Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ QuRT RTOS

User Guide for Hexagon SDK

80-VB419-178 Rev. C

July 13, 2023

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision History

Revision	Date	Description
A	Dec 2017	Initial release.
B	April 2021	Updated for QuRT version 04.00.xx. Updated Sections 3.2.1.1 and 24.3.1.1. Added new Chapters: • Atomic Operations (Chapter 27) • HVX (Chapter 29) • Appendix B Debugging Errors and Cause Codes. Added new functions: • qurt_thread_attr_set_stack_size2 (Section 3.8) • qurt_thread_attr_set_detachstate (Section 3.23) • qurt_thread_stid_set (Section 3.25) • qurt_sleep (Section 3.26) • qurt_thread_get_tls_base (Section 3.27) • qurt_qurt_busy_wait (Section 3.28) • qurt_process_attr_set_max_threads (Section 4.7) • qurt_process_attr_set_ceiling_prio (Section 4.8) • qurt_process_get_thread_count (Section 4.9) • qurt_process_get_thread_ids (Section 4.10) • qurt_process_attr_get (Section 4.11) • qurt_process_dump_register_cb (Section 4.12) • qurt_process_dump_deregister_cb (Section 4.13) • qurt_mutex_lock_timed (Section 5.6) • qurt_rmutex_lock_timed (Section 6.6) • qurt_rmutex_try_lock_block_once (Section 6.7) • qurt_signal_wait_timed (Section 8.10) • qurt_anysignal_wait_timed (Section 9.7) • qurt_sem_down_timed (Section 11.9) • qurt_assert_error (Section 19.5) • qurt_lookup_physaddr2 (Section 21.2) • qurt_mem_cache_phys_clean (Section 21.11) • qurt_mem_pool_is_available (Section 21.21) • qurt_sysenv_get_hw_threads (Section 22.8) • qurt_profile_enable2 (Section 23.3) • qurt_profile_get (Section 23.4) • qurt_get_hthread_pcycles (Section 23.7) • qurt_get_hthread_commits (Section 23.8) • qurt_etm_set_pc_range (Section 26.3) • qurt_etm_set_atb (Section 26.4) • qurt_stm_trace_set_config (Section 26.5) • qurt_cb_data_set_cbarg (Section 28.1) • qurt_cb_data_set_cbfunc (Section 28.2) • qurt_cb_data_init (Section 28.3)

Revision	Date	Description
C	July 2023	Updated for QuRT 4.3. Updated Tables B.5 and B.6. Added new Chapters: • Memory Mapping (Chapter 22) • SRM Drivers (Chapter 30) Added new functions: • qurt_thread_attr_set_autostack (Section 3.4) • qurt_thread_set_autostack (Section 3.5) • qurt_thread_get_running_ids (Section 3.27) • qurt_thread_get_thread_id (Section 3.28) • qurt_system_set_priority_floor (Section 3.30) • qurt_process_attr_set_binary_path (Section 4.3) • qurt_process_attr_set_dtb_path (Section 4.4) • qurt_process_attr_set_sw_id (Section 4.10) • qurt_process_exit (Section 4.17) • qurt_process_kill (Section 4.18) • qurt_debugger_register_process (Section 4.19) • qurt_debugger_deregister_process (Section 4.20) • qurt_process_exec_callback (Section 4.21) • qurt_process_get_pid (Section 4.22) • qurt_interrupt_register2 (Section 17.9) • qurt_interrupt_set_config2 (Section 17.11) • qurt_interrupt_set_config3 (Section 17.12) • qurt_interrupt_raise2 (Section 17.15) • qurt_exception_wait3 (Section 19.5) • qurt_mem_pool_attach2 (Section 21.19) • qurt_sysenv_get_process_name2 (Section 23.5) • qurt_pmu_get_pmucnt (Section 25.4) • qurt_etm_set_range (Section 27.4) • qurt_etm_set_sync_period (Section 27.6)

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This document is designed to serve as a reference for C programmers experienced in real-time software development. It provides only basic information on real-time concurrent programming. For more information, see ISBN 0470128720.

The QuRT™ operating system is a real-time operating system (RTOS) for the Qualcomm Hexagon™ processor. It supports multithreading, thread communication and synchronization, interrupt handling, and memory management.

Note: This document has been updated to reflect APIs introduced in QuRT 4.3.

1.2 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands and command variables appear in a different font, for example, **`copy a:*.* b:`**.

