1: Running in Secure World

QRT_CFG_SEMAPHORE_ENABLE (1)

Enable semaphore support.

Options:

- 0: Semaphores disabled
- 1: Semaphores enabled

QRT_CFG_MUTEX_ENABLE (1)

Enable mutex support.

Options:

- 0: Mutexes disabled
- 1: Mutexes enabled

QRT_CFG_MESSAGE_QUEUE_ENABLE (1)

Enable message queue support.

Options:

- 0: Message queues disabled
- 1: Message queues enabled

QRT_CFG_TIMER_ENABLE (1)

Enable software timer support.

Options:

- 0: Software timers disabled
- 1: Software timers enabled

QRT_CFG_ERRNO_ENABLE (1)

Enable errno support.

Options:

- 0: errno disabled
- 1: errno enabled

QRT_CFG_DEFER_ENABLE (1u)

Enable deferred calls.

Options:

- 0: Deferred calls disabled
- 1: Deferred calls enabled

QRT_CFG_DEFER_LIMIT (8u)

Maximum number of deferred calls.

QRT_CFG_SYSCALL_EXT_ENABLE (0u)

Enable System Call Extension support (requires MPU)

Options:

- 0: System Call Extensions disabled
- 1: System Call Extensions enabled

QRT_CFG_SYSCALL_EXT_LIMIT (8u)

Maximum number of registered System Call Extensions.

QRT_CFG_KERNEL_IDLE_HOOK_REGISTER_ENABLE (1)

Per-system-call enable switches.

Each option below includes (1) or excludes (0) a single system call (or a small related group) from the build, allowing the linker to drop the associated kernel service and so reduce the kernel memory footprint. A call that is disabled but still invoked returns ENOSYS.

Options:

- 0: System call disabled
- 1: System call enabled Enable *qrt_kernel_idlehookregister()*.

QRT_CFG_CLOCK_GETRES_ENABLE (1)

Enable *clock_getres()*.

QRT_CFG_CLOCK_GETTIME_ENABLE (1)

Enable *clock_gettime()*.

QRT_CFG_CLOCK_SETTIME_ENABLE (1)

Enable *clock_settime()*.

QRT_CFG_CLOCK_NANOSLEEP_ENABLE (1)

Enable *clock_nanosleep()*.

QRT_CFG_NANOSLEEP_ENABLE (1)

Enable *nanosleep()*.

QRT_CFG_PTHREAD_CLEANUP_ENABLE (1)

Enable pthread cleanup handlers: *pthread_cleanup_push()*/pop().

QRT_CFG_PTHREAD_GET_SCHED_PARAM_ENABLE (1)

Enable *pthread_getschedparam()*.

QRT_CFG_PTHREAD_SET_SCHED_PARAM_ENABLE (1)

Enable *pthread_setschedparam()*.

QRT_CFG_PTHREAD_SET_SCHED_PRIO_ENABLE (1)

Enable *pthread_setschedprio()*.

QRT_CFG_PTHREAD_KEY_ENABLE (1)

Enable pthread thread-specific data: *pthread_key_create()*/delete(), *pthread_getspecific()*/setspecific().

QRT_CFG_PTHREAD_ONCE_ENABLE (1)

Enable *pthread_once()*.

QRT_CFG_PTHREAD_ATTR_GET_INHERIT_SCHED_ENABLE (1)

Enable *pthread_attr_getinheritsched()*.

QRT_CFG_PTHREAD_ATTR_SET_INHERIT_SCHED_ENABLE (1)

Enable *pthread_attr_setinheritsched()*.

QRT_CFG_PTHREAD_ATTR_GET_STACK_ENABLE (1)

Enable *pthread_attr_getstack()*.

QRT_CFG_PTHREAD_ATTR_SET_STACK_ENABLE (1)

Enable *pthread_attr_setstack()*.

QRT_CFG_PTHREAD_ATTR_GET_STACK_SIZE_ENABLE (1)

Enable *pthread_attr_getstacksize()*.

QRT_CFG_PTHREAD_ATTR_SET_STACK_SIZE_ENABLE (1)

Enable *pthread_attr_setstacksize()*.

QRT_CFG_PTHREAD_ATTR_GET_SCHED_POLICY_ENABLE (1)

Enable *pthread_attr_getschedpolicy()*.

QRT_CFG_PTHREAD_ATTR_SET_SCHED_POLICY_ENABLE (1)

Enable *pthread_attr_setschedpolicy()*.

QRT_CFG_PTHREAD_ATTR_GET_SCHED_PARAM_ENABLE (1)

Enable *pthread_attr_getschedparam()*.

QRT_CFG_PTHREAD_ATTR_SET_SCHED_PARAM_ENABLE (1)

Enable *pthread_attr_setschedparam()*.

QRT_CFG_PTHREAD_ATTR_GET_DETACH_STATE_ENABLE (1)

Enable *pthread_attr_getdetachstate()*.

QRT_CFG_PTHREAD_ATTR_SET_DETACH_STATE_ENABLE (1)

Enable *pthread_attr_setdetachstate()*.

QRT_CFG_PTHREAD_ATTR_SET_TIMESLICE_ENABLE (1)

Enable *pthread_attr_settimeslice_np()*.

QRT_CFG_PTHREAD_ATTR_GET_TIMESLICE_ENABLE (1)

Enable *pthread_attr_gettimeslice_np()*.

QRT_CFG_PTHREAD_ATTR_SET_PRIVILEGE_ENABLE (1)

Enable *pthread_attr_setprivilege_np()* (requires MPU).

QRT_CFG_PTHREAD_ATTR_GET_PRIVILEGE_ENABLE (1)

Enable *pthread_attr_getprivilege_np()* (requires MPU).

QRT_CFG_PTHREAD_CANCEL_ENABLE (1)

Enable pthread cancellation: *pthread_cancel()*, *pthread_setcancelstate()*/type(), *pthread_testcancel()*.

QRT_CFG_SYSTIMER_GET_MONOTONIC_ENABLE (1)

Enable *qrt_systimer_getmonotonic()*.

QRT_CFG_SCHED_YIELD_ENABLE (1)

Enable *sched_yield()*.

QRT_CFG_KERNEL_HEAP_SIZE (4096u)

Kernel heap size in bytes.

QRT_CFG_FPU_ENABLE (0)

Enable Floating-Point Unit support.

Options:

- 0: FPU disabled (no floating-point support)
- 1: FPU enabled (floating-point operations supported)

QRT_CFG_FPU_STACK_ENABLE (0)

Enable FPU context stacking.

Options:

- 0: Never save FPU context (single-thread FPU use only)
- 1: Save FPU context on context switches (multi-thread FPU support)

QRT_CFG_PERF_ENABLE (0)

Enable performance (WCET) instrumentation.

When enabled, the kernel times each system call and context switch with the CPU cycle counter and records the worst case per service (see the qrt_perf module). Adds a small overhead to every system call; leave disabled in production.

Options:

- 0: Performance instrumentation disabled
- 1: Performance instrumentation enabled

QRT_CFG_CONSOLE_ENABLE (1)

Enable kernel console output.

When enabled, the kernel prints a boot banner and diagnostic messages via the qrt_console_puts() hook. When disabled, every console call site compiles to nothing, leaving zero footprint.

Options:

- 0: Console output disabled
- 1: Console output enabled

CHAPTER

SIX

PART VI: MEMORY FOOTPRINT

This chapter reports the kernel's ROM and RAM footprint.

Note

Footprint figures measured with `arm-none-eabi-gcc (Arm GNU Toolchain 15.3.Rel1 (Build arm-15.180)) 15.3.1 20260627` at `-Os`

6.1 How the figures are measured

- **QRT_ROM** — kernel code and read-only data.
- **QRT_RAM** — kernel statically-allocated data and BSS, **excluding** the configured kernel heap (*QRT_CFG_KERNEL_HEAP_SIZE*) and the per-thread stack pool (*QRT_CFG_STACK_POOL_SIZE*). Those are integrator-sized working memory, not fixed kernel overhead; the total kernel RAM is `QRT_RAM` plus the heap and stack sizes you configure.

All builds use `-Os` with the MPU and FPU disabled.

6.2 Minimal configuration

The *absolute floor*: every optional feature compiled out (all `QRT_CFG_*_ENABLE` options set to `0`) — a single-threaded scheduler with the System Timer and the system-call dispatch path only.

Table 1: Minimal kernel footprint (absolute floor, -Os, MPU/FPU disabled)

Target	QRT_ROM	QRT_RAM
Cortex-M3	12.4 KiB (12,708 B)	3.6 KiB (3,648 B)

6.3 Per-feature ROM cost

Starting from that floor, the table below gives the ROM each optional subsystem adds when enabled on its own. See *Part V: Kernel Configuration* for the matching `QRT_CFG_*_ENABLE` options.

Table 2: Per-feature ROM cost on Cortex-M3 (added to the 12.4 KiB floor, -Os)

Feature	ROM added
Semaphores	452 B
Mutexes	1,264 B
Message queues	1,364 B
Software timers	676 B
Thread naming	284 B
Pthread keys	284 B
Deferred calls	140 B

6.4 Kernel object sizes

The RAM each kernel object instance occupies. Allocation differs by lifetime:

- **Synchronization / IPC objects / thread handles** (*sem_t*, *pthread_mutex_t*, *mqd_t*, *pthread_t*, *timer_t* ...) are opaque handles; the figures below are the internal control blocks they refer to, allocated from the kernel heap on create.
- **Attribute objects** (*pthread_attr_t*, *pthread_mutexattr_t*) are *caller-owned*: the caller declares them (typically on the stack) and the kernel initialises them in place, so they are transient and never touch the heap.

A message queue additionally allocates its name and its message buffer (*mq_attr.mq_maxmsg* × *mq_attr.mq_msgsize*) alongside the control block. The TCB entry is a range because it is configuration-dependent.

Table 3: Kernel object sizes (RAM per instance)

Object	Size	Notes
Thread (*pthread_t*)	80-184 B	Allocated by *pthread_create()*. Size varies with config; see *pthread table*
Thread attributes (*pthread_attr_t*)	56 B	Transient, on the creator's stack
Semaphore (*sem_t*)	44 B	Allocated by *sem_init()*
Mutex (*pthread_mutex_t*)	56 B	Allocated by *pthread_mutex_init()*
Mutex attributes (*pthread_mutexattr_t*)	20 B	Transient, on the creator's stack
Message queue	116 B	Allocated by *mq_open()*
Message queue descriptor (*mqd_t*)	12 B	Allocated by *mq_open()*
Software timer (*timer_t*)	56 B	Allocated by *timer_create()*

6.5 Thread Control Block

Each thread needs one Thread Control Block (TCB). Its size depends on which optional features are enabled. The table gives the absolute TCB size for several representative configurations. The MPU-enabled rows are shown for *QRT_CFG_MPU_ENTRIES* of 8, 12 and 16, because the per-thread RAM grows with the configured region count.

Table 4: TCB size on Cortex-M3

Configuration	TCB size
Minimal (all optional fields off)	80 B
Thread name	88 B
Mutexes	88 B
MPU (8 regions)	112 B
MPU (12 regions)	144 B
MPU (16 regions)	176 B
Thread naming + mutexes + MPU (8 regions)	120 B
Thread naming + mutexes + MPU (12 regions)	152 B
Thread naming + mutexes + MPU (16 regions)	184 B

CHAPTER

SEVEN

PART VII: PROFILING

This chapter covers the kernel's timing characteristics. Unlike the *Part VI: Memory Footprint* figures, which are fixed at build time, these have to be measured on the target — they depend on the exact hardware and build.

> **Note**
>
> Every timing figure depends on the CPU core, clock frequency, compiler and optimisation level, and the cache / flash wait-state configuration. Record those conditions next to any measurement; a cycle count without them is not comparable.

7.1 Enabling instrumentation

Build with *QRT_CFG_PERF_ENABLE* to have the kernel measure its own worst-case execution time (WCET). It records the longest time taken by each system call, by a context switch and by the SysTimer handler, and the longest it holds interrupts off. Each measured operation adds a small, fixed cost, so leave the option disabled in production builds.

Measurement needs a CPU cycle counter. Every supported core has one, except Cortex-M0+ and Cortex-M23 have none.

7.2 Reading the results

Read the figures with *qrt_perf_read()*, which writes a plain-text report into a buffer you supply:

```
#include "qrt.h"

char buf[1024];

qrt_perf_read(buf, sizeof(buf));
fputs(buf, stdout);
```

Each line is a name and its worst-case cycle count. The exercised system calls come first, then context_switch, irq_mask and systimer. For example:

```
sem_wait         1234
mutex_lock        456
kernel_boot      8901
context_switch    312
irq_mask           48
systimer          176
```

The report is truncated to fit the buffer and is always NULL-terminated; *qrt_perf_read()* returns the number of bytes written.

Note

Halting the core in a debugger while instrumentation is enabled does not corrupt the reported worst-case figures.

7.3 Worst-case figures

Measured on an ARM Cortex-M3 at 64 MHz. The system-call and context-switch figures are not skewed by an interrupt landing mid-call.

Three kernel-wide figures accompany the per-call system-call times:

- `context_switch` — saving the outgoing thread's context and restoring the next.
- `irq_mask` — the longest the kernel keeps interrupts disabled; nothing can be serviced in that window, so it bounds the kernel's contribution to interrupt latency.
- `systimer` — the longest run of the SysTimer handler, which runs at the highest priority and so holds off every lower-priority interrupt while it executes, adding to their latency.

Figure	Cycles	Time
`context_switch`	—	—
`irq_mask`	—	—
`systimer`	—	—

CHAPTER

EIGHT

PART VIII: API REFERENCE

This section provides a comprehensive reference for all QRT API functions, data types, and constants.

8.1 POSIX

This section provides a comprehensive reference for all QRT API functions, data types, and constants.

8.1.1 POSIX Compliance Matrix

QRT implements a portable subset of the POSIX.1-2024 API focused on real-time and embedded use. The following tables list supported APIs.

Threads (pthread.h)

Function	Status
pthread_create()	Supported
pthread_join()	Supported
pthread_detach()	Supported
pthread_exit()	Supported
pthread_self()	Supported
pthread_equal()	Supported
pthread_cleanup_push()	Supported
pthread_cleanup_pop()	Supported
pthread_cancel()	Supported
pthread_setcancelstate()	Supported
pthread_setcanceltype()	Supported
pthread_testcancel()	Supported
pthread_once()	Supported
pthread_getschedparam()	Supported
pthread_setschedparam()	Supported
pthread_setschedprio()	Supported
pthread_atfork()	Not supported
pthread_kill()	Not supported
pthread_setconcurrency()	Not supported
pthread_getconcurrency()	Not supported
pthread_getcpuclockid()	Not supported
pthread_sigmask()	Not supported
pthread_spin_init()	Not supported
pthread_spin_destroy()	Not supported
pthread_spin_lock()	Not supported
pthread_spin_trylock()	Not supported
pthread_spin_unlock()	Not supported

Function	Status
pthread_attr_init()	Supported
pthread_attr_destroy()	Supported
pthread_attr_setstack()	Supported
pthread_attr_getstack()	Supported
pthread_attr_setstacksize()	Supported
pthread_attr_getstacksize()	Supported
pthread_attr_setschedparam()	Supported
pthread_attr_getschedparam()	Supported
pthread_attr_setschedpolicy()	Supported
pthread_attr_getschedpolicy()	Supported
pthread_attr_setdetachstate()	Supported
pthread_attr_getdetachstate()	Supported
pthread_attr_setinheritsched()	Supported
pthread_attr_getinheritsched()	Supported
pthread_attr_setscope()	Not supported
pthread_attr_getscope()	Not supported
pthread_attr_setguardsize()	Not supported
pthread_attr_getguardsize()	Not supported

Function	Status
pthread_mutex_init()	Supported
pthread_mutex_destroy()	Supported
pthread_mutex_lock()	Supported
pthread_mutex_trylock()	Supported
pthread_mutex_timedlock()	Supported
pthread_mutex_clocklock()	Supported
pthread_mutex_unlock()	Supported
pthread_mutex_setprioceiling()	Supported
pthread_mutex_getprioceiling()	Supported
pthread_mutex_consistent()	Not supported

Function	Status
pthread_mutexattr_init()	Supported
pthread_mutexattr_destroy()	Supported
pthread_mutexattr_settype()	Supported
pthread_mutexattr_gettype()	Supported
pthread_mutexattr_setprotocol(	Supported
pthread_mutexattr_getprotocol(	Supported
pthread_mutexattr_setprioceili	Supported
pthread_mutexattr_getprioceili	Supported
pthread_mutexattr_setpshared()	Not supported
pthread_mutexattr_getpshared()	Not supported
pthread_mutexattr_setrobust()	Not supported
pthread_mutexattr_getrobust()	Not supported

Function	Status
pthread_cond_init()	Not supported
pthread_cond_destroy()	Not supported
pthread_cond_wait()	Not supported
pthread_cond_timedwait()	Not supported
pthread_cond_signal()	Not supported
pthread_cond_broadcast()	Not supported
pthread_condattr_init()	Not supported
pthread_condattr_destroy()	Not supported
pthread_condattr_getclock()	Not supported
pthread_condattr_setclock()	Not supported
pthread_condattr_getpshared()	Not supported
pthread_condattr_setpshared()	Not supported

Function	Status
pthread_rwlock_init()	Not supported
pthread_rwlock_destroy()	Not supported
pthread_rwlock_rdlock()	Not supported
pthread_rwlock_tryrdlock()	Not supported
pthread_rwlock_wrlock()	Not supported
pthread_rwlock_trywrlock()	Not supported
pthread_rwlock_unlock()	Not supported
pthread_rwlock_timedrdlock()	Not supported
pthread_rwlock_timedwrlock()	Not supported
pthread_rwlock_clockrdlock()	Not supported
pthread_rwlock_clockwrlock()	Not supported
pthread_rwlockattr_init()	Not supported
pthread_rwlockattr_destroy()	Not supported
pthread_rwlockattr_setpshared(	Not supported
pthread_rwlockattr_getpshared(	Not supported

Function	Status
pthread_barrier_wait()	Not supported
pthread_barrier_init()	Not supported
pthread_barrier_destroy()	Not supported
pthread_barrierattr_init()	Not supported
pthread_barrierattr_destroy()	Not supported
pthread_barrierattr_setpshared	Not supported
pthread_barrierattr_getpshared	Not supported

Function	Status
pthread_key_create()	Supported
pthread_key_delete()	Supported
pthread_setspecific()	Supported
pthread_getspecific()	Supported

Semaphores (semaphore.h)

Function	Status
sem_init()	Supported
sem_destroy()	Supported
sem_wait()	Supported
sem_trywait()	Supported
sem_timedwait()	Supported
sem_clockwait()	Supported
sem_post()	Supported
sem_getvalue()	Supported
sem_open()	Not supported
sem_close()	Not supported
sem_unlink()	Not supported

Message Queues (mqueue.h)

Function	Status
mq_open()	Supported
mq_close()	Supported
mq_unlink()	Supported
mq_send()	Supported
mq_timedsend()	Supported
mq_receive()	Supported
mq_timedreceive()	Supported
mq_setattr()	Supported
mq_getattr()	Supported
mq_notify()	Not supported

Timers and Clocks (time.h, timer.h)

Function	Status
clock_gettime()	Supported
clock_settime()	Supported
clock_getres()	Supported
clock_nanosleep()	Supported
nanosleep()	Supported
clock_getcpuclockid()	Not supported
timer_create()	Supported
timer_delete()	Supported
timer_settime()	Supported
timer_gettime()	Supported
timer_getoverrun()	Not supported

Scheduling (sched.h)

Function	Status
sched_yield()	Supported
sched_getscheduler()	Not supported
sched_setscheduler()	Not supported
sched_getparam()	Not supported
sched_setparam()	Not supported
sched_get_priority_min()	Supported
sched_get_priority_max()	Supported
sched_rr_get_interval()	Not supported

Miscellaneous (unistd.h)

Function	Status
sleep()	Supported
write()	Not supported
read()	Not supported
close()	Not supported

8.1.2 POSIX Thread

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/pthread.h.html

Defines

PTHREAD_CREATE_JOINABLE (0x0002u)

Joinable thread attribute.

PTHREAD_CREATE_DETACHED (0x0004u)

Detached thread attribute.

PTHREAD_CANCEL_ENABLE (0x0008u)

Cancel enable attribute.

PTHREAD_CANCEL_DISABLE (0x0010u)

Cancel disable attribute.

PTHREAD_CANCEL_DEFERRED (0x0020u)

Deferred cancel attribute.

PTHREAD_CANCEL_ASYNCHRONOUS (0x0040u)

Asynchronous cancel attribute.

PTHREAD_PRIO_INHERIT (0x0001u)

Priority inheritance protocol.

PTHREAD_PRIO_NONE (0x0002u)

No priority protocol.

PTHREAD_PRIO_PROTECT (0x0004u)

Priority protect protocol.

PTHREAD_MUTEX_DEFAULT (*PTHREAD_MUTEX_NORMAL*)

Default mutex type.