Parameter directions are indicated as follows:

- `[in]` indicates an input parameter.
- `[out]` indicates an output parameter.
- `[in,out]` indicates a parameter used for both input and output.

1.3 Technical Assistance

For assistance or clarification on information in this document, open a technical support case at https://support.qualcomm.com.

You will need to register for a Qualcomm ID account and your company must have support enabled to access our Case system.

Other systems and support resources are listed on https://support.qualcomm.com.

If you need further assistance, you can send an email to qualcomm.support@qti.qualcomm.com.

1.4 QuRT Features

QuRT offers low overhead (both in memory and processing), simple implementation, and eases porting porting standalone user programs to the QuRT environment and modification to accommodate specific target requirements

QuRT consists of:

- The kernel, which provides system operations for a minimal set of operating system facilities. The kernel handles thread creation, scheduling, blocking, and performs basic memory management.
- The library, which provides an application programming interface (API) to the kernel operations and additional library functions to aid in programming.
- The configuration files, which encapsulate target-specific information used to configure QuRT for target platforms.

Note: QuRT is a simplified operating system – it does not provide many facilities that are commonly available in other operating systems.

QuRT offers:

- Real-time priority-based preemptive Multithreading:
  - Multiple threads (or flows of execution) can execute at the same time in a user program. QuRT initially assigns the program a single thread of execution, the program can then create additional threads. The Hexagon processor can execute a fixed number of threads simultaneously – additional threads must share the processor. QuRT handles the sharing details.
  - Each thread is assigned a priority level that determines which thread has execution priority.
  - A thread can be preempted – have the processor taken away – when a higher-priority thread is ready to execute.
  - The operating system can perform operations within certain periods of time.
- Processes, which enable programs and threads to execute in separate protected address spaces for improved system security and stability.
- Mutexes, which synchronize threads to ensure mutually exclusive access to shared resources.
- Signals: QuRT objects that synchronize threads on sets of mutex-like signals.
- Semaphores, which synchronize threads to ensure limited access to shared resources.
- Barriers: QuRT object used to synchronize threads to meet at a specific point in a user program.
- Condition variables: QuRT objects used to synchronize threads based on the value of a data item.
- Pipes, which support synchronized data exchange between threads.
- Timers, with which threads can schedule actions to occur at specific times or intervals.
- Interrupt handling, which registers threads to serve as interrupt handlers.
- Thread local storage, which allocates global storage that is private to specific threads.
- Exception handling, which supports exception handling for fatal and non-fatal exceptions.
- Memory management, which enables user programs to dynamically manage their memory space.
- Profiling, which record cycle counts (both running and idle) for specific threads.

- Performance monitor, which supports code performance measurement during user program execution.
- Function tracing, which supports debugging macros for tracing function calls and returns.

1.5 Processor Versions

QuRT 4.3 supports Hexagon processor versions V66, V67, V68, V69, V71, V73, and and V75.

2 Using QuRT

2.1 User Programs

A QuRT system contains one or more user programs. Each user program is a complete program that uses the QuRT API (see Section 2.3) to access the QuRT services. A user program is assigned a single thread when the program starts. To create additional threads, the program uses the QuRT thread services.

A user program consists of one or more C or assembly source files (some of which include the QuRT API header file).

A user program memory image includes:

- Default global heap
- Main thread call stack
- Data and text sections of the program
- Heaps and thread call stacks allocated by the program

The user specifies the size of the global heap when building the user program (Section 2.2).

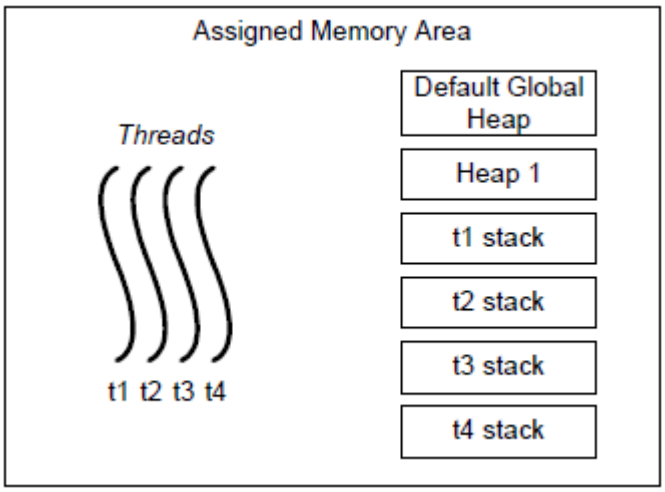

Figure 2-1 User program image

QuRT prevents user programs from accessing unauthorized areas of system memory. If a thread attempts to access memory outside its assigned memory area, QuRT generates a memory exception.