PTHREAD_MUTEX_ERRORCHECK (0x0008u)

Error-checking mutex type.

PTHREAD_MUTEX_NORMAL (0x0010u)

Normal mutex type.

PTHREAD_MUTEX_RECURSIVE (0x0020u)

Recursive mutex type.

PTHREAD_MUTEX_ROBUST (0x0040u)

Robust mutex type. Not supported.

PTHREAD_MUTEX_STALLED (0x0080u)

Stalled mutex type. Not supported.

PTHREAD_PROCESS_SHARED (0x0100u)

Process-shared mutex attribute. Not supported.

PTHREAD_PROCESS_PRIVATE (0x0200u)

Process-private mutex attribute. Not supported.

PTHREAD_SCOPE_PROCESS (0x0400u)

Process scope mutex attribute. Not supported.

PTHREAD_SCOPE_SYSTEM (0x0800u)

System scope mutex attribute. Not supported.

PTHREAD_INHERIT_SCHED (0x0001u)

A thread created with this setting takes the scheduling policy and priority of its creator.

The policy and priority stored in the attribute object are ignored. The time slice, stack, detach state, name and privilege level still come from the attribute object. The startup thread created by *qrt_kernel_boot()* has no creator, so it uses the attribute object's values.

PTHREAD_EXPLICIT_SCHED (0x0002u)

A thread created with this setting takes its scheduling policy and priority from the attribute object. This is the default.

PTHREAD_PRIVILEGED_NP (0x0001u)

Privileged thread attribute.

PTHREAD_UNPRIVILEGED_NP (0x0002u)

Unprivileged thread attribute.

PTHREAD_FPCONTEXT_ENABLE_NP (0x0001u)

Enable FPU context tracking for the thread.

PTHREAD_FPCONTEXT_DISABLE_NP (0x0002u)

Disable FPU context tracking for the thread.

PTHREAD_ONCE_INIT (0)

Once initialization.

PTHREAD_COND_INITIALIZER (0)

Condition variable initialization. Not supported.

PTHREAD_MUTEX_INITIALIZER

Mutex initialization.

A mutex initialized this way is allocated from the kernel heap on its first use. If the heap is exhausted, that call returns *ENOMEM* and the mutex keeps this initial value, so a later call can retry.

PTHREAD_RWLOCK_INITIALIZER

Read-write lock initialization. Not supported.

PTHREAD_NULL ((pthread_t)-1)

Not equal to valid thread.

PTHREAD_KEYS_MAX (16)

Maximum number of thread-specific data keys.

PTHREAD_DESTRUCTOR_ITERATIONS (1)

Number of iterations for destructors.

Functions

int **pthread_mutex_clocklock**(*pthread_mutex_t* *mutex, *clockid_t* clock_id, const struct *timespec* *abstime)

Acquire `mutex`, blocking until `abstime` on `clock_id` if it is already held.

Honors the mutex's type and protocol attributes: error-check detects self-deadlock, recursive permits re-entrant locks up to the kernel's recursion limit, and priority-protect/inherit promote the owner to keep priority inversion bounded. A *PTHREAD_MUTEX_NORMAL* mutex (the default type) relocked by its owner is not detected: the call waits until `abstime`. Every `abstime`, {0, 0} included, is an absolute time on `clock_id`; one with negative seconds lies before the epoch and has already passed.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.
- *EDEADLK*: The mutex is owned by the caller and its type is *PTHREAD_MUTEX_ERRORCHECK*.
- *EAGAIN*: The mutex is recursive and the recursion count saturated.
- *EINVAL*: The thread would have blocked but `abstime` has nanoseconds outside [0, 999'999'999].
- *EINVAL*: The thread would have blocked but `clock_id` is not a supported clock.
- *ETIMEDOUT*: `abstime` passed before the mutex became available; returned at once if it has already passed.

Parameters

- **mutex** – Pointer to the mutex to lock.
- **clock_id** – Clock ID to use for the timeout.
- **abstime** – Pointer to the absolute time to wait until.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

pthread_mutex_t lock    = PTHREAD_MUTEX_INITIALIZER;
```

(continues on next page)

(continued from previous page)

```
unsigned int    counter = 0u;

int update_with_timeout(void)
{
    struct timespec deadline;
    int             status;

    clock_gettime(CLOCK_MONOTONIC, &deadline);
    deadline.tv_sec += 1;

    status = pthread_mutex_clocklock(&lock, CLOCK_MONOTONIC, &deadline);
    if (status == 0)
    {
        counter++;
        pthread_mutex_unlock(&lock);
    }

    return status;
}
```

int **pthread_mutex_timedlock**(*pthread_mutex_t* *mutex, const struct *timespec* *abstime)

Acquire `mutex`, blocking until `abstime` on *CLOCK_REALTIME* if it is already held.

Equivalent to *pthread_mutex_clocklock()* with `clock_id` fixed to *CLOCK_REALTIME*; {0, 0} is an ordinary deadline, and negative seconds lie before the epoch and have passed.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.
- *EDEADLK*: The mutex is owned by the caller and its type is *PTHREAD_MUTEX_ERRORCHECK*.
- *EAGAIN*: The mutex is recursive and the recursion count saturated.
- *EINVAL*: The thread would have blocked but `abstime` has nanoseconds outside [0, 999'999'999].
- *ETIMEDOUT*: `abstime` passed before the mutex became available; returned at once if it has already passed.

Parameters

- **mutex** – Pointer to the mutex to lock.
- **abstime** – Pointer to the absolute time to wait until.

Returns

0 on success, or an error number on failure.

int **pthread_mutex_destroy**(*pthread_mutex_t* *mutex)

Tear down a previously initialized mutex.

The mutex must be currently unlocked; destroying a held mutex fails rather than corrupting kernel bookkeeping. All internal state is reset and the mutex must be re-initialized before reuse.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized.
- *EBUSY*: The mutex is currently locked.

Parameters

- **mutex** – Pointer to the mutex to destroy.

Returns

0 on success, or an error number on failure.

int **pthread_mutex_init**(*pthread_mutex_t* *mutex, const *pthread_mutexattr_t* *attr)

Initialize `mutex` using the type, protocol and ceiling captured in `attr`, or kernel defaults if `attr` is NULL.

The default attribute set is *PTHREAD_MUTEX_DEFAULT* type and *PTHREAD_PRIO_NONE* protocol with a zero priority ceiling. The statically-initialized form *PTHREAD_MUTEX_INITIALIZER* produces the same result, but allocates the mutex on its first use.

Error numbers:

- *EINVAL*: The mutex pointer is NULL, or `attr` is non-NULL but has not been initialized or holds a type, protocol or priority ceiling that its setters would not store.
- *ENOMEM*: The kernel heap has insufficient memory for the mutex.

Parameters

- **mutex** – Pointer to the mutex to initialize.
- **attr** – Pointer to the mutex attributes. NULL for default attributes.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

pthread_mutex_t lock;

int lock_init(void)
{
    pthread_mutexattr_t attr;
    int                 status;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);

    status = pthread_mutex_init(&lock, &attr);

    pthread_mutexattr_destroy(&attr);

    return status;
}
```

int **pthread_mutex_getprioceiling**(const *pthread_mutex_t* *mutex, int *prioceiling)

Read the configured priority ceiling of a mutex.

Only meaningful for mutexes whose protocol is not *PTHREAD_PRIO_NONE*.

Error numbers:

- *EINVAL*: The mutex pointer or `prioceiling` is NULL, the mutex is not initialized, or the mutex protocol is *PTHREAD_PRIO_NONE*.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.

Parameters

- **mutex** – Pointer to the mutex.
- **prioceiling** – Pointer to store the priority ceiling value.

Returns

0 on success, or an error number on failure.

int **pthread_mutex_setprioceiling**(*pthread_mutex_t* *mutex, int prioceiling, int *old_ceiling)

Update the priority ceiling of a mutex and optionally return the previous value.

Honors the ceiling invariant: when the protocol is *PTHREAD_PRIO_PROTECT*, the caller's priority must not exceed the current ceiling. Non-recursive mutexes cannot have their ceiling changed while held; recursive mutexes can, up to the recursion limit.

On an unlocked mutex the new ceiling is stored and applies from the next lock. When the ceiling of a held *PTHREAD_PRIO_PROTECT* mutex is raised above its owner's current priority, the owner is raised to it at once; a lower ceiling leaves the owner's priority unchanged until it releases the last mutex it holds. A *PTHREAD_PRIO_INHERIT* mutex stores the ceiling but does not use it. `old_ceiling` may be NULL.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized, the mutex protocol is *PTHREAD_PRIO_NONE*, `prioceiling` is below the minimum or above the maximum thread priority, or the protocol is *PTHREAD_PRIO_PROTECT* and the caller's priority is above the current ceiling.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.
- *EDEADLK*: The mutex is owned by the caller and its type is *PTHREAD_MUTEX_ERRORCHECK*.
- *EAGAIN*: The mutex is held; either it is non-recursive or its recursion count is saturated.

Parameters

- **mutex** – Pointer to the mutex.
- **prioceiling** – New priority ceiling value.
- **old_ceiling** – Pointer to store the old priority ceiling value.

Returns

0 on success, or an error number on failure.

int **pthread_mutex_lock**(*pthread_mutex_t* *mutex)

Acquire `mutex`, blocking indefinitely if it is already held.

Equivalent to *pthread_mutex_timedlock()* with no deadline. Relocking a *PTHREAD_MUTEX_NORMAL* mutex that the caller already holds deadlocks: the call never returns.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.

- *EDEADLK*: The mutex is owned by the caller and its type is *PTHREAD_MUTEX_ERRORCHECK*.
- *EAGAIN*: The mutex is recursive and the recursion count saturated.

Parameters

- **mutex** – Pointer to the mutex to lock.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

pthread_mutex_t lock    = PTHREAD_MUTEX_INITIALIZER;
unsigned int    counter = 0u;

void counter_increment(void)
{
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
}
```

int **pthread_mutex_trylock**(*pthread_mutex_t* *mutex)

Acquire `mutex` if it is free; otherwise fail immediately.

If the mutex is locked by any thread, the caller included, the call returns *EBUSY* at once instead of waiting. A *PTHREAD_MUTEX_RECURSIVE* mutex the caller holds is locked one more level instead.

Error numbers:

- *EINVAL*: The mutex pointer is NULL or the mutex is not initialized.
- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.
- *EBUSY*: The mutex is held by another thread, or by the caller and its type is *PTHREAD_MUTEX_NORMAL* or *PTHREAD_MUTEX_ERRORCHECK*.
- *EAGAIN*: The mutex is recursive and the recursion count saturated.

Parameters

- **mutex** – Pointer to the mutex to lock.

Returns

0 on success, or an error number on failure.

int **pthread_mutex_unlock**(*pthread_mutex_t* *mutex)

Release one level of ownership of `mutex` and, when fully released, wake one waiter.

For recursive mutexes the call decrements the recursion counter; the mutex is only released and any pending waiters woken once the counter reaches zero. Priority promotions are undone once the thread releases the last mutex it holds.

Error numbers:

- *EINVAL*: The mutex pointer is NULL, the mutex is not initialized, or the caller does not own the mutex and its type is *PTHREAD_MUTEX_NORMAL*.

- *ENOMEM*: This is the first use of a mutex initialized with *PTHREAD_MUTEX_INITIALIZER*, and the kernel heap has insufficient memory for it.
- *EPERM*: The caller does not own the mutex and the mutex type is *PTHREAD_MUTEX_ERRORCHECK* or *PTHREAD_MUTEX_RECURSIVE*.

Parameters

- **mutex** – Pointer to the mutex to unlock.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_destroy**(*pthread_mutexattr_t* *attr)

Invalidate a mutex attributes object so it cannot be used until re-initialized.

Error numbers:

- *EINVAL*: `attr` is NULL or was not previously initialized.

Parameters

- **attr** – Pointer to the mutex attributes to destroy.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_init**(*pthread_mutexattr_t* *attr)

Populate `attr` with default mutex settings: *PTHREAD_MUTEX_DEFAULT* type, *PTHREAD_PRIO_NONE* protocol and a zero priority ceiling.

Error numbers:

- *EINVAL*: `attr` is NULL.

Parameters

- **attr** – Pointer to the mutex attributes to initialize.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_getprioceiling**(const *pthread_mutexattr_t* *attr, int *prioceiling)

Read the priority ceiling currently stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `prioceiling` is NULL, or `attr` has not been initialized.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **prioceiling** – Pointer to store the priority ceiling value.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_setprioceiling**(*pthread_mutexattr_t* *attr, int prioceiling)

Record a new priority ceiling in `attr`.

The ceiling must lie strictly above the idle priority. Mutexes created from `attr` afterwards inherit the new value.

Error numbers:

- *EINVAL*: `attr` is NULL, has not been initialized, or `prioceiling` is below the minimum or above the maximum thread priority.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **prioceiling** – New priority ceiling value.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_getprotocol**(const *pthread_mutexattr_t* *attr, int *protocol)

Read the priority protocol currently stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `protocol` is NULL, `attr` has not been initialized, or it stores a value outside *PTHREAD_PRIO_INHERIT*, *PTHREAD_PRIO_PROTECT*, *PTHREAD_PRIO_NONE*.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **protocol** – Pointer to store the protocol value.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_setprotocol**(*pthread_mutexattr_t* *attr, int protocol)

Record a new priority protocol in `attr`.

Error numbers:

- *EINVAL*: `attr` is NULL, has not been initialized, or `protocol` is not one of *PTHREAD_PRIO_INHERIT*, *PTHREAD_PRIO_PROTECT*, *PTHREAD_PRIO_NONE*.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **protocol** – New protocol value.

Returns

0 on success, or an error number on failure.

int **pthread_mutexattr_gettype**(const *pthread_mutexattr_t* *attr, int *type)

Read the mutex type currently stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `type` is NULL, `attr` has not been initialized, or it stores a value outside *PTHREAD_MUTEX_NORMAL*, *PTHREAD_MUTEX_RECURSIVE*, *PTHREAD_MUTEX_ERRORCHECK*.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **type** – Pointer to store the type value.

Returns
0 on success, or an error number on failure.

int **pthread_mutexattr_settype**(*pthread_mutexattr_t* *attr, int type)

Record a new mutex type in `attr`.

Error numbers:

- *EINVAL*: `attr` is NULL, has not been initialized, or `type` is not one of *PTHREAD_MUTEX_NORMAL*, *PTHREAD_MUTEX_RECURSIVE*, *PTHREAD_MUTEX_ERRORCHECK*.

Parameters

- **attr** – Pointer to the mutex attributes object.
- **type** – New type value.

Returns
0 on success, or an error number on failure.

int **pthread_mutexattr_getpshared**(const *pthread_mutexattr_t* *attr, int *pshared)

Not supported.

Returns
ENOSYS.

int **pthread_mutexattr_setpshared**(*pthread_mutexattr_t* *attr, int pshared)

Not supported.

Returns
ENOSYS.

int **pthread_mutexattr_getrobust**(const *pthread_mutexattr_t* *attr, int *robust)

Not supported.

Returns
ENOSYS.

int **pthread_mutexattr_setrobust**(*pthread_mutexattr_t* *attr, int robust)

Not supported.

Returns
ENOSYS.

int **pthread_mutex_consistent**(*pthread_mutex_t* *mutex)

Not supported.

Returns
ENOSYS.

int **pthread_create**(*pthread_t* *thread, const *pthread_attr_t* *attr, void *(*start_routine)(void *arg), void *arg)

Spawn a new thread that runs `start_routine(arg)` under the settings in `attr`.

When `attr` is NULL the default attribute set is used. If `attr` selects *PTHREAD_INHERIT_SCHED* the new thread takes the caller's scheduling policy and priority. The inherited priority is the one reported by *pthread_getschedparam()*, so a temporary raise by a mutex protocol is not inherited. `attr` is not modified. A

stack is allocated from the thread stack pool when `stack` is NULL in the attributes; otherwise the caller-supplied stack is used as-is. If the new thread's priority is higher than the priority the caller is running at, the new thread runs before the call returns.

Error numbers:

- *EINVAL*: `start_routine` is NULL, the stack size in `attr` is not 8-byte aligned, exceeds 1 GiB or cannot hold the thread's initial context, `attr` holds a priority, scheduling policy, inherit-scheduler setting, detach state or time slice that its setters would not store, or (with MPU support) the stack set with *pthread_attr_setstack()* does not form a valid MPU region.
- *EPERM*: `attr` requests a privileged thread while the caller is unprivileged (only with MPU support).
- *ENOMEM*: The stack pool or kernel heap is exhausted.

Parameters

- **thread** – Pointer to the thread handle. NULL if not needed.
- **attr** – Pointer to the thread attributes. NULL for default attributes.
- **start_routine** – Pointer to the function to be executed by the thread.
- **arg** – Argument to be passed to the thread function.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

unsigned int counter;

void *worker(void *arg)
{
    counter++;

    return NULL;
}

void *main_thread(void *arg)
{
    pthread_t thread;

    if (pthread_create(&thread, NULL, worker, NULL) == 0)
    {
        pthread_join(thread, NULL);
    }

    return NULL;
}
```

void **pthread_cleanup_push**(void (*routine)(void *arg), void *arg)

Register a cleanup handler that will run if the thread exits or is cancelled before the matching *pthread_cleanup_pop()*.

Handler count is limited to *QRT_CFG_CLEANUP_LIMIT*; a push beyond the limit is ignored. The call is restricted to privileged threads; unprivileged callers are silently ignored.

Handlers run in kernel context, not in the thread, and no thread runs until they return, so they must not block or call kernel services. Handlers still registered when the thread terminates run, most recently pushed first, when its resources are released: inside *pthread_join()* for a joinable thread, or at the next context switch for a detached thread. The thread's stack is released only after its handlers have run, so `arg` may point to one of the thread's local variables, provided the function that declares it has not returned when the thread terminates.

Parameters

- **routine** – Pointer to the cleanup handler function.
- **arg** – Argument to be passed to the cleanup handler.

```
#include <pthread.h>

int busy;

void clear_busy(void *arg)
{
    *(int *)arg = 0;
}

void *worker(void *arg)
{
    busy = 1;
    pthread_cleanup_push(clear_busy, &busy);

    pthread_testcancel();

    pthread_cleanup_pop(1);

    return NULL;
}
```

void **pthread_cleanup_pop**(int execute)

Discard the most recently pushed cleanup handler, optionally executing it.

Restricted to privileged threads; unprivileged callers are silently ignored.

See also

pthread_cleanup_push() for example.

Parameters

- **execute** – If non-zero, the cleanup handler is executed.

int **pthread_getschedparam**(*pthread_t* thread, int *policy, struct *sched_param* *param)

Read `thread's` scheduling policy and priority.

For threads that currently own a mutex with priority promotion active, the priority reported is the thread's saved original priority, not the elevated one.

Error numbers:

- *EINVAL*: `thread` or `param` is NULL.

- *ESRCH*: `thread` has terminated.

Parameters

- **thread** – The thread to query.
- **policy** – Pointer to a location where the scheduling policy will be stored.
- **param** – Pointer to a location where the scheduling parameters will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_setschedparam**(*pthread_t* thread, int policy, const struct *sched_param* *param)

Replace `thread's` scheduling policy and priority. May reschedule.

If `thread` owns a mutex whose protocol has temporarily raised its priority, a new priority below the raised one is stored as the saved original and takes effect when the thread releases the last mutex it holds; a higher one takes effect at once. Otherwise the new priority is applied immediately, and a calling thread that lowers its own priority below that of another ready thread is preempted before the call returns. In both cases *pthread_getschedparam()* reports the new priority at once. A ready or running thread goes behind the other ready threads of its new priority, also when the priority is unchanged, so a calling thread yields to a ready thread of equal priority.

Error numbers:

- *EINVAL*: `thread` or `param` is NULL.
- *ESRCH*: `thread` has terminated.
- *ENOTSUP*: `policy` is neither *SCHED_FIFO* nor *SCHED_RR*, or `param->sched_priority` is below *sched_get_priority_min()* or above *sched_get_priority_max()*.

Parameters

- **thread** – The thread to set the scheduling policy and parameters for.
- **policy** – The new scheduling policy. One of:
  - *SCHED_FIFO*
  - *SCHED_RR*
- **param** – Pointer to a structure containing the new scheduling parameters.

Returns

0 on success, or an error number on failure.

int **pthread_setschedprio**(*pthread_t* thread, int prio)

Replace `thread's` scheduling priority while keeping its current policy. May reschedule.

Equivalent to *pthread_setschedparam()* with the thread's current policy, including its handling of a thread whose priority a mutex protocol has raised, except for where the thread is placed among the ready threads of its priority: an unchanged priority leaves it where it is, a raised one places it behind them and a lowered one ahead of them. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*), which *pthread_setschedparam()* may not.

Error numbers:

- *EINVAL*: `thread` is NULL.
- *ESRCH*: `thread` has terminated.
- *ENOTSUP*: `prio` is below *sched_get_priority_min()* or above *sched_get_priority_max()*.