2.2 Build Procedure

QuRT user programs are written in C/C++ and Hexagon assembly language, and use the QuRT APIs to access the RTOS services.

The build procedure for a QuRT user program is similar to the standard procedure for building a standalone C/C++ program.

QuRT libraries (including the RTOS kernel) are provided as object files – no source code is provided. Multiple versions of the QuRT libraries are provided to support different hardware and software targets. Each library version is optimized for its specific target.

Before building a QuRT system, users must define the system configuration in a user-editable configuration file (cust_config.xml). This configuration file is used to generate a configuration object file, which is linked with the QuRT RTOS when it is built.

Building a QuRT system creates a single boot image, which can via the following methods:

- Software simulation using the Hexagon simulator
- In-circuit emulation using a hardware test platform (Rumi, ZeBu, SURF)

Note: QuRT user programs use the standard C library to perform operations supported by the standard library (in particular, malloc and printf).

2.3 API

The QuRT APIs are a C header file named qurt.h, which is included in the source code of each QuRT user program. For example:

```
#include "qurt.h"
...
qurt_mutex_lock(&my_mutex); /* QuRT API function */
```

To indicate that they are part of QuRT, the function, type, and constant names defined in the QuRT API begin with the prefix qurt_. Preprocessor definitions in the QuRT API include the prefix QURT_. Functions and data structures in the kernel include the prefix QURTK_.

2.4 Objects

A QuRT user program accesses most QuRT services by defining objects and performing operations on them. For example:

```
qurt_mutex_t my_mutex; /* Mutex object */
...
qurt_mutex_init(&my_mutex); /* Init mutex object */
...
qurt_mutex_lock(&my_mutex); /* Lock mutex */
...
qurt_mutex_destroy(&my_mutex); /* Destroy mutex object */
```

QuRT objects support two sets of operations to manage objects:

- Initialize and destroy operations (shown in the preceding example) for objects that are stored entirely in memory that the user program allocates.
- Create and delete operations for objects that are stored partly in memory allocated automatically by the RTOS kernel.

Pipe objects support both operation pairs: use initialize and destroy when the pipe buffer is user-allocated, use create and delete when the pipe buffer is automatically allocated by the kernel as part of initializing a pipe object.

Timer objects support only create and delete for object management. All other QuRT objects support only initialize and destroy for object management.

In addition to object management, most objects define additional operations that perform services associated with that object, such as qurt_mutex_lock() in the previous example.

Note: Objects must be destroyed (with the destroy or delete operation) when no longer in use. Failure to do so causes resource leaks in the QuRT kernel.

Treat QuRT objects as having opaque types, accessed only through QuRT functions.

2.5 Nonblocking and Cancellable Operations

QuRT defines operations that are nonblocking or cancellable versions of other QuRT operations (lock, down, wait, send, receive). For example:

- qurt_mutex_try_lock
- qurt_sem_try_down
- qurt_signal_wait_cancellable
- qurt_pipe_send_cancellable

The nonblocking operations are identified by the prefix "try_" in their operation names, the cancellable operations use the suffix "_cancellable".

Nonblocking operations enable a thread to attempt to perform an operation without the risk of having the thread suspended - if the operation fails, it immediately returns with an error result.

Cancellable operations automatically return if a system-level event interrupts the calling thread: in particular, if the user process of the thread is killed, or if the thread must finish its current QuRT driver

invocation (QDI) and return to user space.

When an operation is canceled, the calling thread must assume that the operation never completes: the caller must stop waiting for the specified resource or event, and assume that the event never occurs or the resource never becomes available.

Note: Cancellation differs from a process shutdown, and should not be handled as such.

> If a driver detects a canceled operation, it must propagate an error result back to its caller as directly as possible. The driver must also be sure to leave its internal data structures in a valid and predictable state.

2.6 64-bit Operations

The QuRT memory management service defines both 32-bit and 64-bit versions of certain operations. The 32-bit operations are provided for backward compatibility with earlier versions of QuRT. The 64-bit operations are functionally equivalent to the corresponding 32-bit operations, but can access memory addresses above 4 GB.