Parameters

- **thread** – The thread to set the priority for.
- **prio** – The new priority value.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

void urgent_increment(unsigned int *count)
{
    struct sched_param param;
    int                policy;

    pthread_getschedparam(pthread_self(), &policy, &param);
    pthread_setschedprio(pthread_self(),
                         sched_get_priority_max(policy));

    (*count)++;

    pthread_setschedprio(pthread_self(), param.sched_priority);
}
```

void **pthread_exit**(void *value_ptr)

Terminate the calling thread with the given exit value.

Wakes any thread joining it, runs the thread's thread-specific-data destructors, and releases the thread's resources; a joinable thread's resources are retained until it has been joined.

```
#include <pthread.h>

void exit_on_error(int status)
{
    if (status != 0)
    {
        pthread_exit(NULL);
    }
}

void *worker(void *arg)
{
    exit_on_error(pthread_setname_np(pthread_self(), "worker"));

    return NULL;
}
```

Parameters

- **value_ptr** – Pointer to the return value of the thread.

int **pthread_equal**(*pthread_t* t1, *pthread_t* t2)

Test whether two thread handles refer to the same thread.

Parameters

- **t1** – First thread identifier.
- **t2** – Second thread identifier.

Returns

0 if equal, non-zero otherwise.

pthread_t **pthread_self**(void)

Return the handle of the calling thread.

Returns

The calling thread's ID.

int **pthread_cancel**(*pthread_t* thread)

Request cancellation of `thread`.

A thread with asynchronous cancellation enabled is terminated immediately. Otherwise it is flagged and the actual cancellation occurs at the next cancellation point. Unprivileged callers may only cancel themselves or other unprivileged threads. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL*: `thread` is NULL or already deleted.
- *EPERM*: The caller is an unprivileged thread and the target is a privileged thread (only with MPU support).

Parameters

- **thread** – The thread to cancel.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

unsigned int iterations;

void *worker(void *arg)
{
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    for (;;)
    {
        iterations++;
        pthread_testcancel();
    }

    return NULL;
}

void worker_stop(pthread_t thread)
{
    pthread_cancel(thread);
    pthread_join(thread, NULL);
}
```

int **pthread_setcancelstate**(int state, int *oldstate)

Enable or disable cancellation on the calling thread and report the previous state.

If the new state combines with the current cancel type to be asynchronously cancellable, and a cancellation is pending, the thread is terminated before this call returns.

Error numbers:

- *EINVAL*: `state` is neither *PTHREAD_CANCEL_ENABLE* nor *PTHREAD_CANCEL_DISABLE*.

Parameters

- **state** – One of the cancelability states:
  - *PTHREAD_CANCEL_ENABLE*
  - *PTHREAD_CANCEL_DISABLE*
- **oldstate** – Pointer to a location where the old state will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_setcanceltype**(int type, int *oldtype)

Choose whether cancellations on the calling thread are honored only at cancellation points or immediately, and report the previous type.

Switching to *PTHREAD_CANCEL_ASYNCHRONOUS* is rejected if the thread is currently the target of a pending join.

Error numbers:

- *EINVAL*: `type` is neither *PTHREAD_CANCEL_DEFERRED* nor *PTHREAD_CANCEL_ASYNCHRONOUS*, or the caller is being joined and asynchronous cancellation was requested.

See also

pthread_cancel() for example.

Parameters

- **type** – One of the cancelability types:
  - *PTHREAD_CANCEL_DEFERRED*
  - *PTHREAD_CANCEL_ASYNCHRONOUS*
- **oldtype** – Pointer to a location where the old type will be stored.

Returns

0 on success, or an error number on failure.

void **pthread_testcancel**(void)

Yield to a pending cancellation, if cancellation is enabled.

If a cancellation request has been delivered while the thread was in deferred state, the thread is terminated here; otherwise the call returns without effect.

See also

pthread_cancel() for example.

int **pthread_join**(*pthread_t* thread, void **value_ptr)

Block the caller until `thread` exits, optionally collecting its exit value.

A thread can be joined by only one thread at a time. Detached threads, asynchronously-cancellable threads, and self-joins are rejected. If the target has already exited the result is collected immediately.

Error numbers:

- *EINVAL*: `thread` is NULL, detached, asynchronously cancellable, or already being joined by another thread.
- *EDEADLK*: The caller tried to join itself.

See also

pthread_create() for example.

Parameters

- **thread** – The thread to wait for.
- **value_ptr** – Pointer to a location where the return value of the thread will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_detach**(*pthread_t* thread)

Transition `thread` to detached state so its resources are reclaimed automatically when it exits.

Error numbers:

- *EINVAL*: `thread` is NULL, already detached, or no longer alive.

Parameters

- **thread** – The thread to detach.

Returns

0 on success, or *EINVAL* if the thread does not exist.

int **pthread_attr_init**(*pthread_attr_t* *attr)

Populate `attr` with kernel defaults: *QRT_CFG_THREAD_STACK_DEFAULT*, *PTHREAD_CREATE_JOINABLE*, *PTHREAD_EXPLICIT_SCHED*, *SCHED_RR* with the default thread priority and the configured time slice, and (with MPU support) *PTHREAD_UNPRIVILEGED_NP*.

Error numbers:

- *EINVAL*: `attr` is NULL.

```c
#include <pthread.h>

void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setname_np(&attr, "worker");
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

Parameters

- **attr** – Pointer to the thread attributes object to initialize.

Returns

0 on success, or an error number on failure.

int **pthread_attr_destroy**(*pthread_attr_t* *attr)

Clear a thread attributes object, after which it must be re-initialized before reuse.

Error numbers:

- *EINVAL*: `attr` is NULL.

Parameters

- **attr** – Pointer to the thread attributes object to destroy.

Returns

0 on success, or an error number on failure.

int **pthread_attr_getinheritsched**(const *pthread_attr_t* *attr, int *inheritsched)

Read the inherit-scheduling setting stored in `attr`.

Determines whether threads created from `attr` inherit the caller's scheduling policy and priority or use the attribute object's own.

Error numbers:

- *EINVAL*: `attr` or `inheritsched` is NULL.

Parameters

- **attr** – Pointer to the thread attributes object.
- **inheritsched** – Pointer to a location where the inherit scheduling attribute will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setinheritsched**(*pthread_attr_t* *attr, int inheritsched)

Record an inherit-scheduling setting in `attr`.

Selects whether threads created from `attr` inherit the caller's scheduling policy and priority (*PTHREAD_INHERIT_SCHED*) or use the attribute object's own (*PTHREAD_EXPLICIT_SCHED*, the default).

Error numbers:

- *EINVAL*: `attr` is NULL.
- *ENOTSUP*: `inheritsched` is neither *PTHREAD_INHERIT_SCHED* nor *PTHREAD_EXPLICIT_SCHED*.

Parameters

- **attr** – Pointer to the thread attributes object.
- **inheritsched** – The inherit scheduling attribute to set. One of:
  - *PTHREAD_INHERIT_SCHED*
  - *PTHREAD_EXPLICIT_SCHED*

Returns

0 on success, or an error number on failure.

int **pthread_attr_getdetachstate**(const *pthread_attr_t* *attr, int *detachstate)

Read the detach state stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `detachstate` is NULL, or `attr` holds a value outside *PTHREAD_CREATE_JOINABLE* and *PTHREAD_CREATE_DETACHED*.

Parameters

- **attr** – Pointer to the thread attributes object.
- **detachstate** – Pointer to a location where the detach state attribute will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setdetachstate**(*pthread_attr_t* *attr, int detachstate)

Record the detach state in `attr`.

Error numbers:

- *EINVAL*: `attr` is NULL, or `detachstate` is neither *PTHREAD_CREATE_JOINABLE* nor *PTHREAD_CREATE_DETACHED*.

Parameters

- **attr** – Pointer to the thread attributes object.
- **detachstate** – The detach state attribute to set. One of:
  - *PTHREAD_CREATE_JOINABLE*
  - *PTHREAD_CREATE_DETACHED*

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getstack**(const *pthread_attr_t* *attr, void **stackaddr, *size_t* *stacksize)

Read the stack base and size stored in `attr`.

A NULL stack base means *pthread_create()* will allocate the stack from the kernel pool.

Error numbers:

- *EINVAL*: `attr`, `stackaddr` or `stacksize` is NULL.

Parameters

- **attr** – Pointer to the thread attribute object.
- **stackaddr** – Pointer to the stack base address.
- **stacksize** – Pointer to a `size_t` variable where the stack size will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setstack**(*pthread_attr_t* *attr, void *stackaddr, *size_t* stacksize)

Record a caller-provided stack region in `attr`.

Both the base address and the size must be 8-byte aligned, and the size must be at least *PTHREAD_STACK_MIN* and at most 1 GiB. With MPU support the stack must also form a valid MPU region, or *pthread_create()* fails with *EINVAL*. On ARMv6-M and ARMv7-M the size must be a power of two and the base aligned to the size; on ARMv8-M both must be multiples of 32 bytes.

Error numbers:

- *EINVAL*: `attr` or `stackaddr` is NULL, `stacksize` is below *PTHREAD_STACK_MIN* or above 1 GiB, or either is not 8-byte aligned.

Parameters

- **attr** – Pointer to the thread attribute object.
- **stackaddr** – Pointer to the stack base address.
- **stacksize** – Stack size.

Returns

0 on success, or an error number on failure.

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

uint64_t worker_stack[256] __attribute__((aligned(2048)));

void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setstack(&attr, worker_stack, sizeof(worker_stack));
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getstacksize**(const *pthread_attr_t* *attr, *size_t* *stacksize)

Read the stack size stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `stacksize` is NULL.

Parameters

- **attr** – Pointer to the thread attribute object.
- **stacksize** – Pointer to a `size_t` variable where the stack size will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setstacksize**(*pthread_attr_t* *attr, *size_t* stacksize)

Record a requested stack size in `attr`.

pthread_create() allocates the stack from the thread stack pool. With MPU support the size is rounded up to form a valid MPU region: on ARMv6-M and ARMv7-M to the next power of two, taking twice that from the pool so the stack can be aligned to its size; on ARMv8-M to a multiple of 32 bytes, taking 32 bytes more. The size must be 8-byte aligned, at least *PTHREAD_STACK_MIN* and at most 1 GiB.

Error numbers:

- *EINVAL*: `attr` is NULL, `stacksize` is below *PTHREAD_STACK_MIN* or above 1 GiB, or `stacksize` is not 8-byte aligned.

Parameters

- **attr** – Pointer to the thread attribute object.
- **stacksize** – Stack size.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>
```

(continues on next page)

(continued from previous page)

```
void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setstacksize(&attr, 4096u);
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getschedpolicy**(const *pthread_attr_t* *attr, int *policy)

Read the scheduling policy stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `policy` is NULL.

Parameters

- **attr** – Pointer to the thread attributes object.
- **policy** – Pointer to a location where the scheduling policy attribute will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setschedpolicy**(*pthread_attr_t* *attr, int policy)

Record a scheduling policy in `attr`.

The default policy set by *pthread_attr_init()* is *SCHED_RR*.

Error numbers:

- *EINVAL*: `attr` is NULL.
- *ENOTSUP*: `policy` is neither *SCHED_FIFO* nor *SCHED_RR*.

Parameters

- **attr** – Pointer to the thread attributes object.
- **policy** – The scheduling policy attribute to set. One of:
  - *SCHED_FIFO*
  - *SCHED_RR*

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

void *worker(void *arg);

void start_worker(void)
{
```

(continues on next page)

(continued from previous page)

```
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getschedparam**(const *pthread_attr_t* *attr, struct *sched_param* *param)

Read the scheduling parameters stored in `attr` into `param`.

Error numbers:

- *EINVAL*: `attr` or `param` is NULL.

Parameters

- **attr** – Pointer to the thread attributes object.
- **param** – Pointer to a location where the scheduling parameters will be stored.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setschedparam**(*pthread_attr_t* *attr, const struct *sched_param* *param)

Record a scheduling-parameter block in `attr`.

The requested priority must lie strictly above the idle priority and must not exceed *sched_get_priority_max()*. Idle priority is reserved for the kernel and cannot be assigned to threads; it is also the floor for priority promotions applied to threads that hold mutexes with priority ceiling or inheritance protocols. On failure `attr` is unchanged.

Error numbers:

- *EINVAL*: `attr` or `param` is NULL.
- *ENOTSUP*: `param->sched_priority` is below *sched_get_priority_min()* or above *sched_get_priority_max()*.

Parameters

- **attr** – Pointer to the thread attributes object.
- **param** – Pointer to the scheduling parameters to set.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t     attr;
    struct sched_param param;

    param.sched_priority = 10;
```

(continues on next page)

(continued from previous page)

```
    pthread_attr_init(&attr);
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getscope**(const *pthread_attr_t* *attr, int *contentionscope)

Not supported.

Returns
ENOSYS.

int **pthread_attr_setscope**(*pthread_attr_t* *attr, int contentionscope)

Not supported.

Returns
ENOSYS.

int **pthread_attr_getguardsize**(const *pthread_attr_t* *attr, *size_t* *guardsize)

Not supported.

Returns
ENOSYS.

int **pthread_attr_setguardsize**(*pthread_attr_t* *attr, *size_t* guardsize)

Not supported.

Returns
ENOSYS.

int **pthread_attr_getname_np**(const *pthread_attr_t* *attr, char *name, *size_t* len)

Copy the thread name stored in `attr` into `name`, truncating to `len` bytes and ensuring null termination.

At most 16 bytes, including the terminating null, are copied.

Error numbers:

- *EINVAL*: `attr` or `name` is NULL.
- *ERANGE*: `len` is 0; nothing is written to `name`.

Parameters

- **attr** – Pointer to the thread attributes object.
- **name** – Pointer to a buffer to store the thread name.
- **len** – Length of the buffer.

Returns
0 on success, or an error number on failure.

int **pthread_attr_setname_np**(*pthread_attr_t* *attr, const char *name)

Record a thread name in `attr`.

The name is copied into `attr` and truncated to 15 characters if longer.

Error numbers:

- *EINVAL*: `attr` or `name` is NULL.

Parameters

- **attr** – Pointer to the thread attributes object.
- **name** – The name to set for the thread.

Returns

0 on success, or an error number on failure.

int **pthread_attr_setprivilege_np**(*pthread_attr_t* *attr, int privilege)

Record the MPU privilege level that threads created from `attr` will run with.

Available only on builds with MPU support; the default produced by *pthread_attr_init()* is *PTHREAD_UNPRIVILEGED_NP*.

Error numbers:

- *EINVAL*: `attr` is NULL, or `privilege` is neither *PTHREAD_PRIVILEGED_NP* nor *PTHREAD_UNPRIVILEGED_NP*.

Parameters

- **attr** – Pointer to the thread attribute object.
- **privilege** – One of the following values:
  - *PTHREAD_PRIVILEGED_NP*
  - *PTHREAD_UNPRIVILEGED_NP*

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

void *worker(void *arg);

void start_worker(void)
{
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setprivilege_np(&attr, PTHREAD_PRIVILEGED_NP);
    pthread_create(NULL, &attr, worker, NULL);
    pthread_attr_destroy(&attr);
}
```

int **pthread_attr_getprivilege_np**(*pthread_attr_t* *attr, int *privilege)

Read the privilege level stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `privilege` is NULL.

Parameters

- **attr** – Pointer to the thread attribute object.
- **privilege** – One of the following values:

- *PTHREAD_PRIVILEGED_NP*
- *PTHREAD_UNPRIVILEGED_NP*

Returns
0 on success, or an error number on failure.

int **pthread_setprivilege_np**(*pthread_t* thread, int privilege)

Change the MPU privilege level of a live thread.

For the calling thread the change takes effect immediately; for another thread the change applies the next time that thread runs. Only privileged threads may call this. The initial main thread starts privileged; drop its privilege with this call if it is not needed.

Error numbers:

- *EINVAL*: `thread` is NULL, or `privilege` is neither *PTHREAD_PRIVILEGED_NP* nor *PTHREAD_UNPRIVILEGED_NP*.
- *EPERM*: The caller is unprivileged.

Parameters

- **thread** – Thread which privilege to set.
- **privilege** – One of the following values:
  - *PTHREAD_PRIVILEGED_NP*
  - *PTHREAD_UNPRIVILEGED_NP*

Returns
0 on success, or an error number on failure.

```
#include <pthread.h>
#include <qrt.h>
#include <time.h>

volatile unsigned int idle_count;
unsigned int          idle_snapshot;

void count_idle(void)
{
    idle_count++;
}

void *main_thread(void *arg)
{
    struct timespec period = { 0, 10000000L };

    (void)arg;
    (void)qrt_kernel_idlehookregister(count_idle);
    (void)pthread_setprivilege_np(pthread_self(), PTHREAD_UNPRIVILEGED_
↪NP);

    for (;;)
    {
        idle_snapshot = idle_count;
        (void)nanosleep(&period, NULL);
```

(continues on next page)

(continued from previous page)

```
    }
    return NULL;
}
```

int **pthread_setname_np**(*pthread_t* thread, const char *name)

Update the human-readable name of a running thread.

The name buffer is allocated on demand from the kernel heap and the string is copied in, so the call can fail when memory is exhausted.

Error numbers:

- *EINVAL*: `thread` or `name` is NULL.
- *ENOMEM*: The thread name buffer could not be allocated.

Parameters

- **thread** – The thread to set the name of.
- **name** – The new name for the thread.

Returns

0 on success, or an error number on failure.

int **pthread_getname_np**(*pthread_t* thread, char *name, *size_t* len)

Copy a running thread's name into `name`, truncating to `len` bytes and ensuring null termination.

Error numbers:

- *EINVAL*: `thread` or `name` is NULL.
- *ERANGE*: `len` is 0; nothing is written to `name`.

Parameters

- **thread** – The thread to get the name of.
- **name** – Pointer to a buffer to store the thread name.
- **len** – Length of the buffer.

Returns

0 on success, or an error number on failure.

int **pthread_setfpcontext_np**(int context)

Enable or disable FPU context tracking for the calling thread.

With *PTHREAD_FPCONTEXT_ENABLE_NP*, the calling thread is marked as an FPU-using thread: every subsequent context switch involving it preserves the full floating-point register file (S0-S31 and FPSCR), a cost paid only for threads that actually use the FPU. With *PTHREAD_FPCONTEXT_DISABLE_NP*, the kernel stops saving and restoring the upper floating-point register bank (S16-S31 and FPSCR) on context switches involving this thread. Acts on the calling thread only.

```
#include <pthread.h>

float level;
```

(continues on next page)

(continued from previous page)

```
void *worker(void *arg)
{
    pthread_setfpcontext_np(PTHREAD_FPCONTEXT_ENABLE_NP);

    level = level * 0.5f;

    pthread_setfpcontext_np(PTHREAD_FPCONTEXT_DISABLE_NP);

    return NULL;
}
```

Warning

After disabling, the thread must not use any FPU instructions unless FPU access is explicitly re-enabled with *pthread_setfpcontext_np*. Failing to do so may result in data corruption or undefined behavior.

Parameters

- **context** – *PTHREAD_FPCONTEXT_ENABLE_NP* to opt into FPU context tracking, or *PTHREAD_FPCONTEXT_DISABLE_NP* to opt out.

Returns

0 on success, *EINVAL* if `context` is not a valid value, or *ENOSYS* if FPU context support is not compiled in.

int **pthread_getconcurrency**(void)

Not supported.

Returns

ENOSYS.

int **pthread_setconcurrency**(int new_level)

Not supported.

Returns

ENOSYS.

int **pthread_atfork**(void (*prepare)(void), void (*parent)(void), void (*child)(void))

Not supported.

Returns

ENOSYS.

void ***pthread_getspecific**(*pthread_key_t* key)

Return the value the calling thread has associated with `key`.

Returns NULL when `key` is out of range or the thread has not stored a value.

Parameters

- **key** – The key to get the thread-specific data for.

Returns

The thread-specific data associated with the key, or NULL if not set.

int **pthread_setspecific**(*pthread_key_t* key, const void *value)

Bind `value` to `key` for the calling thread.

The previous association is overwritten without invoking the destructor.

Parameters

- **key** – The key to set the thread-specific data for.
- **value** – The thread-specific data to associate with the key.

Returns

0 on success, or *EINVAL* if the key is not valid.

int **pthread_key_create**(*pthread_key_t* *key, void (*destructor)(void *value))

Allocate a thread-specific data key with an optional destructor.

The destructor (if non-NULL) is invoked at thread exit for every thread that has stored a non-NULL value under this key. Each thread's association is cleared before the destructor runs, so destructors do not cascade. Destructors run in kernel context while the thread terminates, and no thread runs until they return, so they must not block or call kernel services. With MPU support only privileged threads can register a destructor; unprivileged threads can still create keys without one. Keys live in a fixed-size table sized by *PTHREAD_KEYS_MAX*.

Error numbers:

- *EINVAL*: `key` is NULL.
- *EPERM*: `destructor` is non-NULL and the caller is unprivileged (only with MPU support).
- *EAGAIN*: No free slot in the key table.

Parameters

- **key** – Pointer to the key to be created.
- **destructor** – Pointer to a destructor function to be called when a thread exits.