The 64-bit operations are identified by the suffix "_64" in their operation names.

3 Threads

A thread is a sequence of instructions running in a user program. Multitasking allows multiple instruction sequences in a user program to execute in parallel. When started, a thread exists in one of four states listed in Table 3-1. The kernel is responsible for switching threads between these states.

Table 3-1 Thread states

State	Description
Ready	The thread is ready to run, but prevented from running because a higher priority thread is executing.
Running	The thread is executing.
Waiting	The thread is waiting for an event to occur or a shared resource to become available.
Stopped	The thread no longer exists, having been destroyed.

A thread is suspended when it changes state from Running to Ready or Waiting, and awakened when it changes from Waiting to Ready. All threads are initialized to Ready.

Thread priority determines the execution priority of the thread and how often the thread executes relative to the other threads in the system. Threads are assigned priorities when they are first created. Priorities are specified as numeric values in a range as large as 0 to 255, with lower values representing higher priorities. 0 represents the highest possible thread priority.

The kernel uses a scheduler to determine which threads to run – the scheduler always selects the highest-priority ready threads. During system startup, the scheduler selects the highest-priority threads for execution and changes their thread state to Running. If two ready-state threads have different priorities, but only one hardware thread is available, the kernel executes the thread with higher priority until it is suspended. In some cases a user program system must adjust the priority of a thread after it has been created. For instance, to prevent priority inversion, a thread might need to raise its own priority or the priority of another thread.

Note: QuRT can be configured to have different priority ranges (Section 2.2).

The thread entry point is the function executed by the thread when it starts. The thread argument is a pointer passed to the thread function when the thread starts. It allows a single function to be written so it can be executed by multiple threads. The function is defined in the user program, and must accept a single void pointer as a function parameter.

The kernel-generated thread identifier returned by the thread create operation specifies the thread in API thread operations. This identifier is different from the thread name or timetest ID, which identify threads during debugging or profiling.

QuRT is preemptive – a context switch occurs when a kernel operation suspends the current thread or awakens a higher-priority thread. The following kernel operations can cause a context switch:

- Creating or exiting a thread
- Changing a thread priority
- Waiting on or releasing a mutex or semaphore
- Waiting on or resuming from a signal, barrier, or condition variable
- Reading or writing from a pipe
- Interrupt

Figure 3-1 shows the events that can cause the kernel to perform a context switch.

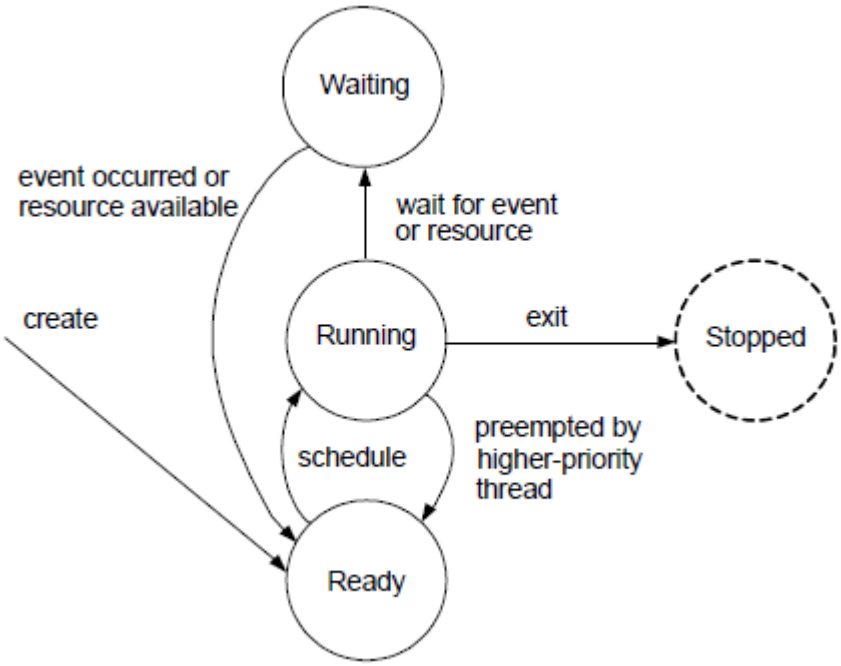

Figure 3-1 Thread state transitions

Threads have two kinds of attributes:

- Static attributes: cannot be changed after a thread is created
- Dynamic attributes: can be changed after the thread is created

The only dynamic thread attributes are priority, timetest ID, and cache partition – all the other threads are static.

Bus priority is the internal bus priority state. TCB partition is memory used for allocating thread control blocks (TCBs). The TCB partition specifies the maximum number of threads that have their TCBs allocated in tightly coupled memory (TCM)/low power memory (LPM) instead of regular memory.

The thread stack address and stack size (in bytes) specify the memory area used as a call stack for the thread. The user is responsible for allocating the memory area used for the stack.

Static attributes are set both before a thread is created using the qurt_thread_attr_init and

qurt_thread_attr_set functions, and when a thread is created by directly passing the attributes as arguments to qurt_thread_create().

Dynamic attributes are set after a thread is created using the qurt_thread_set functions.

Note: The timetest ID attribute is stored in a Hexagon processor register.

Threads are represented as shared objects in QuRT. Thread objects support the following operations:

- qurt_thread_attr_get()
- qurt_thread_attr_init()
- qurt_thread_attr_set_bus_priority()
- qurt_thread_attr_set_autostack()
- qurt_thread_set_autostack()
- qurt_thread_attr_set_name()
- qurt_thread_attr_set_priority()
- qurt_thread_attr_set_stack_addr()
- qurt_thread_attr_set_stack_size()
- qurt_thread_attr_set_stack_size2()
- qurt_thread_attr_set_tcb_partition()
- qurt_thread_attr_set_timetest_id()
- qurt_thread_create()
- qurt_thread_exit()
- qurt_thread_get_id()
- qurt_thread_get_l2cache_partition()
- qurt_thread_get_name()
- qurt_thread_get_priority()
- qurt_thread_get_timetest_id()
- qurt_thread_join()
- qurt_thread_resume()
- qurt_thread_set_cache_partition()
- qurt_thread_set_priority()
- qurt_thread_attr_set_detachstate()
- qurt_thread_set_timetest_id()
- qurt_thread_stid_set()
- qurt_thread_get_running_ids()
- qurt_thread_get_thread_id()
- qurt_sleep()
- qurt_system_set_priority_floor()
- qurt_thread_get_tls_base()
- qurt_busywait()

- Data Types
- Constants and Macros

3.1 qurt_thread_attr_get()

3.1.1 Function Documentation

3.1.1.1 int qurt_thread_attr_get (qurt_thread_t *thread_id,* qurt_thread_attr_t * *attr*)

Gets the attributes of the specified thread.

Associated data types

qurt_thread_t
qurt_thread_attr_t

Parameters

in	*thread_id*	Thread identifier.
out	*attr*	Pointer to the destination structure for thread attributes.

Returns

QURT_EOK – Success.
QURT_EINVALID – Invalid argument.

Dependencies

None.

3.2 qurt_thread_attr_init()

3.2.1 Function Documentation

3.2.1.1 static void qurt_thread_attr_init (qurt_thread_attr_t * *attr*)

Initializes the structure used to set the thread attributes when a thread is created. After an attribute structure is initialized, Explicity set the individual attributes in the structure using the thread attribute operations.

The initialize operation sets the following default attribute values:

- Name – NULL string
- TCB partition – QURT_THREAD_ATTR_TCB_PARTITION_DEFAULT
- Priority – QURT_THREAD_ATTR_PRIORITY_DEFAULT
- Autostack – QURT_THREAD_ATTR_AUTOSTACK_DEFAULT
- Bus priority – QURT_THREAD_ATTR_BUS_PRIO_DEFAULT
- Timetest ID – QURT_THREAD_ATTR_TIMETEST_ID_DEFAULT
- stack_size – 0
- stack_addr – NULL
- detach state – QURT_THREAD_ATTR_CREATE_LEGACY
- STID – QURT_THREAD_ATTR_STID_DEFAULT

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.

Returns

None.

Dependencies

None.

3.3 qurt_thread_attr_set_bus_priority()

3.3.1 Function Documentation

3.3.1.1 static void qurt_thread_attr_set_bus_priority (qurt_thread_attr_t * *attr,* unsigned short *bus_priority*)

Sets the internal bus priority state in the Hexagon core for this software thread attribute. Memory requests generated by the thread with bus priority enabled are given priority over requests generated by the thread with bus priority disabled. The default value of bus priority is disabled.

NOTE Sets the internal bus priority for Hexagon processor version V60 or greater. The priority is not propagated to the bus fabric.