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

pthread_key_t count_key;

void count_event(void)
{
    unsigned int *count;

    count = pthread_getspecific(count_key);
    (*count)++;
}

void *worker(void *arg)
{
    pthread_setspecific(count_key, arg);

    for (;;)
    {
        count_event();
    }
```

(continues on next page)

(continued from previous page)

```
    return NULL;
}

void start_workers(unsigned int *count_a, unsigned int *count_b)
{
    if (pthread_key_create(&count_key, NULL) == 0)
    {
        pthread_create(NULL, NULL, worker, count_a);
        pthread_create(NULL, NULL, worker, count_b);
    }
}
```

int **pthread_key_delete**(*pthread_key_t* key)

Release `key` so it can be reused.

Existing per-thread values are not freed and no destructors are run; the caller must release any owned resources before deleting the key.

Parameters

- **key** – The key to be deleted.

Returns

0 on success, or *EINVAL* if the key is not valid.

int **pthread_kill**(*pthread_t* thread, int sig)

Not supported.

Returns

ENOSYS.

int **pthread_once**(*pthread_once_t* *once_control, void (*init_routine)(void))

Guarantee `init_routine` runs at most once across all threads that share `once_control`.

The first caller observing the control object in its initial state runs the routine and marks the control as completed; later callers return immediately. The control object must have been initialized with *PTHREAD_ONCE_INIT*; any other value counts as completed. The routine runs in kernel context on behalf of that caller, and no thread runs until it returns, so it must not block or call kernel services.

With MPU support only a privileged thread can run the routine. While the control object is in its initial state, an unprivileged caller gets *EPERM*, the routine does not run and the control object is left unchanged; once it is completed, unprivileged callers return 0 as well. Make the first call from a privileged thread, for example during start-up.

Error numbers:

- *EINVAL*: `once_control` or `init_routine` is NULL.
- *EPERM*: `once_control` is in its initial state and the caller is unprivileged (only with MPU support).

Returns

0 on success, or an error number on failure.

```
#include <pthread.h>

pthread_once_t table_once = PTHREAD_ONCE_INIT;
```

(continues on next page)

(continued from previous page)

```
int            table[16];

void table_init(void)
{
    int i;

    for (i = 0; i < 16; i++)
    {
        table[i] = i * i;
    }
}

int table_get(int index)
{
    pthread_once(&table_once, table_init);

    return table[index];
}
```

int **pthread_sigmask**(int how, const *sigset_t* *set, *sigset_t* *oset)

Not supported.

Returns
ENOSYS.

int **pthread_getcpuclockid**(*pthread_t* thread_id, *clockid_t* *clock_id)

Not supported.

Returns
ENOSYS.

int **pthread_spin_destroy**(*pthread_spinlock_t* *lock)

Not supported.

Returns
ENOSYS.

int **pthread_spin_init**(*pthread_spinlock_t* *lock, int pshared)

Not supported.

Returns
ENOSYS.

int **pthread_spin_lock**(*pthread_spinlock_t* *lock)

Not supported.

Returns
ENOSYS.

int **pthread_spin_trylock**(*pthread_spinlock_t* *lock)

Not supported.

Returns
ENOSYS.

int **pthread_spin_unlock**(*pthread_spinlock_t* *lock)

Not supported.

Returns
ENOSYS.

int **pthread_barrier_destroy**(*pthread_barrier_t* *barrier)

Not supported.

Returns
ENOSYS.

int **pthread_barrier_init**(*pthread_barrier_t* *barrier, const *pthread_barrierattr_t* *attr, unsigned count)

Not supported.

Returns
ENOSYS.

int **pthread_barrier_wait**(*pthread_barrier_t* *barrier)

Not supported.

Returns
ENOSYS.

int **pthread_barrierattr_destroy**(*pthread_barrierattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_barrierattr_init**(*pthread_barrierattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_barrierattr_getpshared**(const *pthread_barrierattr_t* *attr, int *pshared)

Not supported.

Returns
ENOSYS.

int **pthread_barrierattr_setpshared**(*pthread_barrierattr_t* *attr, int pshared)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_clockrdlock**(*pthread_rwlock_t* *rwlock, *clockid_t* clock_id, const struct *timespec* *abstime)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_timedrdlock**(*pthread_rwlock_t* *rwlock, const struct *timespec* *abstime)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_clockwrlock**(*pthread_rwlock_t* *rwlock, *clockid_t* clock_id, const struct *timespec* *abstime)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_timedwrlock**(*pthread_rwlock_t* *rwlock, const struct *timespec* *abstime)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_destroy**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_init**(*pthread_rwlock_t* *rwlock, const *pthread_rwlockattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_rdlock**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_tryrdlock**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_trywrlock**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_wrlock**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlock_unlock**(*pthread_rwlock_t* *rwlock)

Not supported.

Returns
ENOSYS.

int **pthread_rwlockattr_destroy**(*pthread_rwlockattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_rwlockattr_init**(*pthread_rwlockattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_rwlockattr_getpshared**(const *pthread_rwlockattr_t* *attr, int *pshared)

Not supported.

Returns
ENOSYS.

int **pthread_rwlockattr_setpshared**(*pthread_rwlockattr_t* *attr, int pshared)

Not supported.

Returns
ENOSYS.

int **pthread_cond_init**(*pthread_cond_t* *cond, const *pthread_condattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_cond_destroy**(*pthread_cond_t* *cond)

Not supported.

Returns
ENOSYS.

int **pthread_cond_wait**(*pthread_cond_t* *cond, *pthread_mutex_t* *mutex)

Not supported.

Returns
ENOSYS.

int **pthread_cond_timedwait**(*pthread_cond_t* *cond, *pthread_mutex_t* *mutex, const struct *timespec* *abstime)

Not supported.

Returns
ENOSYS.

int **pthread_cond_signal**(*pthread_cond_t* *cond)

Not supported.

Returns
ENOSYS.

int **pthread_cond_broadcast**(*pthread_cond_t* *cond)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_init**(*pthread_condattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_destroy**(*pthread_condattr_t* *attr)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_getclock**(const *pthread_condattr_t* *attr, *clockid_t* *clock_id)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_setclock**(*pthread_condattr_t* *attr, *clockid_t* clock_id)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_getpshared**(const *pthread_condattr_t* *attr, int *pshared)

Not supported.

Returns
ENOSYS.

int **pthread_condattr_setpshared**(*pthread_condattr_t* *attr, int pshared)

Not supported.

Returns
ENOSYS.

int **pthread_attr_settimeslice_np**(*pthread_attr_t* *attr, const struct *timespec* *time_slice)

Record the round-robin time slice that threads created from `attr` will run with.

The duration is converted to ticks at thread creation; only meaningful when the policy is *SCHED_RR*.

Error numbers:

- *EINVAL*: `attr` or `time_slice` is NULL, `time_slice` is not a valid timespec, or it is shorter than `QRT_CFG_SCHED_TIME_SLICE_MIN_US` (100 us by default).

Parameters

- **attr** – Pointer to the thread attribute object.
- **time_slice** – Pointer to a timespec structure that specifies the time slice duration.

Returns
0 on success, or an error number on failure.

int **pthread_attr_gettimeslice_np**(*pthread_attr_t* *attr, struct *timespec* *time_slice)

Read the round-robin time slice stored in `attr`.

Error numbers:

- *EINVAL*: `attr` or `time_slice` is NULL.

Parameters

- **attr** – Pointer to the thread attribute object.
- **time_slice** – Pointer to a timespec structure that specifies the time slice duration.

Returns
0 on success, or an error number on failure.

Limits

PTHREAD_STACK_MIN (512u)

Minimum size, in bytes, for a thread's stack.

Stack sizes passed to *pthread_attr_setstacksize()* or *pthread_attr_setstack()* must be at least this value.

8.1.3 Semaphore

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/semaphore.h.html

Defines

SEM_FAILED ((sem_t *)0)

Semaphore initialization failed.

SEM_VALUE_MAX (0x7FFFFFFF)

Maximum value a semaphore may hold.

Typedefs

typedef void ***sem_t**

Functions

int **sem_init**(*sem_t* *semaphore, int pshared, unsigned count)

Initialize an unnamed semaphore with the given initial count.

Initializes the semaphore with an initial count of `count`. When the MPU is enabled the semaphore state is protected from unprivileged threads. The `pshared` flag is accepted but ignored — semaphores are always process-private on this kernel.

Error numbers:

- *EINVAL* if `semaphore` is NULL or `count` exceeds *SEM_VALUE_MAX*.
- *ENOMEM* if the kernel heap has insufficient memory for the semaphore.

See also

sem_post() for example.

Parameters

- **semaphore** – Pointer to the semaphore.
- **pshared** – Ignored; semaphores are always process-private.
- **count** – Initial semaphore count, at most *SEM_VALUE_MAX*.

Returns
0 on success, -1 on error and errno set.

int **sem_destroy**(*sem_t* *semaphore)

Release the resources held by a semaphore.

The semaphore must have no waiters at the time of the call. The kernel object is freed back to the kernel heap and the caller's handle is cleared, so any subsequent use of `semaphore` fails with *EINVAL*.

Error numbers:

- *EINVAL* if the semaphore is invalid.
- *EBUSY* if one or more threads are blocked on the semaphore.

Parameters

- **semaphore** – Pointer to the semaphore.

Returns

int 0 on success, -1 on error and errno set.

int **sem_post**(*sem_t* *semaphore)

Increment a semaphore and wake one waiter, if any.

If the count is already at *SEM_VALUE_MAX* the call leaves the semaphore unchanged and reports an overflow. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL* if the semaphore is invalid.
- *EOVERFLOW* if the semaphore count is already at its maximum value.

Parameters

- **semaphore** – Pointer to the semaphore.

Returns

int 0 on success, -1 on error and errno set.

```
static sem_t        event_sem;
static unsigned int events;

static void event_deferred(qrt_u32_t parameter)
{
    (void)parameter;
    (void)sem_post(&event_sem);
}

void Device_IRQHandler(void)
{
    (void)qrt_defer_call(event_deferred, 0u);
}

void *worker(void *arg)
{
    (void)arg;
    if (sem_init(&event_sem, 0, 0u) == 0)
    {
        for (;;)
```

(continues on next page)

(continued from previous page)

```
        {
            (void)sem_wait(&event_sem);
            events++;
        }
    }
    return NULL;
}
```

int **sem_timedwait**(*sem_t* *semaphore, const struct *timespec* *abstime)

Decrement a semaphore, waiting until `abstime` on *CLOCK_REALTIME* if it is currently zero.

A thin wrapper around *sem_clockwait()* that fixes the clock to *CLOCK_REALTIME*.

Error numbers:

- *EINVAL* if the semaphore is invalid.
- *EINVAL* if the thread would have to wait and `abstime` has nanoseconds outside [0, 999'999'999].
- *ETIMEDOUT* if the thread would have to wait and the deadline has passed (negative seconds lie before the epoch), or elapses before the count becomes positive.

See also

sem_clockwait() for example.

Parameters

- **semaphore** – Pointer to the semaphore.
- **abstime** – Pointer to the absolute time to wait until.

Returns

0 on success, -1 on error and errno set.

int **sem_clockwait**(*sem_t* *semaphore, *clockid_t* clock_id, const struct *timespec* *abstime)

Decrement a semaphore, waiting until `abstime` on the chosen clock if it is currently zero.

If the count is positive it is decremented and the call returns immediately. Otherwise the caller blocks on the semaphore's wait list and is woken by *sem_post()* or when the deadline elapses. Every `abstime`, {0, 0} included, is an absolute time on `clock_id`; one with negative seconds lies before the epoch and has already passed. The call must be made from a thread, never from a deferred function or a timer callback.

Error numbers:

- *EINVAL* if the semaphore is invalid.
- *EINVAL* if the thread would have to wait and `clock_id` is not a supported clock.
- *EINVAL* if the thread would have to wait and `abstime` has nanoseconds outside [0, 999'999'999].
- *ETIMEDOUT* if the thread would have to wait and the deadline has passed (negative seconds lie before the epoch), or elapses before the count becomes positive.

Parameters

- **semaphore** – Pointer to the semaphore.
- **clock_id** – Clock ID to use for the timeout.

- **abstime** – Pointer to the absolute time to wait until.

Returns

0 on success, -1 on error and errno set.

```
static unsigned int events;
static unsigned int timeouts;

void *worker(void *arg)
{
    sem_t           *sem = (sem_t *)arg;
    struct timespec  deadline;

    for (;;)
    {
        (void)clock_gettime(CLOCK_MONOTONIC, &deadline);
        deadline.tv_sec += 1;

        if (sem_clockwait(sem, CLOCK_MONOTONIC, &deadline) == 0)
        {
            events++;
        }
        else if (errno == ETIMEDOUT)
        {
            timeouts++;
        }
    }
    return NULL;
}
```

int **sem_trywait**(*sem_t* *semaphore)

Decrement a semaphore if doing so will not block.

Never blocks: the call succeeds when the count is positive and reports *EAGAIN* otherwise. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*), which *sem_wait()*, *sem_timedwait()* and *sem_clockwait()* may not.

Error numbers:

- *EINVAL* if the semaphore is invalid.
- *EAGAIN* if the semaphore count is zero.

Parameters

- **semaphore** – Pointer to the semaphore.

Returns

0 on success, -1 on error and errno set.

int **sem_wait**(*sem_t* *semaphore)

Decrement a semaphore, waiting indefinitely if it is currently zero.

Equivalent to *sem_timedwait()* with a NULL deadline.

Error numbers:

- *EINVAL* if the semaphore is invalid.

See also

sem_post() for example.

Parameters

- **semaphore** – Pointer to the semaphore.

Returns

0 on success, -1 on error and errno set.

int **sem_getvalue**(*sem_t* *semaphore, int *sval)

Read a semaphore's current count into `sval`.

Stores the current count, or 0 when the semaphore is locked (held with no remaining permits). When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL* if the semaphore is invalid.

Parameters

- **semaphore** – Pointer to the semaphore.
- **sval** – Pointer to the location where the current value will be stored.

Returns

int 0 on success, -1 on error and errno set.

sem_t ***sem_open**(const char *name, int oflag, *mode_t* mode, unsigned int value)

Not supported.

Returns

Always *SEM_FAILED* and sets errno to *ENOSYS*.

int **sem_close**(*sem_t* *semaphore)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

int **sem_unlink**(const char *name)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

8.1.4 Message Queue

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/mqueue.h.html

Defines

MQ_PRIO_MAX (32u)

Priority levels; valid msg_prio is [0, MQ_PRIO_MAX-1].

Typedefs

typedef void ***mqd_t**

Message queue descriptor.

Functions

mqd_t **mq_open**(const char *name, int oflag, *mode_t* mode, struct *mq_attr* *attr)

Open an existing message queue or create a new one, returning a descriptor for subsequent send and receive operations.

Queues are looked up by name and allocated on demand from the kernel heap. A new queue uses a default attribute set (10 messages of 256 bytes) when `attr` is NULL. Storage for all `mq_maxmsg` messages is reserved when the queue is created, so a call with attributes within the ranges below can still fail with *ENOMEM*. A failed call allocates nothing.

Error numbers:

- *EINVAL* if the name is NULL.
- *EINVAL* if the oflag is invalid.
- *EINVAL* if *O_CREAT* is specified, `attr` is not NULL, and mq_maxmsg is outside [1, 255] or mq_msgsize is outside [1, 16777215].
- *ENAMETOOLONG* if *O_CREAT* is specified and the name is longer than 16 characters. Without *O_CREAT* such a name reports *ENOENT*.
- *EEXIST* if the queue exists and both *O_CREAT* and *O_EXCL* are specified.
- *ENOMEM* if there is insufficient memory for the descriptor, message queue, or name.
- *ENOENT* if the queue does not exist and *O_CREAT* is not specified.

Note

Name resolution is O(N) in the number of message queues that currently exist. This is a setup-time operation and is NOT part of the kernel's O(1) operational guarantee. The operational calls - *mq_send()*, *mq_receive()*, *mq_getattr()*, *mq_close()* - are O(1), independent of the number of queues or threads in the system.

Parameters

- **name** – Name of the message queue.
- **oflag** – One of the access mode flags:
  - *O_RDONLY* : Open the queue to receive messages only.
  - *O_WRONLY* : Open the queue to send messages only.
  - *O_RDWR* : Open the queue for reading and writing. And optionally OR'ed with:
  - *O_CREAT* : Create the queue if it does not exist.

  - *O_EXCL* : If *O_CREAT* is also specified, and a queue with the given name already exists, then *mq_open()* will fail.
  - *O_NONBLOCK* : Open this descriptor in non-blocking mode.
- **mode** – Not supported.
- **attr** – Attributes of a new queue; only `mq_maxmsg` and `mq_msgsize` are used. NULL for default attributes.

Returns

mqd_t Message queue descriptor on success, (*mqd_t*)-1 on error and errno set.

```
void *producer(void *arg)
{
    struct mq_attr attr  = { .mq_maxmsg  = 8,
                             .mq_msgsize = sizeof(uint32_t) };
    uint32_t       value = 0u;
    mqd_t          mq;

    (void)arg;
    mq = mq_open("/data", O_CREAT | O_WRONLY, 0, &attr);
    if (mq != (mqd_t)-1)
    {
        for (;;)
        {
            value++;
            (void)mq_send(mq, (const char *)&value, sizeof(value), 0u);
        }
    }
    return NULL;
}
```

int **mq_notify**(*mqd_t* mqdes, const struct *sigevent* *notification)

Not supported.

Returns

-1 and sets errno to *ENOSYS*.

int **mq_setattr**(*mqd_t* mqdes, const struct *mq_attr* *mqstat, struct *mq_attr* *omqstat)

Update the per-descriptor flags of an open message queue.

Only the *O_NONBLOCK* bit of `mq_flags` is honored; the other fields of `mqstat` are ignored. The change applies to `mqdes` only, not to other descriptors of the same queue. When `omqstat` is non-NULL it receives the attributes as *mq_getattr()* reported them before the update, so its `mq_flags` is the descriptor's previous *O_NONBLOCK* setting.

Error numbers:

- *EBADF* if the descriptor is invalid.
- *EINVAL* if `mqstat` is NULL.

Parameters

- **mqdes** – Message queue descriptor.
- **mqstat** – New message queue attributes.
- **omqstat** – Old message queue attributes.

Returns
0 on success, -1 on error and errno set.

int **mq_getattr**(*mqd_t* mqdes, struct *mq_attr* *mqstat)

Copy the attributes of an open message queue into `mqstat`.

`mq_maxmsg`, `mq_msgsize` and `mq_curmsgs` are those of the queue. `mq_flags` is that of the descriptor: *O_NONBLOCK* when `mqdes` is non-blocking (set by *mq_open()* or *mq_setattr()*) and 0 otherwise. No other flag is reported.

Error numbers:

- *EBADF* if the descriptor is invalid.
- *EINVAL* if `mqstat` is NULL.

Parameters

- **mqdes** – Message queue descriptor.
- **mqstat** – Message queue attributes.

Returns
0 on success, -1 on error and errno set.

int **mq_send**(*mqd_t* mqdes, const char *msg_ptr, *size_t* msg_len, unsigned msg_prio)

Append a message to a queue, waiting indefinitely if the queue is full and the descriptor is in blocking mode.

Equivalent to *mq_timedsend()* with a NULL deadline. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*), on a descriptor with *O_NONBLOCK* set.

Error numbers:

- *EAGAIN* if the queue is full and *O_NONBLOCK* is set.
- *EINVAL* if msg_ptr is NULL.
- *EINVAL* if msg_prio is not less than *MQ_PRIO_MAX*.
- *EMSGSIZE* if msg_len is 0 or exceeds the queue's message size.
- *EBADF* if the descriptor is invalid or not opened for writing, or the call is made from a deferred function or a timer callback without *O_NONBLOCK* set (a debug build asserts instead).

Parameters

- **mqdes** – Message queue descriptor.
- **msg_ptr** – Pointer to the message to be sent.
- **msg_len** – Length of the message.
- **msg_prio** – Message priority in [0, MQ_PRIO_MAX-1]. Higher-priority messages are received first; equal priorities stay FIFO.

Returns
0 on success, -1 on error and errno set.

int **mq_timedsend**(*mqd_t* mqdes, const char *msg_ptr, *size_t* msg_len, unsigned msg_prio, const struct *timespec* *abstime)

Append a message to a queue, waiting at most until `abstime` (*CLOCK_REALTIME*) when the queue is full.

If the descriptor has *O_NONBLOCK* set the call never blocks regardless of `abstime`. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*), on a descriptor with *O_NONBLOCK* set. This and *mq_send()* are the only message queue services available there.

Error numbers:

- *EAGAIN* if the queue is full and *O_NONBLOCK* is set.
- *EINVAL* if msg_ptr is NULL.
- *EINVAL* if msg_prio is not less than *MQ_PRIO_MAX*.
- *EINVAL* if the thread would have to wait and `abstime` has nanoseconds outside [0, 999'999'999].
- *EMSGSIZE* if msg_len is 0 or exceeds the queue's message size.
- *EBADF* if the descriptor is invalid or not opened for writing, or the call is made from a deferred function or a timer callback without *O_NONBLOCK* set (a debug build asserts instead).
- *ETIMEDOUT* if the queue is full and `abstime` has passed (negative seconds lie before the epoch, and {0, 0} is the epoch), or passes before a slot frees.

Parameters

- **mqdes** – Message queue descriptor.
- **msg_ptr** – Pointer to the message to be sent.
- **msg_len** – Length of the message.
- **msg_prio** – Message priority in [0, MQ_PRIO_MAX-1]. Higher-priority messages are received first; equal priorities stay FIFO.
- **abstime** – Pointer to the absolute time (*CLOCK_REALTIME*) to wait until when the queue is full.

Returns

0 on success, -1 on error and errno set.

ssize_t **mq_receive**(*mqd_t* mqdes, char *msg_ptr, *size_t* msg_len, unsigned *msg_prio)

Remove the highest-priority message (the oldest of equal priority) from a queue, waiting indefinitely if it is empty and the descriptor is in blocking mode.

Equivalent to *mq_timedreceive()* with a NULL deadline, so the call must likewise be made from a thread.

Error numbers:

- *EAGAIN* if the queue is empty and *O_NONBLOCK* is set.
- *EINVAL* if msg_ptr is NULL.
- *EMSGSIZE* if msg_len is smaller than the queue's message size.
- *EBADF* if the descriptor is invalid or not opened for reading.

Parameters

- **mqdes** – Message queue descriptor.
- **msg_ptr** – Pointer to the buffer that will receive the message.

- **msg_len** – Size of the receive buffer.
- **msg_prio** – If non-NULL, set to the priority of the received message.

Returns
Message length in bytes on success, -1 on error and errno set.

ssize_t **mq_timedreceive**(*mqd_t* mqdes, char *msg_ptr, *size_t* msg_len, unsigned *msg_prio, const struct *timespec* *abstime)

Remove the highest-priority message (the oldest of equal priority) from a queue, waiting at most until `abstime` (*CLOCK_REALTIME*) when the queue is empty.

If the descriptor has *O_NONBLOCK* set the call never blocks regardless of `abstime`. Unlike *mq_timedsend()*, the call must be made from a thread, never from a deferred function or a timer callback, even on a descriptor with *O_NONBLOCK* set.

Error numbers:

- *EAGAIN* if the queue is empty and *O_NONBLOCK* is set.
- *EINVAL* if msg_ptr is NULL.
- *EINVAL* if the thread would have to wait and `abstime` has nanoseconds outside [0, 999'999'999].
- *EMSGSIZE* if msg_len is smaller than the queue's message size.
- *EBADF* if the descriptor is invalid or not opened for reading.
- *ETIMEDOUT* if the queue is empty and `abstime` has passed (negative seconds lie before the epoch, and {0, 0} is the epoch), or passes before a message arrives.

Parameters

- **mqdes** – Message queue descriptor.
- **msg_ptr** – Pointer to the buffer that will receive the message.
- **msg_len** – Size of the receive buffer.
- **msg_prio** – If non-NULL, set to the priority of the received message.
- **abstime** – Pointer to the absolute time (*CLOCK_REALTIME*) to wait until when the queue is empty.

Returns
Message length in bytes on success, -1 on error and errno set.