Associated data types

qurt_thread_attr_t

Parameters

in	*attr*	Pointer to the thread attribute structure.
in	*bus_priority*	Enabling flag. Values: • QURT_THREAD_BUS_PRIO_DISABLED • QURT_THREAD_BUS_PRIO_ENABLED

Returns

None

Dependencies

None.

3.4 qurt_thread_attr_set_autostack()

3.4.1 Function Documentation

3.4.1.1 static void qurt_thread_attr_set_autostack (qurt_thread_attr_t * *attr,* unsigned short *autostack*)

Enables autostack v2 feature in the thread attributes.

When autostack is enabled by the subsystem, in the case that an autostack enabled thread gets framelimit exception, kernel will allocate more stack for thread and return to normal execution.

If autostack is not enabled by the subsystem, or it is not enabled for the thread, the framelimit exception will be fatal.

Associated data types

qurt_thread_attr_t

Parameters

in	*attr*	Pointer to the thread attribute structure.
in	*autostack*	Autostack enable or disable flag. Values: • QURT_THREAD_AUTOSTACK_DISABLED • QURT_THREAD_AUTOSTACK_ENABLED

Returns

None.

Dependencies

None.

3.5 qurt_thread_set_autostack()

3.5.1 Function Documentation

3.5.1.1 void qurt_thread_set_autostack (void *)

Sets autostack enable in the TCB. Takes as input a pointer to the user general pointer (UGP).

Returns

None.

Dependencies

None.

3.6 qurt_thread_attr_set_name()

3.6.1 Function Documentation

3.6.1.1 static void qurt_thread_attr_set_name (qurt_thread_attr_t * *attr,* const char * *name*)

Sets the thread name attribute.

This function specifies the name to use by a thread. Thread names identify a thread during debugging or profiling. Maximum name length is 16 charactes

NOTE Thread names differ from the kernel-generated thread identifiers used to specify threads in the API thread operations.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*name*	Pointer to the character string containing the thread name.

Returns

None.

Dependencies

None.

3.7 qurt_thread_attr_set_priority()

3.7.1 Function Documentation

3.7.1.1 static void qurt_thread_attr_set_priority (qurt_thread_attr_t * *attr,* unsigned short *priority*)

Sets the thread priority to assign to a thread. Thread priorities are specified as numeric values in the range 1 to 254, with 1 representing the highest priority. Priority 0 and 255 are internally used by the kernel for special purposes.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*priority*	Thread priority.

Returns

None.

Dependencies

None.

3.8 qurt_thread_attr_set_stack_addr()

3.8.1 Function Documentation

3.8.1.1 static void qurt_thread_attr_set_stack_addr (qurt_thread_attr_t * *attr,* void * *stack_addr*)

Sets the thread stack address attribute.

Specifies the base address of the memory area to use for a call stack of a thread.

stack_addr must contain an address value that is 8-byte aligned.

The thread stack address and stack size (Section 3.10.1.1) specify the memory area used as a call stack for the thread.

NOTE The user is responsible for allocating the memory area used for the thread stack. The memory area must be large enough to contain the stack that the thread creates.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*stack_addr*	Pointer to the 8-byte aligned address of the thread stack.

Returns

None.

Dependencies

None.

3.9 qurt_thread_attr_set_stack_size()

3.9.1 Function Documentation

3.9.1.1 static void qurt_thread_attr_set_stack_size (qurt_thread_attr_t * *attr,* unsigned int *stack_size*)

Sets the thread stack size attribute.

Specifies the size of the memory area to use for a call stack of a thread.

The thread stack address (Section 3.8.1.1) and stack size specify the memory area used as a call stack for the thread. The user is responsible for allocating the memory area used for the stack.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*stack_size*	Size (in bytes) of the thread stack.

Returns

None.

Dependencies

None.

3.10 qurt_thread_attr_set_stack_size2()

3.10.1 Function Documentation

3.10.1.1 static void qurt_thread_attr_set_stack_size2 (qurt_thread_attr_t * *attr,* unsigned short *user_stack_size,* unsigned short *root_stack_size*)

Sets the thread stack size attribute for island threads that require a higher guest OS stack size than the stack size defined in the configuration XML.

Specifies the size of the memory area to use for a call stack of an island thread in User and Guest mode.