```
static uint32_t     total;
static unsigned int timeouts;

void *consumer(void *arg)
{
    struct timespec deadline;
    uint32_t        value;
    mqd_t           mq;

    (void)arg;
    mq = mq_open("/data", O_RDONLY, 0, NULL);
    if (mq != (mqd_t)-1)
    {
        for (;;)
```

(continues on next page)

(continued from previous page)

```
        {
            (void)clock_gettime(CLOCK_REALTIME, &deadline);
            deadline.tv_sec += 1;

            if (mq_timedreceive(mq, (char *)&value, sizeof(value),
                                NULL, &deadline) > 0)
            {
                total += value;
            }
            else if (errno == ETIMEDOUT)
            {
                timeouts++;
            }
        }
    }
    return NULL;
}
```

int **mq_unlink**(const char *name)

Remove a message queue's name from the registry, deferring deletion until the last descriptor is closed.

If no descriptors remain at the time of the call the queue and its backing memory are released immediately; otherwise the queue is marked unlinked and destroyed when *mq_close()* drops the last reference. Until then the descriptors still open keep working as before, and the unlink does not change the attributes *mq_getattr()* reports for them.

Error numbers:

- *EINVAL* if `name` is NULL.
- *ENAMETOOLONG* if `name` is longer than 16 characters.
- *ENOENT* if the message queue does not exist.

Parameters

- **name** – Name of the message queue.

Returns

0 on success, -1 on error and errno set.

int **mq_close**(*mqd_t* mqdes)

Drop a descriptor and, if it was the last reference to an unlinked queue, release the queue itself.

Error numbers:

- *EBADF* if the descriptor is invalid.

Parameters

- **mqdes** – Message queue descriptor.

Returns

0 on success, -1 on error and errno set.

struct **mq_attr**

#include <mqueue.h> Message queue attributes.

Public Members

long **mq_flags**

Message queue flags.

long **mq_maxmsg**

Maximum number of messages.

long **mq_msgsize**

Maximum message size.

long **mq_curmsgs**

Number of messages currently queued.

8.1.5 Sched

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/sched.h.html

Part of QRT Real-Time Kernel.

Defines

SCHED_FIFO (0x0100u)

FIFO scheduling policy.

SCHED_RR (0x0200u)

Round-robin scheduling policy.

SCHED_SPORADIC (0x0400u)

Sporadic scheduling policy. Not supported.

SCHED_OTHER (0x0800u)

Other scheduling policy. Not supported.

Functions

int **sched_get_priority_max**(int policy)

Return the highest priority value accepted by the scheduler.

The value is independent of `policy`.

Parameters

- **policy** – Scheduling policy. Ignored.

Returns

Maximum priority value.

int **sched_get_priority_min**(int policy)

Return the lowest priority value usable by application threads.

The kernel's idle thread sits below this value, so it is never returned. The value is independent of `policy`.

Parameters

- **policy** – Scheduling policy. Ignored.

Returns

Minimum priority value.

int **sched_yield**(void)

Surrender the CPU to the next runnable thread of equal or higher priority.

If no such thread exists the call returns immediately and the caller keeps running.

Returns

Always 0.

int **sched_getparam**(*pid_t* pid, struct *sched_param* *param)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

int **sched_getscheduler**(*pid_t* pid)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

int **sched_rr_get_interval**(*pid_t* pid, struct *timespec* *interval)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

int **sched_setparam**(*pid_t* pid, const struct *sched_param* *param)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

int **sched_setscheduler**(*pid_t* pid, int policy, const struct *sched_param* *param)

Not supported.

Returns

Always -1 and sets errno to *ENOSYS*.

struct **sched_param**

#include <sched.h> Scheduling parameters for a thread.

Public Members

int **sched_priority**

Thread scheduling priority.

8.1.6 Time

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/time.h.html

Defines

CLOCK_REALTIME (0u)

Settable wall-clock (calendar) time.

At boot it reads the same as *CLOCK_MONOTONIC*, but it can be set to an arbitrary value with *clock_settime()* (when the *QRT_CFG_CLOCK_SETTIME_ENABLE* build option is enabled).

Because it can be set, this clock is **not** monotonic — setting it may move the time forwards or backwards. A forward jump immediately fires any pending absolute (*TIMER_ABSTIME*) sleep or timer that the new time has passed; a backward jump never fires one early. Relative sleeps and timers are never affected by setting the clock, even when created with `CLOCK_REALTIME`.

Use this clock for calendar timestamps and absolute scheduling that should follow *clock_settime()* changes.

CLOCK_MONOTONIC (1u)

Monotonic time since boot.

Counts elapsed time since the system started. It always advances at a steady rate, never jumps, and cannot be set — *clock_settime()* rejects it with *EINVAL* — so it is unaffected by any wall-clock change.

Use this clock to measure how much time has elapsed. Query its resolution with *clock_getres()*.

TIMER_ABSTIME (0x01)

Absolute time flag.

Functions

int **clock_getcpuclockid**(*pid_t* pid, *clockid_t* *clock_id)

Not supported.

Returns
ENOSYS.

int **clock_getres**(*clockid_t* clock_id, struct *timespec* *res)

Report the granularity of `clock_id` into `res`.

Both *CLOCK_MONOTONIC* and *CLOCK_REALTIME* report the same resolution, derived from the frequency of the System Timer.

Error numbers:

- *EINVAL* if `clock_id` is neither *CLOCK_MONOTONIC* nor *CLOCK_REALTIME*.

Parameters

- **clock_id** – Clock identifier.
- **res** – Pointer to a timespec where the resolution will be stored. Can be NULL, in which case the resolution is not stored.

Returns
0 on success, or -1 on error.

int **clock_gettime**(*clockid_t* clock_id, struct *timespec* *tp)

Read the current value of `clock_id` into `tp`.

CLOCK_MONOTONIC reports the time elapsed since boot. *CLOCK_REALTIME* reports the wall-clock time as last set by *clock_settime()*. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL* if `tp` is NULL or `clock_id` is not supported.

Parameters

- **clock_id** – Clock identifier.
- **tp** – Pointer to a timespec where the current time will be stored.

Returns

0 on success, or -1 on error.

int **clock_settime**(*clockid_t* clock_id, const struct *timespec* *tp)

Adjust *CLOCK_REALTIME* so that it currently reads `tp`.

Error numbers:

- *EINVAL* if `clock_id` is not *CLOCK_REALTIME*.
- *EINVAL* if `tp` is NULL or has a negative seconds value or nanoseconds outside [0, 999'999'999].
- *EINVAL* if `tp` is later than the clock can represent (2^62 - 1 System Timer ticks).

Parameters

- **clock_id** – Clock identifier.
- **tp** – Pointer to a timespec that specifies the new time for the specified clock.

Returns

0 on success, or -1 on error.

int **clock_nanosleep**(*clockid_t* clock_id, int flags, const struct *timespec* *rqtp, struct *timespec* *rmtp)

Block the calling thread for the duration specified by `rqtp`, measured on `clock_id`.

With *TIMER_ABSTIME*, `rqtp` is treated as a deadline on `clock_id`; otherwise it is treated as a relative interval. A deadline that has already passed, one with negative seconds (before the epoch) included, returns 0 at once without suspending the caller. Remaining time is not reported back: `rmtp` is accepted but ignored.

Error numbers:

- *EINVAL* if `rqtp` is NULL or has nanoseconds outside [0, 999'999'999].
- *EINVAL* if `rqtp` is a relative interval with negative seconds.
- *EINVAL* if `clock_id` is neither *CLOCK_MONOTONIC* nor *CLOCK_REALTIME*.

Parameters

- **clock_id** – Clock identifier.
- **flags** – Flags to control the behavior of the sleep operation.
  - *TIMER_ABSTIME* : the sleep time is interpreted as an absolute time.
- **rqtp** – Pointer to a timespec that specifies the requested sleep duration.

- **rmtp** – Not supported.

Returns

0 on success, or error number.

```
void *periodic_thread(void *arg)
{
    unsigned int    *periods = (unsigned int *)arg;
    struct timespec  next;

    (void)clock_gettime(CLOCK_MONOTONIC, &next);

    for (;;)
    {
        next.tv_nsec += 10000000L;
        if (next.tv_nsec >= 1000000000L)
        {
            next.tv_nsec -= 1000000000L;
            next.tv_sec++;
        }
        (void)clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                              &next, NULL);

        (*periods)++;
    }

    return NULL;
}
```

int **nanosleep**(const struct *timespec* *rqtp, struct *timespec* *rmtp)

Block the calling thread for the relative duration `rqtp`.

Equivalent to *clock_nanosleep()* against *CLOCK_REALTIME* with no flags. Remaining time is not reported back.

Error numbers:

- *EINVAL* if `rqtp` is NULL or has a negative seconds value or nanoseconds outside [0, 999'999'999].

Parameters

- **rqtp** – Pointer to a timespec that specifies the requested sleep duration.
- **rmtp** – Not supported.

Returns

0 on success, or -1 on error.

int **timer_create**(*clockid_t* clockid, struct *sigevent* *evp, *timer_t* *timerid)

Allocate a kernel timer and bind it to an expiration callback.

The timer is allocated from the privileged kernel heap and released by *timer_delete()*. Signal notification is not supported: `evp` must either request *SIGEV_NONE* (callback-less timer) or *SIGEV_THREAD* (invoke `sigev_notify_function` on expiry).

The callback does not run in a thread: the kernel calls it directly on expiry, in the same context as a deferred call, and no thread runs until it returns. It must not block. Of the services declared here it may call only *clock_gettime()*, *timer_gettime()* and *timer_settime()*, and only when *QRT_CFG_DEFER_ENABLE* is enabled.

Because the callback runs in kernel context, with MPU support only a privileged thread can request *SIGEV_THREAD*; an unprivileged caller gets *EPERM* and no timer is created. *SIGEV_NONE* timers need no privilege.

Note

errno:

- *EINVAL* if the clock ID is invalid.
- *EINVAL* if timerid is NULL.
- *ENOTSUP* if `evp` is NULL or specifies an unsupported notification type (e.g. *SIGEV_SIGNAL*).
- *EPERM* if `evp` requests *SIGEV_THREAD* and the caller is unprivileged (only with MPU support).
- *EAGAIN* if the limit on the number of existing timers is reached.
- *ENOMEM* if the kernel heap has insufficient memory for the timer.

```
static timer_t                timer;
static volatile unsigned int expirations;

static void on_expiry(union sigval value)
{
    (void)value;
    expirations++;
}

int start_periodic_timer(void)
{
    struct sigevent   sev = { .sigev_notify          = SIGEV_THREAD,
                              .sigev_notify_function = on_expiry };
    struct itimerspec its = { .it_interval = { 0, 10000000L },
                              .it_value    = { 0, 10000000L } };
    int               status;

    status = timer_create(CLOCK_MONOTONIC, &sev, &timer);
    if (status == 0)
    {
        status = timer_settime(timer, 0, &its, NULL);
    }

    return status;
}
```

Parameters

- **clockid** – Clock identifier, *CLOCK_MONOTONIC* or *CLOCK_REALTIME*.
- **evp** – Pointer to a sigevent that specifies the event to be delivered when the timer expires.
- **timerid** – Pointer to a `timer_t` where the created timer ID will be stored.

Returns

0 on success, -1 on error.

int **timer_delete**(*timer_t* timerid)

Disarm a timer and return its memory to the kernel heap.

Error numbers:

- *EINVAL* if `timerid` does not refer to a currently allocated timer.

Parameters

- **timerid** – Timer ID to be deleted.

Returns

0 on success, -1 on error.

int **timer_getoverrun**(*timer_t* timerid)

Not supported.

Returns

-1 and sets errno to *ENOSYS*.

int **timer_gettime**(*timer_t* timerid, struct *itimerspec* *value)

Report the time until the next expiration and the configured reload interval of `timerid`.

A disarmed timer reports zero in `it_value`. A non-periodic timer reports zero in `it_interval`. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL* if `timerid` does not refer to a currently allocated timer, or `value` is NULL.

Parameters

- **timerid** – Timer ID.
- **value** – Pointer to a itimerspec where the remaining time will be stored.

Returns

0 on success, -1 on error and errno set.

int **timer_settime**(*timer_t* timerid, int flags, const struct *itimerspec* *value, struct *itimerspec* *ovalue)

Arm, re-arm, or disarm `timerid`.

The new configuration in `value` is applied as follows:

- `it_value` zero disarms the timer, with or without *TIMER_ABSTIME*, whatever `it_interval` holds. `it_interval` still sets the reload interval that *timer_gettime()* reports, zero when it is malformed.
- Non-zero `it_value` arms a one-shot timer when `it_interval` is zero, or a periodic timer otherwise.
- With *TIMER_ABSTIME*, `it_value` is treated as a deadline on the clock chosen at *timer_create()* time; otherwise it is relative. A deadline that has already passed, one with negative seconds (before the epoch) included, expires at once.

An expiration later than the clock can represent (2^62 - 1 System Timer ticks) is held at that limit, and a periodic timer whose next expiration reaches or passes it is disarmed after its last expiration.

The previous configuration is written to `ovalue` when it is non-NULL. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Error numbers:

- *EINVAL* if `timerid` does not refer to a currently allocated timer, or `value` is NULL.

- *EINVAL* if `value` has a non-zero `it_value` and contains a malformed timespec (nanoseconds outside [0, 999'999'999], or negative seconds in `it_interval` or in a relative `it_value`).

Parameters

- **timerid** – Timer ID.
- **flags** – Flags to control the behavior of the timer set operation.
  - *TIMER_ABSTIME* : the time is interpreted as an absolute time.
- **value** – Pointer to a itimerspec that specifies the new timer settings.
- **ovalue** – Pointer to a itimerspec where the old timer settings will be stored. Can be NULL if not needed.

Returns

0 on success, -1 on error and errno set.

```
int timeout_restart(timer_t timer)
{
    struct itimerspec one_shot = { .it_value = { 0, 100000000L } };

    return timer_settime(timer, 0, &one_shot, NULL);
}

int timeout_cancel(timer_t timer)
{
    struct itimerspec disarm = { .it_value = { 0, 0 } };

    return timer_settime(timer, 0, &disarm, NULL);
}
```

struct **timespec**

#include <time.h> Time value expressed as seconds plus nanoseconds.

Public Members

time_t **tv_sec**

Seconds.

long **tv_nsec**

Nanoseconds. [0, 999'999'999]

struct **itimerspec**

#include <time.h> Timer setting: initial expiration and reload period.

Public Members

struct *timespec* **it_interval**

Timer period.

struct *timespec* **it_value**

Timer expiration.

8.1.7 Signal

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/signal.h.html

Part of QRT Real-Time Kernel.

Defines

SIGEV_NONE (0)

No event notification.

SIGEV_SIGNAL (1)

Signal notification. Not supported.

SIGEV_THREAD (2)

Callback notification: `sigev_notify_function` is called with `sigev_value` in kernel context, not in a thread.

With MPU support only privileged threads can request it; see *timer_create()*.

Typedefs

typedef void ***sigset_t**

Not supported.

union **sigval**

#include <signal.h> Value passed to a notification callback on event delivery.

Public Members

int **sival_int**

Integer value.

void ***sival_ptr**

Pointer value.

struct **sigevent**

#include <signal.h> Notification request describing how an asynchronous event (such as a timer expiration) is delivered.

Public Members

int **sigev_notify**

SIGEV value.

union *sigval* **sigev_value**

Callback value.

void (***sigev_notify_function**)(union *sigval* value)

Callback.

void ***sigev_notify_attributes**

Not supported.

8.1.8 Unix Standard

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/unistd.h.html

Functions

ssize_t **write**(int fildes, const void *buf, *size_t* nbyte)

Not supported.

ssize_t **read**(int fildes, void *buf, *size_t* nbyte)

Not supported.

int **close**(int fildes)

Not supported.

unsigned **sleep**(unsigned seconds)

Block the calling thread for at least `seconds` whole seconds.

Implemented as *nanosleep* with `tv_sec` set to `seconds` and `tv_nsec` set to 0.

Parameters

- **seconds** – Number of seconds to sleep.

Returns

0.

8.1.9 File Control

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/fcntl.h.html

Part of QRT Real-Time Kernel.

Defines

O_RDONLY

Open for read only.

O_WRONLY

Open for write only.

O_RDWR

Open for read/write.

O_NONBLOCK

Non-blocking mode.

O_CREAT

Create file if not exists.

O_EXCL

Fail if the file already exists. Only meaningful with *O_CREAT*.

8.1.10 Types

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/sys_types.h.html

Typedefs

typedef int **clock_t**

typedef unsigned char **clockid_t**

typedef int **mode_t**

typedef int **pid_t**

typedef int **pthread_t**

typedef int **pthread_key_t**

typedef int **pthread_once_t**

typedef void ***pthread_cond_t**

typedef void ***pthread_condattr_t**

typedef void ***pthread_rwlock_t**

typedef void ***pthread_rwlockattr_t**

typedef void ***pthread_barrier_t**

typedef void ***pthread_barrierattr_t**

typedef void ***pthread_spinlock_t**

typedef void ***pthread_mutex_t**

typedef int **dev_t**

typedef int **ino_t**

typedef unsigned short **nlink_t**

typedef unsigned short **uid_t**

typedef unsigned short **gid_t**

typedef unsigned long **blksize_t**

typedef unsigned long **blkcnt_t**

typedef unsigned **size_t**

typedef int **ssize_t**

typedef unsigned **useconds_t**

typedef long long **time_t**

typedef void ***timer_t**

struct **pthread_attr_t**

#include <types.h> Thread creation attribute object.

Opaque storage; initialize with *pthread_attr_init()* and access it only through the pthread_attr_*() functions.

struct **pthread_mutexattr_t**

8.1.11 Error Number

https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/errno.h.html

Defines

EACCES (1)

Permission denied.

EADDRINUSE (2)

Address in use.

EADDRNOTAVAIL (3)

Address not available.

EAFNOSUPPORT (4)

Address family not supported.

EAGAIN (5)

Resource unavailable, try again (may be the same value as [EWOULDBLOCK]).

EALREADY (6)

Connection already in progress.

EBADF (7)

Bad file descriptor.

EBADMSG (8)

Bad message.

EBUSY (9)

Device or resource busy.

ECANCELED (10)

Operation canceled.

ECHILD (11)

No child processes.

ECONNABORTED (12)

Connection aborted.

ECONNREFUSED (13)

Connection refused.

ECONNRESET (14)

Connection reset.

EDEADLK (15)

Resource deadlock would occur.

EDESTADDRREQ (16)

Destination address required.

EDOM (17)

Mathematics argument out of domain of function.

EDQUOT (18)

Reserved.

EEXIST (19)
File exists.

EFAULT (20)
Bad address.

EFBIG (21)
File too large.

EHOSTUNREACH (22)
Host is unreachable.

EIDRM (23)
Identifier removed.

EILSEQ (24)
Illegal byte sequence.

EINPROGRESS (25)
Operation in progress.

EINTR (26)
Interrupted function.

EINVAL (27)
Invalid argument.

EIO (28)
I/O error.

EISCONN (29)
Socket is connected.

EISDIR (30)
Is a directory.

ELOOP (31)
Too many levels of symbolic links.

EMFILE (32)
File descriptor value too large.

EMLINK (33)
Too many hard links.

EMSGSIZE (34)

Message too large.

EMULTIHOP (35)

Reserved.

ENAMETOOLONG (36)

Filename too long.

ENETDOWN (37)

Network is down.

ENETRESET (38)

Connection aborted by network.

ENETUNREACH (39)

Network unreachable.

ENFILE (40)

Too many files open in system.

ENOBUFS (41)

No buffer space available.

ENODEV (42)

No such device.

ENOENT (43)

No such file or directory.

ENOEXEC (44)

Executable file format error.

ENOLCK (45)

No locks available.

ENOLINK (46)

Reserved.

ENOMEM (47)

Not enough space.

ENOMSG (48)

No message of the desired type.

ENOPROTOOPT (49)

Protocol not available.

ENOSPC (50)

No space left on device.

ENOSYS (51)

Functionality not supported.

ENOTCONN (52)

The socket is not connected.

ENOTDIR (53)

Not a directory or a symbolic link to a directory.

ENOTEMPTY (54)

Directory not empty.

ENOTRECOVERABLE (55)

State not recoverable.

ENOTSOCK (56)

Not a socket.

ENOTSUP (57)

Not supported (may be the same value as [EOPNOTSUPP]).

ENOTTY (58)

Inappropriate I/O control operation.

ENXIO (59)

No such device or address.

EOPNOTSUPP (60)

Operation not supported on socket (may be the same value as [ENOTSUP]).

EOVERFLOW (61)

Value too large to be stored in data type.

EOWNERDEAD (62)

Previous owner died.

EPERM (63)

Operation not permitted.

EPIPE (64)

Broken pipe.

EPROTO (65)

Protocol error.

EPROTONOSUPPORT (66)

Protocol not supported.

EPROTOTYPE (67)

Protocol wrong type for socket.

ERANGE (68)

Result too large.

EROFS (69)

Read-only file system.

ESOCKTNOSUPPORT (70)

Socket type not supported.

ESPIPE (71)

Invalid seek.

ESRCH (72)

No such process.

ESTALE (73)

Reserved.

ETIMEDOUT (74)

Connection timed out.

ETXTBSY (75)

Text file busy.

EWOULDBLOCK (76)

Operation would block (may be the same value as [EAGAIN]).

EXDEV (77)

Improper hard link.

E2BIG (78)

Argument list too long.

errno (*__errno_location())

Get pointer to errno variable.

Functions

int ***__errno_location**(void)

Return the address of the calling thread's errno.

errno is thread-local: the pointer resolves through the current thread's TLS block. With QRT_CFG_ERRNO_ENABLE off the function returns NULL.

Returns

Pointer to the thread's errno variable.

8.2 QRT

This section provides a comprehensive reference for all QRT API functions, data types, and constants.

8.2.1 Kernel

Main header file for the QRT kernel.

This file contains the main declarations for the QRT kernel. It includes function prototypes for initializing and starting the kernel, MPU configuration, as well as type definitions for integer types.

Typedefs

typedef unsigned long long int **qrt_u64_t**

typedef unsigned int **qrt_u32_t**

typedef unsigned short **qrt_u16_t**

typedef unsigned char **qrt_u8_t**

typedef signed long long int **qrt_s64_t**

typedef signed int **qrt_s32_t**

typedef signed short **qrt_s16_t**

typedef signed char **qrt_s8_t**

typedef unsigned long int **qrt_uptr_t**

typedef signed long int **qrt_sptr_t**

typedef *qrt_s32_t* (***qrt_syscall_ext_t**)(void *args)

Functions

qrt_s32_t **qrt_syscall_register**(*qrt_s32_t* (*syscall)(void *args), *qrt_u32_t* *id)

Register a user-defined function as a System Call Extension and return its function ID via `id`.

At most *QRT_CFG_SYSCALL_EXT_LIMIT* extensions can be registered. Only available when MPU support and the syscall-extension feature are both compiled in; callers must be privileged.

The extension runs in privileged (kernel) context and its single argument is a pointer to the caller's saved-register frame, not the `param` value handed to *qrt_syscall_invoke*. That value occupies the first slot of the frame, so an extension reads it as

```
qrt_s32_t extension(void *args)
{
    qrt_uptr_t param = ((qrt_uptr_t *)args)[0u];

    return (qrt_s32_t)param;
}
```

The frame pointer is never NULL, even when the caller passed NULL. Whatever the caller put in the slot – an address or a plain integer – arrives unchanged and unchecked: the kernel neither dereferences nor validates it, so an extension must treat it as untrusted unprivileged input.

Error numbers:

- *EPERM* if the caller is not running with privileged access.
- *EINVAL* if `syscall` or `id` is NULL.
- *ENOSPC* if every extension slot is already occupied.

Parameters

- **syscall** – Pointer to the System Call Extension.
- **id** – Pointer to store the assigned function ID.

Returns

0 if success, error number otherwise. The example below registers an extension from the privileged startup thread; an unprivileged worker then invokes it with an index in the parameter slot.

```
#define DEVICE_BASE (0x40010000u)
#define DEVICE_REGS (4u)

static qrt_u32_t read_id;
static qrt_s32_t value;

qrt_s32_t read_extension(void *args)
{
    qrt_uptr_t index = ((qrt_uptr_t *)args)[0u];

    if (index >= DEVICE_REGS)
    {
        return -1;
    }
    return ((volatile qrt_s32_t *)DEVICE_BASE)[index];
}
```

(continues on next page)

(continued from previous page)

```
void *unprivileged_worker(void *arg)
{
    (void)arg;
    value = qrt_syscall_invoke(read_id, (void *)2u);
    return NULL;
}

void *startup_thread(void *arg)
{
    pthread_t worker_id;

    (void)arg;
    if (qrt_syscall_register(read_extension, &read_id) == 0)
    {
        (void)pthread_create(&worker_id, NULL,
                             unprivileged_worker, NULL);
    }
    return NULL;
}
```

qrt_s32_t **qrt_syscall_invoke**(*qrt_u32_t* f, void *param)

Invoke an extension previously installed with *qrt_syscall_register* through the kernel's syscall path.

The extension runs in privileged context. `param` travels unchanged in the first argument register of the system call; it may be an address or an integer cast to a pointer, and the kernel never dereferences it. The extension does not receive `param` itself but a pointer to the register frame that holds it – see *qrt_syscall_register* for how an extension reads it. The return value is whatever the extension returns.

See also

qrt_syscall_register for the use of System Call Extensions.

Parameters

- **f** – Function ID returned by *qrt_syscall_register*.
- **param** – Parameter for the extension: an address, or an integer cast to a pointer. May be NULL.

Returns

Return value from the extension, or *ENOSYS* if `f` names no registered extension (a debug build asserts instead).

qrt_s32_t **qrt_perf_read**(char *buf, *qrt_u32_t* size)

Format the worst-case execution time (WCET) results into a caller-provided buffer.

Provided only when *QRT_CFG_PERF_ENABLE* is 1.

One "`<service> <cycles>\n`" line per exercised call, then `context_switch`, `irq_mask` (worst-case interrupt masking) and `systimer_handling` (worst-case System Timer handling) lines in the same format; NUL-terminated when `size` is not 0; stops when the table is exhausted or `size` is reached.

Parameters

- **buf** – Destination buffer, in the caller's own memory.
- **size** – Buffer size in bytes.

Returns

Number of bytes written (excluding the terminating NUL).

```
void perf_report(void)
{
    static char report[1024];

    (void)qrt_perf_read(report, (qrt_u32_t)sizeof(report));
    (void)printf("%s", report);
}
```

qrt_s32_t **qrt_kernel_boot**(*pthread_t* *thread, const *pthread_attr_t* *attr, void *(*start_routine)(void *arg), void *arg)

Bring the kernel up and hand control to a first thread.

Initializes the kernel and starts scheduling. On success the calling context becomes the main thread and runs `start_routine`; the function does not return.

Error numbers:

- *ENOMEM* Not enough memory to start the kernel, increase the kernel heap size *QRT_CFG_KERNEL_HEAP_SIZE* or increase the thread stack pool size *QRT_CFG_STACK_POOL_SIZE*.
- *EINVAL* if System Timer is not configured.
- *EINVAL* if the MPU is enabled and a memory region defined in the linker script is not aligned to its size (ARMv6 and ARMv7) or to 32 bytes (ARMv8).
- *EINVAL* if `start_routine` is NULL or the thread attributes in `attr` are invalid.
- *EIO* if System Timer is stuck (its count does not advance).
- *EBUSY* if the kernel is already running.

Note

This function must be called after all necessary system initialization is complete. Once the kernel is started, further modifications to system initialization should not be attempted.

```
#include <pthread.h>
#include "bsp.h"

#define TIMER_MAX_PERIOD (0xFFFFFFu)
#define TIMER_HZ         (48000000)

void *worker(void *arg);

void *startup_thread(void *arg)
{
    pthread_t worker_id;

    (void)arg;
    (void)pthread_create(&worker_id, NULL, worker, NULL);
    return NULL;
```

```
}

int main(void)
{
    bsp_init();

    if (qrt_systimer_configure(bsp_systickreload, bsp_systickgetcount,
                               TIMER_MAX_PERIOD, TIMER_HZ,
                               BSP_SYSTICK_IRQN) != 0)
    {
        return -1;
    }
    return qrt_kernel_boot(NULL, NULL, startup_thread, NULL);
}
```

Parameters

- **thread** – Pointer to the thread handle. NULL if not needed.
- **attr** – Pointer to the thread attributes. NULL for default attributes.
- **start_routine** – Pointer to the function to be executed by the thread.
- **arg** – Argument to be passed to the thread function.

Returns

0 if success, error number otherwise.

qrt_s32_t **qrt_kernel_idlehookregister**(void (*hook_function)(void))

Install (or clear) the function that runs whenever the kernel has no other runnable thread.

The hook runs in the idle thread whenever no other thread is runnable; when idle sleep is enabled (*QRT_CFG_IDLE_SLEEP_ENABLE* / *QRT_CFG_IDLE_DEEP_SLEEP_ENABLE*) the CPU enters sleep only after the hook returns. Pass NULL to remove the currently installed hook. Requires the caller to be privileged when the MPU is enabled.

Note

The idle hook function is called from privileged context which exposes an attack vector if untrusted code is executed. It is users responsibility to ensure the hook function does not compromise system security.

Note

System Calls from idle hook lead to undefined behavior.

Parameters

- **hook_function** – Pointer to the idle hook function. NULL pointer to unregister the idle hook.

Returns

Returns 0 on success, *EPERM* if called from unprivileged thread when MPU is enabled, or *ENOSYS* if *QRT_CFG_KERNEL_IDLE_HOOK_REGISTER_ENABLE* is 0.

qrt_s32_t **qrt_systimer_configure**(void (*reload_ptr)(*qrt_u64_t* period), *qrt_u64_t* (*get_count_ptr)(void), *qrt_u64_t* max_period, *qrt_s32_t* frequency_hz, *qrt_s32_t* irqn)

Bind the kernel's SysTimer to a hardware timer described by the supplied accessors. Must be called before *qrt_kernel_boot*.

The kernel uses `reload_ptr` to program the next deadline and `get_count_ptr` to sample the current count. The supplied `irqn` is configured and enabled in the interrupt controller.

Error numbers:

- *EINVAL* if `reload_ptr` or `get_count_ptr` is NULL.
- *EBUSY* if the kernel is already running.

Parameters

- **reload_ptr** – Pointer to the function to reload the timer.
- **get_count_ptr** – Pointer to the function to get the current timer count.
- **max_period** – Maximum timer period in ticks.
- **frequency_hz** – Timer frequency in Hz.
- **irqn** – Interrupt request number for the timer.

Returns

Returns 0 on success, or an error number on failure.

qrt_s32_t **qrt_systimer_getmonotonic**(*qrt_s64_t* *ticks)

Sample the kernel's monotonic tick counter.

The reading is the SysTimer's accumulated tick count since the timer was configured at boot. It is unaffected by *clock_settime()*. A call made before *qrt_kernel_boot* has started the kernel fails with *EINVAL* and leaves `ticks` unchanged. When *QRT_CFG_DEFER_ENABLE* is enabled the call may also be made from a deferred function or a timer callback (see *qrt_defer_call()*).

Parameters

- **ticks** – Pointer to store the current monotonic tick count.

Returns

0 on success, *EINVAL* if the kernel has not been started with *qrt_kernel_boot* or the System Timer has not been configured with *qrt_systimer_configure*, or *ENOSYS* if *QRT_CFG_SYSTIMER_GET_MONOTONIC_ENABLE* is 0.

void **qrt_systimer_handler**(void)

Service the SysTimer deadline interrupt and hand control to the scheduler.

QRT schedules to event deadlines rather than fixed periodic ticks: the SysTimer is armed to fire when the next scheduling decision is due. The application's hardware timer ISR must invoke this handler when that interrupt fires; it reprograms the timer for the following deadline. Calling it from any other context is undefined behavior.

```
void SysTick_Handler(void)
{
    qrt_systimer_handler();
}
```

qrt_s32_t **qrt_defer_call**(void (*deferred_function)(*qrt_u32_t* parameter), *qrt_u32_t* parameter)

Queue `deferred_function` to run in kernel context at the next dispatch opportunity, typically right after the ISR returns.

The canonical way to escape from an ISR before issuing any kernel call: hardware interrupt service routines may not invoke kernel services directly. The deferred function does not run in a thread, no thread runs until it returns, and it must not block. It may call only these kernel services:

- *pthread_cancel()* and *pthread_setschedprio()*.
- *sem_post()*, *sem_trywait()* and *sem_getvalue()*.
- *mq_send()* and *mq_timedsend()*, on a descriptor with *O_NONBLOCK* set.
- *clock_gettime()*, *timer_gettime()* and *timer_settime()*.
- *qrt_systimer_getmonotonic()*.

Deferred calls require *QRT_CFG_DEFER_ENABLE*, and *QRT_CFG_DEFER_LIMIT* sets how many calls the buffer holds. A timer callback (see *timer_create()*) runs in the same context: it may call the same services, and only when *QRT_CFG_DEFER_ENABLE* is enabled.

Error numbers:

- *ENOSPC* Deferred call buffer is full.
- *EINVAL* Invalid pointer to deferred function.
- *ENOSYS* Deferred calls are disabled (*QRT_CFG_DEFER_ENABLE* is 0).

Parameters

- **deferred_function** – Pointer to the deferred function.
- **parameter** – Parameter to the deferred function.

Returns

0 on success, error number otherwise.

```
static sem_t     event_sem;
static qrt_u32_t events;

void event_deferred(qrt_u32_t param)
{
    (void)param;
    (void)sem_post(&event_sem);
}

void Device_IRQHandler(void)
{
    (void)qrt_defer_call(event_deferred, 0u);
}

void *event_thread(void *arg)
{
    (void)arg;
    (void)sem_init(&event_sem, 0, 0u);
    for (;;)
    {
        (void)sem_wait(&event_sem);
```

(continues on next page)

(continued from previous page)

```
        events++;
    }
    return NULL;
}
```

8.2.2 MPU

Definitions for ARMv6, ARMv7 and ARMv8 MPU configuration.

This file contains definitions and macros for configuring the Memory Protection Unit (MPU) on ARM cores. It includes definitions for execution permissions, access permissions, memory attributes, and memory types.

Defines

QRT_MPU_EXECUTE (0u)

Execute

QRT_MPU_EXECUTE_NEVER (1u)

Execute never

QRT_MPU_READWRITE (0u)

Read/write access for privileged and unprivileged code

QRT_MPU_READONLY (1u)

Read-only access for privileged and unprivileged code

QRT_MPU_DEVICE_NGNRNE (0u)

QRT_MPU_DEVICE_NGNRE (1u)

QRT_MPU_DEVICE_GRE (2u)

Not supported on ARMv6/7

QRT_MPU_STRONGLY_ORDERED (QRT_MPU_DEVICE_NGNRNE)

ARMv6/7 compatibility

QRT_MPU_DEVICE (QRT_MPU_DEVICE_NGNRE)

ARMv6/7 compatibility

QRT_MPU_NORM_NCACHE (3u)

QRT_MPU_NORM_WB_WRA (4u)

Outer & inner cacheable, write-back, write allocate

QRT_MPU_NORM_WT_NWA (5u)

Outer & inner cacheable, write-through, no write allocate

QRT_MPU_NORM_WB_NWA (6u)

Outer & inner cacheable, write-back, no write allocate

QRT_MPU_NONSHAREABLE (0x00u)

Non-shareable

QRT_MPU_INNER_SHAREABLE (0x01u)

Inner Shareable (not supported on ARMv6/7)

QRT_MPU_OUTER_SHAREABLE (0x02u)

Outer Shareable (Inner + Outer Shareable for ARMv6/7)

QRT_MPU_SHAREABLE (*QRT_MPU_OUTER_SHAREABLE*)

Inner + Outer Shareable

QRT_MPU_SUBREGION_DISABLE(mask) (((mask) & 0xFFu) << 8)

Not supported on ARMv8

Functions

qrt_s32_t **qrt_mpu_threadregionadd**(*pthread_t* thread, *qrt_s32_t* *region_id, *qrt_u32_t* executability, *qrt_u32_t* permissions, *qrt_u32_t* shareability, *qrt_u32_t* attributes, *qrt_u32_t* address, *qrt_u32_t* size)

Add a memory region to threads region table.

The region size is rounded up, not rejected: on ARMv6 and ARMv7 to the next power of two and to at least the minimum size (256 bytes for ARMv6, 32 bytes for ARMv7); on ARMv8 the end of the region is rounded up to the next 32-byte boundary.

Error numbers:

- *EINVAL* Region base address is not aligned to the rounded-up region size (ARMv6 and ARMv7) or is not 32-byte aligned (ARMv8).
- *EINVAL* `executability`, `permissions`, `shareability` or `attributes` is not one of the values supported on the core.
- *ENOSPC* No more MPU regions available.
- *EPERM* Called from unprivileged thread.
- *ENOSYS* MPU support is disabled (*QRT_CFG_MPU_ENABLE* is 0).

Parameters

- **thread** – Thread handle; 0 selects the calling thread.
- **region_id** – Pointer to assigned region ID. NULL if not needed.
- **executability** – Executability of the region.
- **permissions** – Permissions of the region: *QRT_MPU_READWRITE* or *QRT_MPU_READONLY*.
- **shareability** – Shareability of the region.
- **attributes** – Attributes of the region.

- **address** – Base address of the region, aligned as described above.
- **size** – Size of the region in bytes.

Returns

0 if success, error number otherwise.