The thread stack address (Section 3.8.1.1) and stack size specify the memory area used as a call stack for the thread. The user is responsible for allocating the memory area used for the stack.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*user_stack_size*	Size (in bytes) of the stack usage in User mode.
in	*root_stack_size*	Size (in bytes) of the stack usage in Guest mode.

Returns

None.

Dependencies

None.

3.11 qurt_thread_attr_set_tcb_partition()

3.11.1 Function Documentation

3.11.1.1 static void qurt_thread_attr_set_tcb_partition (qurt_thread_attr_t * *attr,* unsigned char *tcb_partition*)

Sets the thread TCB partition attribute. Specifies the memory type where a TCB of a thread is allocated. Allocates TCBs in RAM or TCM/LPM.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*tcb_partition*	TCB partition. Values: • 0 – TCB resides in RAM • 1 – TCB resides in TCM/LCM

Returns

None.

Dependencies

None.

3.12 qurt_thread_attr_set_timetest_id()

3.12.1 Function Documentation

3.12.1.1 static void qurt_thread_attr_set_timetest_id (qurt_thread_attr_t * *attr,* unsigned short *timetest_id*)

Sets the thread timetest attribute.

Specifies the timetest identifier to use by a thread.

Timetest identifiers are used to identify a thread during debugging or profiling.

NOTE Timetest identifiers differ from the kernel-generated thread identifiers used to specify threads in the API thread operations.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*timetest_id*	Timetest identifier value.

Returns

None.

Dependencies

None.

3.13 qurt_thread_create()

3.13.1 Function Documentation

3.13.1.1 int qurt_thread_create (qurt_thread_t * *thread_id,* qurt_thread_attr_t * *attr,* void(*)(void *) *entrypoint,* void * *arg*)

Creates a thread with the specified attributes, and makes it executable.

Associated data types

qurt_thread_t
qurt_thread_attr_t

Parameters

out	*thread_id*	Returns a pointer to the thread identifier if the thread was successfully created.
in	*attr*	Pointer to the initialized thread attribute structure that specifies the attributes of the created thread.
in	*entrypoint*	C function pointer, which specifies the main function of a thread.
in	*arg*	Pointer to a thread-specific argument structure

Returns

QURT_EOK – Thread created.
QURT_EFAILED – Thread not created.

Dependencies

None.

3.14 qurt_thread_exit()

3.14.1 Function Documentation

3.14.1.1 void qurt_thread_exit (int *status*)

Stops the current thread, awakens threads joined to it, then destroys the stopped thread.

Threads that are suspended on the current thread (by performing a thread join Section 3.20.1.1) are awakened and passed a user-defined status value that indicate the status of the stopped thread.

NOTE Exit must be called in the context of the thread to stop.

Parameters

in	*status*	User-defined thread exit status value.

Returns

None.

Dependencies

None.

3.15 qurt_thread_get_id()

3.15.1 Function Documentation

3.15.1.1 qurt_thread_t qurt_thread_get_id (void)

Gets the identifier of the current thread.

Returns the thread identifier for the current thread.

Returns

Thread identifier – Identifier of the current thread.

Dependencies

None.

3.16 qurt_thread_get_l2cache_partition()

3.16.1 Function Documentation

3.16.1.1 qurt_cache_partition_t qurt_thread_get_l2cache_partition (void)

Returns the current value of the L2 cache partition assigned to the caller thread.

Returns

Value of the qurt_cache_partition_t data type.

Dependencies

None.

3.17 qurt_thread_get_name()

3.17.1 Function Documentation

3.17.1.1 void qurt_thread_get_name (char * *name*, unsigned char *max_len*)

Gets the thread name of current thread.

Returns the thread name of the current thread. Thread names are assigned to threads as thread attributes, see qurt_thread_attr_set_name(). Thread names identify a thread during debugging or profiling.

Parameters

out	*name*	Pointer to a character string, which specifies the address where the returned thread name is stored.
in	*max_len*	Maximum length of the character string that can be returned.

Returns

None.

Dependencies

None.

3.18 qurt_thread_get_priority()

3.18.1 Function Documentation

3.18.1.1 int qurt_thread_get_priority (qurt_thread_t *threadid*)

Gets the priority of the specified thread.

Returns the thread priority of the specified thread.

Thread priorities are specified as numeric values in a range as large as 1 through 254, with lower values representing higher priorities. 1 represents the highest possible thread priority.

Priority 0 and 255 are internally used by the kernel for special purposes.

NOTE QuRT can be configured to have different priority ranges.