```
#define DEVICE_BASE (0x40010000u)
#define DEVICE_SIZE (0x100u)

void *driver_thread(void *arg);

qrt_s32_t start_driver(void)
{
    pthread_t driver;
    qrt_s32_t status;

    status = pthread_create(&driver, NULL, driver_thread, NULL);
    if (status == 0)
    {
        status = qrt_mpu_threadregionadd(driver, NULL,
                                         QRT_MPU_EXECUTE_NEVER,
                                         QRT_MPU_READWRITE,
                                         QRT_MPU_SHAREABLE,
                                         QRT_MPU_DEVICE,
                                         DEVICE_BASE, DEVICE_SIZE);
    }
    return status;
}
```

qrt_s32_t **qrt_mpu_threadregionremove**(*pthread_t* thread, *qrt_s32_t* region_number)

Remove a memory region from a thread's region table. The calling thread must be privileged and the following rule applies: (5 < region_number < *QRT_CFG_MPU_ENTRIES*).

Parameters

- **thread** – Thread handle.
- **region_number** – Region number to remove.

Returns

0 if success *EINVAL* if 5 < region_number < *QRT_CFG_MPU_ENTRIES* not fulfilled. *EPERM* if called from unprivileged thread. *ENOSYS* if *QRT_CFG_MPU_ENABLE* is 0.

qrt_s32_t **qrt_mpu_faultcallbackset**(void (*callback)(*pthread_t* thread, *qrt_u32_t* address, *qrt_u32_t* pc, *qrt_u32_t* status))

Set an optional callback function that is invoked when a Memory Protection Unit (MPU) fault occurs. The registered callback allows the application to handle MPU faults, such as invalid memory accesses.

The registered callback will be called in the fault handler context, so it must be minimal and must not make any System Calls. ARMv7 and ARMv8 Mainline only. Only privileged threads can set the callback; an unprivileged caller gets *EPERM* and the callback already set stays in place.

Error numbers:

- *EINVAL* Callback function pointer is NULL.
- *EPERM* Called from unprivileged thread.

- *ENOSYS* MPU support is disabled (*QRT_CFG_MPU_ENABLE* is 0), or the core is ARMv6 or ARMv8 Baseline.

Parameters

- **callback** – Pointer to the callback function. The function must match the signature void (*callback)(

  pthread_t thread,

  qrt_u32_t address,

  qrt_u32_t pc,

  qrt_u32_t status).

Returns

0 if success, error number otherwise.

```
static volatile pthread_t fault_thread;
static volatile qrt_u32_t fault_address;

void mpu_fault(pthread_t thread, qrt_u32_t address,
               qrt_u32_t pc, qrt_u32_t status)
{
    (void)pc;
    (void)status;
    fault_thread  = thread;
    fault_address = address;
}

void *startup_thread(void *arg)
{
    (void)arg;
    (void)qrt_mpu_faultcallbackset(mpu_fault);
    return NULL;
}
```

void **qrt_mpu_faulthandler**(void)

MPU fault handler called when a Memory Protection Unit (MPU) fault occurs.

Must be called from the Memory Management interrupt handler.

```
void MemManage_Handler(void)
{
    qrt_mpu_faulthandler();
}
```

CHAPTER

NINE

CHANGELOG

9.1 Version History

Table 1: Software Version History

Software Version	Revision	Date	Notes
1.2.1	4	Oct 10, 2026	Changed: • *QRT_CFG_SCHED_TIME_SLICE_MIN_US*, the shortest round-robin time slice, is required in qrt_cfg.h; *pthread_attr_settimeslice_np()* and *pthread_create()* reject shorter slices • a preempted thread resumes before the ready threads of its priority and finishes its round-robin time slice; *pthread_setschedparam()* at an unchanged priority yields to equal-priority ready threads • a {0, 0} absolute time is an ordinary past deadline: timed waits return *ETIMEDOUT* instead of *EAGAIN* or *EBUSY*; an absolute time before the epoch has passed instead of reporting *EINVAL* • *qrt_kernel_boot()* and *qrt_systimer_configure()* return *EBUSY* once the kernel is running • the resources of a detached thread are released one scheduling pass after it exits • MPU builds: *EPERM* for unprivileged callers that would have the kernel run their code privileged

continues on next page

Table 1 – continued from previous page

Software Version	Revision	Date	Notes
			Fixed: • a priority change after an earlier lowering takes effect and is reported by *pthread_getschedparam()* • a mutex owner boosted by priority inheritance returns to its own priority on unlock, also after nested boosts, and the unlock yields to a higher-priority ready thread • *pthread_setschedprio()* and *pthread_setschedparam()* at a thread's own priority keep an active priority-inheritance boost • a thread releasing its last mutex a second time keeps its priority instead of dropping to 0 • a priority change of a waiting thread, of a timed-out thread that has not run yet or from a timer callback takes effect without corrupting the wait • a round-robin time slice is armed at dispatch and refilled after a timed wait
			• *PTHREAD_INHERIT_SCHED* inherits the creator's scheduling policy and own priority • *pthread_create()* and *pthread_mutex_init()* reject attribute values their setters would not store • *pthread_attr_setschedparam()* and the stack setters reject out-of-range values • the initial stack pointer of a new thread is 8-byte aligned • a detached thread's stack is freed after its last context switch, and cleanup handlers run before the stack is freed • NULL arguments to *pthread_cancel()*, *pthread_setschedprio()*, *pthread_setprivilege_np()* and four mutex getters return *EINVAL*, and *pthread_getname_np()* with a zero length returns *ERANGE*
			• *PTHREAD_MUTEX_INITIALIZER* compiles with GCC 14 and later • *pthread_mutex_timedlock()* and *pthread_mutex_clocklock()* return *ETIMEDOUT* or *EINVAL* instead of *EBUSY* • relocking a NORMAL mutex from its owner returns *EBUSY* or times out instead of blocking forever • *pthread_mutex_setprioceiling()* sets the ceiling of an unowned mutex • a failed first lock of a statically initialised mutex can be retried

continues on next page

Table 1 – continued from previous page

Software Version	Revision	Date	Notes
			• huge timeouts saturate instead of wrapping and firing at once • unsupported clock identifiers are rejected with *EINVAL*; failing clock calls and *timer_create()* return -1 instead of 255 • the *E2BIG* error number is corrected • *timer_settime()* rejects a NULL value, disarms with a malformed *it_interval* and works from the timer's own callback • *qrt_systimer_getmonotonic()* before the kernel starts returns *EINVAL*
			• *sem_init()* clears the wait list, and *sem_trywait()* works from a deferred function • *mq_open()* takes only *mq_maxmsg* and *mq_msgsize* from the attributes, which stops a queue still in use being freed on *mq_close()* • *mq_getattr()* reports *O_NONBLOCK*, *mq_open()* rejects sizes that overflow and *mq_send()* rejects lengths of 16 MiB or more • *qrt_defer_call()* keeps every call from nested interrupts and restores the caller's interrupt mask
			• a wake-up or timeout in the pass that parks a thread returns the correct result, and a retried call keeps the thread's saved FPU registers intact • ARMv8-M MPU: an interrupt arriving during a context switch no longer raises a spurious memory management fault • ARMv8-M MPU regions honour the requested shareability, and invalid permissions return *EINVAL* • *pthread_cancel()* works from a deferred function or timer callback on MPU builds • ARMv8-M stack limits are cleared when checking is off, loaded from the right address on Baseline Secure and kept inside the thread's stack • *qrt_systimer_configure()* writes the timer interrupt's priority to the right address on ARMv6-M and ARMv8-M Baseline

continues on next page

Table 1 – continued from previous page

Software Version	Revision	Date	Notes
1.2.0	3	Sep 16, 2026	New: • cycle-accurate worst-case execution time (WCET) profiling (*QRT_CFG_PERF_ENABLE*) and the Profiling chapter • the Memory Footprint chapter • constant-time O(1) scheduler and kernel services • bounded-time kernel heap allocation • priority-ordered message queues • per-system-call configuration switches • boot console and banner showing the version and release date • *pthread_setfpcontext_np()* • *SEM_VALUE_MAX*
			Changed: • kernel objects are allocated from the kernel heap, so QRT_CFG_MUTEX_LIST_SIZE, QRT_CFG_MESSAGE_QUEUE_LIMIT, QRT_CFG_MESSAGE_QUEUE_NAME_LIMIT and QRT_CFG_TIMER_LIMIT are removed • qrt_fpu_ctxenable() and qrt_fpu_ctxdisable() are replaced by *pthread_setfpcontext_np()* • *qrt_systimer_getmonotonic()* takes a signed 64-bit count • kernel linker sections .text2/.data2/.bss2 are renamed .qrt_text/.qrt_data/.qrt_bss (update custom linker scripts) • a system call whose configuration switch is absent from qrt_cfg.h is compiled out and reports *ENOSYS*, so start from the shipped qrt_cfg.h when upgrading • evaluation build limits count live objects
			Fixed: • timed pthread waits return *ETIMEDOUT* instead of -1 • *nanosleep()* intervals no longer truncated to 32 bits • scheduling policy from thread attributes honoured at creation • opaque POSIX types sized correctly • *pthread_attr_t* sized correctly on MPU-enabled builds • MPU region alignment, ARMv7-M memory attribute encoding and thread stack alignment • *errno* access from unprivileged threads • monotonic clock reads during a pending timer expiry • 8-byte stack alignment in exception handlers • demo linker script loads kernel data from flash (warm-reset HardFault)

continues on next page

Table 1 – continued from previous page

Software Version	Revision	Date	Notes
1.1.1	2	May 9, 2026	Added pthread thread-local storage (pthread_key) and *pthread_once()*
1.1.0	1	Mar 5, 2026	Added POSIX user-space APIs
1.0.0	0	May 24, 2025	Initial release

CHAPTER

TEN

GLOSSARY

10.1 A

API (Application Programming Interface)
A set of functions and protocols that allow applications to interact with the QRT kernel.

ARM Cortex-M
A family of 32-bit RISC processors designed for embedded and real-time applications. QRT supports the ARMv6-M, ARMv7-M, ARMv8-M Baseline and ARMv8-M Mainline architecture profiles.

ARM EABI (Embedded Application Binary Interface)
The ARM-defined calling and object file conventions a compiler must follow to build QRT and link against `libqrt.a`.

10.2 B

BSP (Board Support Package)
Board-specific code that performs the low-level initialization the kernel needs before booting. QRT ships a default BSP for ARM Cortex-M platforms, initialized with `bsp_init()`.

10.3 C

Cancellation
Terminating a thread from another thread with *pthread_cancel()*. Deferred cancellation takes effect at the thread's next cancellation point; asynchronous cancellation takes effect immediately.

Cancellation Point
A location where a thread using deferred cancellation checks for a pending cancel request: any blocking system call, or an explicit call to *pthread_testcancel()*.

CFSR (Configurable Fault Status Register)
An ARM Cortex-M register recording the cause of a memory, bus or usage fault. Its raw value is included in memory fault reports.

Cleanup Handler
A function registered with *pthread_cleanup_push()* that runs when a thread exits or is cancelled, used to release resources.

Console
The kernel's freestanding `printf` implementation used for boot and diagnostic output, delivered through the platform-supplied `qrt_console_puts()` hook.

Context Switching
The process of saving and restoring thread states when switching execution between threads.

CPU Core
The processing unit of a microcontroller responsible for executing instructions.

Cycle Counter
A hardware counter that increments every processor clock cycle, used by profiling to measure worst-case execution times. Cortex-M0+ and Cortex-M23 have none.

10.4 D

Deadlock
A state in which two or more threads each hold a lock the other needs and wait for it with an infinite timeout, so none of them can proceed.

Deferred Call
A kernel service request queued from an ISR or callback with *qrt_defer_call()* and executed after all interrupt handlers have completed, keeping ISR latency minimal.

Detached Thread
A thread whose resources are released automatically on termination and which cannot be joined.

Deterministic Execution
The property of a kernel where thread execution timing is predictable and guaranteed within a defined time frame.

Device Memory
An ARM memory type for peripheral registers and other memory-mapped hardware; accesses are never cached or reordered. Used for the Thread Metadata region.

10.5 F

FIFO (First-In-First-Out)
A queueing method where items are processed in the same order they were added, commonly used in message queues.

Floating-Point Unit (FPU)
A hardware module that performs floating-point arithmetic operations.

10.6 H

High-Resolution Scheduling
The scheduling mechanism in QRT that programs the System Timer for the next event that falls due instead of relying on periodic ticks, so sleeps, timeouts, timers and time slices resolve to one count of the hardware System Timer.

10.7 I

Idle Thread
A low-priority thread that runs when no other threads are ready to execute, often used for power-saving.

Idle Thread Hook
A user function registered with *qrt_kernel_idlehookregister()* that the idle thread calls during idle periods. It executes in kernel context.

Interrupt Latency
The time between when an interrupt is triggered and when the ISR begins execution.

Interrupt Masking
Disabling interrupts for a short window while the kernel updates its internal state. The longest such window is reported as `irq_mask` by profiling.

Interrupt Service Routine (ISR)
A subroutine that executes in response to hardware or software interrupts.

IPC (Inter-Process Communication)
In QRT, the exchange of data between threads through kernel objects such as message queues and semaphores.

IRQ (Interrupt Request)
A hardware interrupt line, identified by its number, that the interrupt controller routes to an ISR. The System Timer is registered with the kernel by its IRQ number.

ITM (Instrumentation Trace Macrocell)
An ARM Cortex-M debug component that streams trace data to a debugger; one possible target for console output.

10.8 J

Joinable Thread
A thread that another thread can wait for with *pthread_join()* to collect its exit status. Its resources are released once it has been joined.

10.9 K

Kernel
The core component of QRT that manages thread execution, memory protection, and system resources.

Kernel Heap
The memory pool, sized by *QRT_CFG_KERNEL_HEAP_SIZE*, from which the kernel allocates kernel objects such as message queues, timers and thread control blocks.

10.10 L

Linker Script
The linker configuration file that places code and data in memory. With Memory Protection enabled it must define the symbols describing the ROM, RAM, QRT ROM and QRT RAM regions.

10.11 M

Memory Fault
An MPU access violation. The kernel reports the faulting thread, address, instruction and CFSR value, terminates the thread, and unlocks any mutexes it holds.

Memory Protection Unit (MPU)
A hardware component that enforces memory access restrictions for different execution contexts.

Memory Region
An address range with fixed access permissions and memory attributes enforced by the MPU. QRT defines six fixed regions and allows additional per-thread regions.

Message Queue
A data structure that allows asynchronous communication between threads in a FIFO manner.

Monotonic Clock
A clock (*CLOCK_MONOTONIC*) that counts elapsed time from an unspecified starting point and cannot be set.

Mutex (Mutual Exclusion)
A synchronization primitive ensuring only one thread can access a shared resource at a time.

10.12 N

Normal Memory
An ARM memory type for code and data such as ROM and RAM; accesses may be cached and reordered, subject to a cache policy such as Write-Back or Write-Through.

10.13 O

Opaque Handle
A kernel object identifier such as *pthread_t*, *sem_t* or *mqd_t* whose internal layout is hidden from the application. It must only be passed to the API functions that accept it, never inspected or modified.

10.14 P

PendSV (Pendable Service Call)
An ARM Cortex-M system exception reserved for kernel use. The application must not install its own handler for it or change its priority.

Power-Saving Mode
An idle thread mode, enabled with *QRT_CFG_IDLE_DEEP_SLEEP_ENABLE*, that places the processor in a low-power state when no other threads are ready.

Preemptive Scheduling
A scheduling strategy where higher-priority threads can interrupt lower-priority threads.

Priority-Based Scheduling
The scheduling policy (*SCHED_FIFO*) in which the highest-priority ready thread always runs, without time slicing among threads of equal priority.

Priority Ceiling Protocol
A priority-inversion mitigation technique that temporarily raises the priority of a thread holding a mutex.

Priority Inheritance Protocol
A mechanism that prevents priority inversion by temporarily raising a lower-priority thread's priority when it holds a resource needed by a higher-priority thread.

Priority Inversion
A situation where a lower-priority thread prevents a higher-priority thread from running, typically by holding a resource it needs. It is bounded if it ends when the resource is released, and unbounded if intermediate-priority threads can prolong it.

Privileged Mode
An execution level in ARM Cortex-M processors that grants full access to system memory, peripherals, and kernel functions.

Profiling
Measuring the kernel's timing characteristics on the target. Enabled with `QRT_CFG_PERF_ENABLE` and read with *qrt_perf_read()*.

PSP / MSP (Process Stack Pointer / Main Stack Pointer)
The two ARM Cortex-M stack pointers; the processor selects one of them as the active stack pointer depending on the current execution context.

10.15 R

RAM (Random Access Memory)
The primary volatile memory where threads and system data are stored and executed.

Ready State
The state of a thread that is able to run and is waiting only for the scheduler to select it.

Real-Time Kernel
The core software responsible for managing thread execution in a real-time system.

Realtime Clock
The system-wide clock (*CLOCK_REALTIME*) representing wall-clock time. It can be read and set by the application.

ROM (Read-Only Memory)
Non-volatile memory, typically flash, that holds kernel and application code and constant data.

Round-Robin Scheduling
A scheduling algorithm that cycles through threads of equal priority, assigning each a fixed time slice.

10.16 S

Scheduler
The kernel component responsible for selecting the next thread to execute.

Scheduling Policy
A per-thread attribute selecting how the scheduler treats threads of equal priority: *SCHED_FIFO* (Priority-Based) or *SCHED_RR* (Round-Robin, the default).

Semaphore
A synchronization primitive used for signaling between threads or controlling access to shared resources.

Semihosting
A debug mechanism where the target's I/O requests are serviced by the host debugger; one possible target for console output.

Shareable
An ARM memory attribute marking a region as accessible by more than one bus master, which affects how it may be cached.

Software Timer
A kernel-managed POSIX timer that invokes a callback after a delay or at an interval, driven by the System Timer.

Stack Isolation
Preventing threads from accessing each other's stacks or kernel memory by giving each thread its own MPU-protected stack region.

Startup Thread
The first application thread, created by *qrt_kernel_boot()* from the supplied thread attributes and dispatched when the boot sequence completes.

System Call
A request from a thread to the kernel for a service, such as creating a thread or locking a mutex. System calls execute in kernel context and cannot be invoked from ISRs or the idle thread.

System Call Extension
A mechanism that allows applications to invoke custom System Calls.

System Timer
The dedicated hardware timer (also called SysTimer) used for delays, timeouts and scheduling deadlines. Its interrupt must be the only one at the highest priority level.

10.17 T

Thread
The basic unit of execution in QRT, consisting of its own stack and execution state.

Thread Attributes
The properties applied when a thread is created, held in a `pthread_attr_t` object: stack, scheduling policy and priority, detach state, time slice, privilege level and name.

Thread Control Block (TCB)
The per-thread record the kernel keeps for every thread. Its size depends on the kernel configuration (see *Thread Control Block* in the Memory Footprint chapter).

Thread-Local Storage (TLS)
A per-thread private data mechanism accessed through keys created with `pthread_key_create()`.

Thread Stack Pool
The memory pool, sized by `QRT_CFG_STACK_POOL_SIZE`, from which dynamically allocated thread stacks are taken.

Tick
One count of the hardware System Timer; the unit in which the kernel reports system time to applications written for tick-based systems. QRT needs no periodic tick interrupt (see **High-Resolution Scheduling**).

Tick-Resolution Scheduling
A traditional scheduling approach driven by a periodic tick interrupt, in which sleeps, timeouts and time slices resolve to whole tick periods.

Time Slice
The fixed time duration a thread is allowed to execute before being preempted in a round-robin scheduler.

Timeout
A predefined duration after which a blocked thread gives up waiting for an event.

TrustZone
The ARMv8-M security extension that partitions the system into Secure and Non-secure worlds. Supported where available.

10.18 U

UART (Universal Asynchronous Receiver-Transmitter)
A serial communication peripheral; a common target for console output.

Unprivileged Mode
A restricted execution mode where thread access to system resources is controlled by the MPU.

10.19 W

WCET (Worst-Case Execution Time)
: The longest time an operation has taken, measured in CPU cycles. With profiling enabled the kernel records the WCET of each system call, of a context switch and of the System Timer handler.

Weak Symbol
: A function definition the linker replaces when a strong definition exists elsewhere. The console output hook `qrt_console_puts()` is weak so the platform can override it.

Write-Back / Write-Through
: Cache policies for Normal memory. Write-Through writes reach main memory immediately, while Write-Back writes stay in the cache until the line is evicted.

10.20 Y

Yielding
: A thread voluntarily giving up the processor with *sched_yield()* so that other ready threads of equal priority can run.

INDEX

Symbols

__errno_location (*C function*), 105

B

blkcnt_t (*C type*), 99
blksize_t (*C type*), 99

C

clock_getcpuclockid (*C function*), 90
clock_getres (*C function*), 90
clock_gettime (*C function*), 90
CLOCK_MONOTONIC (*C macro*), 90
clock_nanosleep (*C function*), 91
CLOCK_REALTIME (*C macro*), 90
clock_settime (*C function*), 91
clock_t (*C type*), 98
clockid_t (*C type*), 98
close (*C function*), 97

D

dev_t (*C type*), 98

E

E2BIG (*C macro*), 104
EACCES (*C macro*), 99
EADDRINUSE (*C macro*), 99
EADDRNOTAVAIL (*C macro*), 99
EAFNOSUPPORT (*C macro*), 99
EAGAIN (*C macro*), 100
EALREADY (*C macro*), 100
EBADF (*C macro*), 100
EBADMSG (*C macro*), 100
EBUSY (*C macro*), 100
ECANCELED (*C macro*), 100
ECHILD (*C macro*), 100
ECONNABORTED (*C macro*), 100
ECONNREFUSED (*C macro*), 100
ECONNRESET (*C macro*), 100
EDEADLK (*C macro*), 100
EDESTADDRREQ (*C macro*), 100
EDOM (*C macro*), 100
EDQUOT (*C macro*), 100
EEXIST (*C macro*), 100
EFAULT (*C macro*), 101
EFBIG (*C macro*), 101
EHOSTUNREACH (*C macro*), 101
EIDRM (*C macro*), 101
EILSEQ (*C macro*), 101
EINPROGRESS (*C macro*), 101
EINTR (*C macro*), 101
EINVAL (*C macro*), 101
EIO (*C macro*), 101
EISCONN (*C macro*), 101
EISDIR (*C macro*), 101
ELOOP (*C macro*), 101
EMFILE (*C macro*), 101
EMLINK (*C macro*), 101
EMSGSIZE (*C macro*), 101
EMULTIHOP (*C macro*), 102
ENAMETOOLONG (*C macro*), 102
ENETDOWN (*C macro*), 102
ENETRESET (*C macro*), 102
ENETUNREACH (*C macro*), 102
ENFILE (*C macro*), 102
ENOBUFS (*C macro*), 102
ENODEV (*C macro*), 102
ENOENT (*C macro*), 102
ENOEXEC (*C macro*), 102
ENOLCK (*C macro*), 102
ENOLINK (*C macro*), 102
ENOMEM (*C macro*), 102
ENOMSG (*C macro*), 102
ENOPROTOOPT (*C macro*), 102
ENOSPC (*C macro*), 103
ENOSYS (*C macro*), 103
ENOTCONN (*C macro*), 103
ENOTDIR (*C macro*), 103
ENOTEMPTY (*C macro*), 103
ENOTRECOVERABLE (*C macro*), 103
ENOTSOCK (*C macro*), 103
ENOTSUP (*C macro*), 103
ENOTTY (*C macro*), 103
ENXIO (*C macro*), 103

EOPNOTSUPP (*C macro*), 103
EOVERFLOW (*C macro*), 103
EOWNERDEAD (*C macro*), 103
EPERM (*C macro*), 103
EPIPE (*C macro*), 103
EPROTO (*C macro*), 104
EPROTONOSUPPORT (*C macro*), 104
EPROTOTYPE (*C macro*), 104
ERANGE (*C macro*), 104
EROFS (*C macro*), 104
errno (*C macro*), 104
ESOCKTNOSUPPORT (*C macro*), 104
ESPIPE (*C macro*), 104
ESRCH (*C macro*), 104
ESTALE (*C macro*), 104
ETIMEDOUT (*C macro*), 104
ETXTBSY (*C macro*), 104
EWOULDBLOCK (*C macro*), 104
EXDEV (*C macro*), 104

G

gid_t (*C type*), 99

I

ino_t (*C type*), 99
itimerspec (*C struct*), 95
itimerspec.it_interval (*C var*), 95
itimerspec.it_value (*C var*), 95

M

mode_t (*C type*), 98
mq_attr (*C struct*), 87
mq_attr.mq_curmsgs (*C var*), 88
mq_attr.mq_flags (*C var*), 88
mq_attr.mq_maxmsg (*C var*), 88
mq_attr.mq_msgsize (*C var*), 88
mq_close (*C function*), 87
mq_getattr (*C function*), 84
mq_notify (*C function*), 83
mq_open (*C function*), 82
MQ_PRIO_MAX (*C macro*), 82
mq_receive (*C function*), 85
mq_send (*C function*), 84
mq_setattr (*C function*), 83
mq_timedreceive (*C function*), 86
mq_timedsend (*C function*), 84
mq_unlink (*C function*), 87
mqd_t (*C type*), 82

N

nanosleep (*C function*), 92
nlink_t (*C type*), 99

O

O_CREAT (*C macro*), 98
O_EXCL (*C macro*), 98
O_NONBLOCK (*C macro*), 97
O_RDONLY (*C macro*), 97
O_RDWR (*C macro*), 97
O_WRONLY (*C macro*), 97

P

pid_t (*C type*), 98
pthread_atfork (*C function*), 69
pthread_attr_destroy (*C function*), 59
pthread_attr_getdetachstate (*C function*), 60
pthread_attr_getguardsize (*C function*), 65
pthread_attr_getinheritsched (*C function*), 59
pthread_attr_getname_np (*C function*), 65
pthread_attr_getprivilege_np (*C function*), 66
pthread_attr_getschedparam (*C function*), 64
pthread_attr_getschedpolicy (*C function*), 63
pthread_attr_getscope (*C function*), 65
pthread_attr_getstack (*C function*), 61
pthread_attr_getstacksize (*C function*), 62
pthread_attr_gettimeslice_np (*C function*), 76
pthread_attr_init (*C function*), 58
pthread_attr_setdetachstate (*C function*), 60
pthread_attr_setguardsize (*C function*), 65
pthread_attr_setinheritsched (*C function*), 59
pthread_attr_setname_np (*C function*), 65
pthread_attr_setprivilege_np (*C function*), 66
pthread_attr_setschedparam (*C function*), 64
pthread_attr_setschedpolicy (*C function*), 63
pthread_attr_setscope (*C function*), 65
pthread_attr_setstack (*C function*), 61
pthread_attr_setstacksize (*C function*), 62
pthread_attr_settimeslice_np (*C function*), 76
pthread_attr_t (*C struct*), 99
pthread_barrier_destroy (*C function*), 73
pthread_barrier_init (*C function*), 73
pthread_barrier_t (*C type*), 98
pthread_barrier_wait (*C function*), 73
pthread_barrierattr_destroy (*C function*), 73
pthread_barrierattr_getpshared (*C function*), 73
pthread_barrierattr_init (*C function*), 73
pthread_barrierattr_setpshared (*C function*), 73
pthread_barrierattr_t (*C type*), 98
pthread_cancel (*C function*), 56
PTHREAD_CANCEL_ASYNCHRONOUS (*C macro*), 42
PTHREAD_CANCEL_DEFERRED (*C macro*), 42
PTHREAD_CANCEL_DISABLE (*C macro*), 42
PTHREAD_CANCEL_ENABLE (*C macro*), 42
pthread_cleanup_pop (*C function*), 53
pthread_cleanup_push (*C function*), 52
pthread_cond_broadcast (*C function*), 75
pthread_cond_destroy (*C function*), 75

pthread_cond_init (*C function*), 75
PTHREAD_COND_INITIALIZER (*C macro*), 43
pthread_cond_signal (*C function*), 75
pthread_cond_t (*C type*), 98
pthread_cond_timedwait (*C function*), 75
pthread_cond_wait (*C function*), 75
pthread_condattr_destroy (*C function*), 75
pthread_condattr_getclock (*C function*), 76
pthread_condattr_getpshared (*C function*), 76
pthread_condattr_init (*C function*), 75
pthread_condattr_setclock (*C function*), 76
pthread_condattr_setpshared (*C function*), 76
pthread_condattr_t (*C type*), 98
pthread_create (*C function*), 51
PTHREAD_CREATE_DETACHED (*C macro*), 42
PTHREAD_CREATE_JOINABLE (*C macro*), 42
PTHREAD_DESTRUCTOR_ITERATIONS (*C macro*), 44
pthread_detach (*C function*), 58
pthread_equal (*C function*), 55
pthread_exit (*C function*), 55
PTHREAD_EXPLICIT_SCHED (*C macro*), 43
PTHREAD_FPCONTEXT_DISABLE_NP (*C macro*), 43
PTHREAD_FPCONTEXT_ENABLE_NP (*C macro*), 43
pthread_getconcurrency (*C function*), 69
pthread_getcpuclockid (*C function*), 72
pthread_getname_np (*C function*), 68
pthread_getschedparam (*C function*), 53
pthread_getspecific (*C function*), 69
PTHREAD_INHERIT_SCHED (*C macro*), 43
pthread_join (*C function*), 58
pthread_key_create (*C function*), 70
pthread_key_delete (*C function*), 71
pthread_key_t (*C type*), 98
PTHREAD_KEYS_MAX (*C macro*), 44
pthread_kill (*C function*), 71
pthread_mutex_clocklock (*C function*), 44
pthread_mutex_consistent (*C function*), 51
PTHREAD_MUTEX_DEFAULT (*C macro*), 42
pthread_mutex_destroy (*C function*), 45
PTHREAD_MUTEX_ERRORCHECK (*C macro*), 42
pthread_mutex_getprioceiling (*C function*), 46
pthread_mutex_init (*C function*), 46
PTHREAD_MUTEX_INITIALIZER (*C macro*), 43
pthread_mutex_lock (*C function*), 47
PTHREAD_MUTEX_NORMAL (*C macro*), 42
PTHREAD_MUTEX_RECURSIVE (*C macro*), 42
PTHREAD_MUTEX_ROBUST (*C macro*), 42
pthread_mutex_setprioceiling (*C function*), 47
PTHREAD_MUTEX_STALLED (*C macro*), 42
pthread_mutex_t (*C type*), 98
pthread_mutex_timedlock (*C function*), 45
pthread_mutex_trylock (*C function*), 48
pthread_mutex_unlock (*C function*), 48
pthread_mutexattr_destroy (*C function*), 49
pthread_mutexattr_getprioceiling (*C function*), 49
pthread_mutexattr_getprotocol (*C function*), 50
pthread_mutexattr_getpshared (*C function*), 51
pthread_mutexattr_getrobust (*C function*), 51
pthread_mutexattr_gettype (*C function*), 50
pthread_mutexattr_init (*C function*), 49
pthread_mutexattr_setprioceiling (*C function*), 49
pthread_mutexattr_setprotocol (*C function*), 50
pthread_mutexattr_setpshared (*C function*), 51
pthread_mutexattr_setrobust (*C function*), 51
pthread_mutexattr_settype (*C function*), 51
pthread_mutexattr_t (*C struct*), 99
PTHREAD_NULL (*C macro*), 44
pthread_once (*C function*), 71
PTHREAD_ONCE_INIT (*C macro*), 43
pthread_once_t (*C type*), 98
PTHREAD_PRIO_INHERIT (*C macro*), 42
PTHREAD_PRIO_NONE (*C macro*), 42
PTHREAD_PRIO_PROTECT (*C macro*), 42
PTHREAD_PRIVILEGED_NP (*C macro*), 43
PTHREAD_PROCESS_PRIVATE (*C macro*), 43
PTHREAD_PROCESS_SHARED (*C macro*), 43
pthread_rwlock_clockrdlock (*C function*), 73
pthread_rwlock_clockwrlock (*C function*), 73
pthread_rwlock_destroy (*C function*), 74
pthread_rwlock_init (*C function*), 74
PTHREAD_RWLOCK_INITIALIZER (*C macro*), 44
pthread_rwlock_rdlock (*C function*), 74
pthread_rwlock_t (*C type*), 98
pthread_rwlock_timedrdlock (*C function*), 73
pthread_rwlock_timedwrlock (*C function*), 74
pthread_rwlock_tryrdlock (*C function*), 74
pthread_rwlock_trywrlock (*C function*), 74
pthread_rwlock_unlock (*C function*), 74
pthread_rwlock_wrlock (*C function*), 74
pthread_rwlockattr_destroy (*C function*), 74
pthread_rwlockattr_getpshared (*C function*), 75
pthread_rwlockattr_init (*C function*), 74
pthread_rwlockattr_setpshared (*C function*), 75
pthread_rwlockattr_t (*C type*), 98
PTHREAD_SCOPE_PROCESS (*C macro*), 43
PTHREAD_SCOPE_SYSTEM (*C macro*), 43
pthread_self (*C function*), 56
pthread_setcancelstate (*C function*), 56
pthread_setcanceltype (*C function*), 57
pthread_setconcurrency (*C function*), 69
pthread_setfpcontext_np (*C function*), 68
pthread_setname_np (*C function*), 68
pthread_setprivilege_np (*C function*), 67
pthread_setschedparam (*C function*), 54
pthread_setschedprio (*C function*), 54
pthread_setspecific (*C function*), 69

pthread_sigmask (*C function*), 72
pthread_spin_destroy (*C function*), 72
pthread_spin_init (*C function*), 72
pthread_spin_lock (*C function*), 72
pthread_spin_trylock (*C function*), 72
pthread_spin_unlock (*C function*), 72
pthread_spinlock_t (*C type*), 98
PTHREAD_STACK_MIN (*C macro*), 77
pthread_t (*C type*), 98
pthread_testcancel (*C function*), 57
PTHREAD_UNPRIVILEGED_NP (*C macro*), 43

Q

QRT_CFG_ATOMIC_DISABLE_IRQ (*C macro*), 25
QRT_CFG_CLEANUP_LIMIT (*C macro*), 24
QRT_CFG_CLOCK_GETRES_ENABLE (*C macro*), 27
QRT_CFG_CLOCK_GETTIME_ENABLE (*C macro*), 27
QRT_CFG_CLOCK_NANOSLEEP_ENABLE (*C macro*), 27
QRT_CFG_CLOCK_SETTIME_ENABLE (*C macro*), 27
QRT_CFG_CONSOLE_ENABLE (*C macro*), 30
QRT_CFG_CPU_CORE (*C macro*), 23
QRT_CFG_DCACHE_ENABLE (*C macro*), 24
QRT_CFG_DEFER_ENABLE (*C macro*), 26
QRT_CFG_DEFER_LIMIT (*C macro*), 27
QRT_CFG_ERRNO_ENABLE (*C macro*), 26
QRT_CFG_FPU_ENABLE (*C macro*), 29
QRT_CFG_FPU_STACK_ENABLE (*C macro*), 29
QRT_CFG_ICACHE_ENABLE (*C macro*), 24
QRT_CFG_IDLE_DEEP_SLEEP_ENABLE (*C macro*), 25
QRT_CFG_IDLE_SLEEP_ENABLE (*C macro*), 25
QRT_CFG_IDLE_STACK_SIZE (*C macro*), 25
QRT_CFG_KERNEL_HEAP_SIZE (*C macro*), 29
QRT_CFG_KERNEL_IDLE_HOOK_REGISTER_ENABLE (*C macro*), 27
QRT_CFG_MESSAGE_QUEUE_ENABLE (*C macro*), 26
QRT_CFG_MPU_ENABLE (*C macro*), 25
QRT_CFG_MPU_ENTRIES (*C macro*), 25
QRT_CFG_MUTEX_ENABLE (*C macro*), 26
QRT_CFG_NANOSLEEP_ENABLE (*C macro*), 27
QRT_CFG_PERF_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_GET_DETACH_STATE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_GET_INHERIT_SCHED_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_GET_PRIVILEGE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_GET_SCHED_PARAM_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_GET_SCHED_POLICY_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_GET_STACK_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_GET_STACK_SIZE_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_GET_TIMESLICE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_SET_DETACH_STATE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_SET_INHERIT_SCHED_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_SET_PRIVILEGE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_ATTR_SET_SCHED_PARAM_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_SET_SCHED_POLICY_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_SET_STACK_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_SET_STACK_SIZE_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ATTR_SET_TIMESLICE_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_CANCEL_ENABLE (*C macro*), 29
QRT_CFG_PTHREAD_CLEANUP_ENABLE (*C macro*), 27
QRT_CFG_PTHREAD_GET_SCHED_PARAM_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_KEY_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_ONCE_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_SET_SCHED_PARAM_ENABLE (*C macro*), 28
QRT_CFG_PTHREAD_SET_SCHED_PRIO_ENABLE (*C macro*), 28
QRT_CFG_SCHED_TIME_SLICE (*C macro*), 25
QRT_CFG_SCHED_TIME_SLICE_MIN_US (*C macro*), 25
QRT_CFG_SCHED_YIELD_ENABLE (*C macro*), 29
QRT_CFG_SEMAPHORE_ENABLE (*C macro*), 26
QRT_CFG_STACK_POOL_SIZE (*C macro*), 24
QRT_CFG_SYSCALL_EXT_ENABLE (*C macro*), 27
QRT_CFG_SYSCALL_EXT_LIMIT (*C macro*), 27
QRT_CFG_SYSTIMER_GET_MONOTONIC_ENABLE (*C macro*), 29
QRT_CFG_THREAD_LIMIT (*C macro*), 24
QRT_CFG_THREAD_NAME_ENABLE (*C macro*), 24
QRT_CFG_THREAD_STACK_DEFAULT (*C macro*), 24
QRT_CFG_THREAD_STACK_LIMIT_CHECK (*C macro*), 24
QRT_CFG_TIMER_ENABLE (*C macro*), 26
QRT_CFG_TZ_SECURE_WORLD (*C macro*), 26
QRT_CPU_CORTEX_A15 (*C macro*), 23
QRT_CPU_CORTEX_R5 (*C macro*), 23
QRT_CPU_RV32E (*C macro*), 23
QRT_CPU_RV32I (*C macro*), 23
qrt_defer_call (*C function*), 110
qrt_kernel_boot (*C function*), 108
qrt_kernel_idlehookregister (*C function*), 109
QRT_MPU_DEVICE (*C macro*), 112
QRT_MPU_DEVICE_GRE (*C macro*), 112
QRT_MPU_DEVICE_NGNRE (*C macro*), 112
QRT_MPU_DEVICE_NGNRNE (*C macro*), 112

QRT_MPU_EXECUTE (*C macro*), 112
QRT_MPU_EXECUTE_NEVER (*C macro*), 112
qrt_mpu_faultcallbackset (*C function*), 114
qrt_mpu_faulthandler (*C function*), 115
QRT_MPU_INNER_SHAREABLE (*C macro*), 113
QRT_MPU_NONSHAREABLE (*C macro*), 113
QRT_MPU_NORM_NCACHE (*C macro*), 112
QRT_MPU_NORM_WB_NWA (*C macro*), 112
QRT_MPU_NORM_WB_WRA (*C macro*), 112
QRT_MPU_NORM_WT_NWA (*C macro*), 112
QRT_MPU_OUTER_SHAREABLE (*C macro*), 113
QRT_MPU_READONLY (*C macro*), 112
QRT_MPU_READWRITE (*C macro*), 112
QRT_MPU_SHAREABLE (*C macro*), 113
QRT_MPU_STRONGLY_ORDERED (*C macro*), 112
QRT_MPU_SUBREGION_DISABLE (*C macro*), 113
qrt_mpu_threadregionadd (*C function*), 113
qrt_mpu_threadregionremove (*C function*), 114
qrt_perf_read (*C function*), 107
qrt_s16_t (*C type*), 105
qrt_s32_t (*C type*), 105
qrt_s64_t (*C type*), 105
qrt_s8_t (*C type*), 105
qrt_sptr_t (*C type*), 105
qrt_syscall_ext_t (*C type*), 105
qrt_syscall_invoke (*C function*), 107
qrt_syscall_register (*C function*), 106
qrt_systimer_configure (*C function*), 109
qrt_systimer_getmonotonic (*C function*), 110
qrt_systimer_handler (*C function*), 110
qrt_u16_t (*C type*), 105
qrt_u32_t (*C type*), 105
qrt_u64_t (*C type*), 105
qrt_u8_t (*C type*), 105
qrt_uptr_t (*C type*), 105

R

read (*C function*), 97

S

SCHED_FIFO (*C macro*), 88
sched_get_priority_max (*C function*), 88
sched_get_priority_min (*C function*), 88
sched_getparam (*C function*), 89
sched_getscheduler (*C function*), 89
SCHED_OTHER (*C macro*), 88
sched_param (*C struct*), 89
sched_param.sched_priority (*C var*), 89
SCHED_RR (*C macro*), 88
sched_rr_get_interval (*C function*), 89
sched_setparam (*C function*), 89
sched_setscheduler (*C function*), 89
SCHED_SPORADIC (*C macro*), 88
sched_yield (*C function*), 89
sem_clockwait (*C function*), 79
sem_close (*C function*), 81
sem_destroy (*C function*), 78
SEM_FAILED (*C macro*), 77
sem_getvalue (*C function*), 81
sem_init (*C function*), 77
sem_open (*C function*), 81
sem_post (*C function*), 78
sem_t (*C type*), 77
sem_timedwait (*C function*), 79
sem_trywait (*C function*), 80
sem_unlink (*C function*), 81
SEM_VALUE_MAX (*C macro*), 77
sem_wait (*C function*), 80
SIGEV_NONE (*C macro*), 96
SIGEV_SIGNAL (*C macro*), 96
SIGEV_THREAD (*C macro*), 96
sigevent (*C struct*), 96
sigevent.sigev_notify (*C var*), 96
sigevent.sigev_notify_attributes (*C var*), 97
sigevent.sigev_notify_function (*C var*), 97
sigevent.sigev_value (*C var*), 96
sigset_t (*C type*), 96
sigval (*C union*), 96
sigval.sival_int (*C var*), 96
sigval.sival_ptr (*C var*), 96
size_t (*C type*), 99
sleep (*C function*), 97
ssize_t (*C type*), 99

T

time_t (*C type*), 99
TIMER_ABSTIME (*C macro*), 90
timer_create (*C function*), 92
timer_delete (*C function*), 93
timer_getoverrun (*C function*), 94
timer_gettime (*C function*), 94
timer_settime (*C function*), 94
timer_t (*C type*), 99
timespec (*C struct*), 95
timespec.tv_nsec (*C var*), 95
timespec.tv_sec (*C var*), 95

U

uid_t (*C type*), 99
useconds_t (*C type*), 99

W

write (*C function*), 97