Associated data types

qurt_thread_t

Parameters

in	*threadid*	Thread identifier.

Returns

-1 – Invalid thread identifier.
1 through 254 – Thread priority value.

Dependencies

None.

3.19 qurt_thread_get_timetest_id()

3.19.1 Function Documentation

3.19.1.1 unsigned short qurt_thread_get_timetest_id (void)

Gets the timetest identifier of the current thread.

Returns the timetest identifier of the current thread.

Timetest identifiers are used to identify a thread during debugging or profiling.

NOTE Timetest identifiers differ from the kernel-generated thread identifiers used to specify threads in the API thread operations.

Returns

Integer – Timetest identifier.

Dependencies

None.

3.20 qurt_thread_join()

3.20.1 Function Documentation

3.20.1.1 int qurt_thread_join (unsigned int *tid,* int * *status*)

Waits for a specified thread to finish; the specified thread is another thread within the same process. The caller thread is suspended until the specified thread exits. When the unspecified thread exits, the caller thread is awakened.

NOTE If the specified thread has already exited, this function returns immediately with the result value QURT_ENOTHREAD.

Two threads cannot call qurt_thread_join to wait for the same thread to finish. If this occurs, QuRT generates an exception (see Section 19).

Parameters

in	*tid*	Thread identifier.
out	*status*	Destination variable for thread exit status. Returns an application-defined value that indicates the termination status of the specified thread.

Returns

QURT_ENOTHREAD – Thread has already exited.
QURT_EOK – Thread successfully joined with valid status value.

Dependencies

None.

3.21 qurt_thread_resume()

3.21.1 Function Documentation

3.21.1.1 int qurt_thread_resume (unsigned int *thread_id*)

Resumes the execution of a suspended thread.

Parameters

in	*thread_id*	Thread identifier.

Returns

QURT_EOK – Thread successfully resumed.
QURT_EFATAL – Resume operation failed.

Dependencies

None.

3.22 qurt_thread_set_cache_partition()

3.22.1 Function Documentation

3.22.1.1 void qurt_thread_set_cache_partition (qurt_cache_partition_t *l1_icache,* qurt_cache_partition_t *l1_dcache,* qurt_cache_partition_t *l2_cache*)

Sets the cache partition for the current thread. This function uses the qurt_cache_partition_t type to select the cache partition of the current thread for the L1 Icache, L1 Dcache, and L2 cache.

Associated data types

qurt_cache_partition_t

Parameters

in	*l1_icache*	L1 Icache partition.
in	*l1_dcache*	L1 Dcache partition.
in	*l2_cache*	L2 cache partition.

Returns

None.

Dependencies

None.

3.23 qurt_thread_set_priority()

3.23.1 Function Documentation

3.23.1.1 int qurt_thread_set_priority (qurt_thread_t *threadid,* unsigned short *newprio*)

Sets the priority of the specified thread.

Thread priorities are specified as numeric values in a range as large as 1 through 254, with lower values representing higher priorities. 1 represents the highest possible thread priority. Priority 0 and 255 are internally used by the kernel for special purposes.

NOTE QuRT can be configured to have different priority ranges. For more information, see Section 2.2.

Associated data types

qurt_thread_t

Parameters

in	*threadid*	Thread identifier.
in	*newprio*	New thread priority value.

Returns

0 – Priority successfully set.
-1 – Invalid thread identifier.

Dependencies

None.

3.24 qurt_thread_attr_set_detachstate()

3.24.1 Function Documentation

3.24.1.1 static void qurt_thread_attr_set_detachstate (qurt_thread_attr_t * *attr,* unsigned short *detachstate*)

Sets the thread detach state with which thread is created. Thread detach state is either joinable or detached; specified by the following values:

- QURT_THREAD_ATTR_CREATE_JOINABLE
- QURT_THREAD_ATTR_CREATE_DETACHED

When a detached thread is created (QURT_THREAD_ATTR_CREATE_DETACHED), its thread ID and other resources are reclaimed as soon as the thread exits. When a joinable thread is created (QURT_THREAD_ATTR_CREATE_JOINABLE), it is assumed that some thread waits to join on it using a qurt_thread_join() call. By default, detached state is QURT_THREAD_ATTR_CREATE_LEGACY If detached state is QURT_THREAD_ATTR_CREATE_LEGACY then other thread can join before thread exits but it will not wait other thread to join.

NOTE For a joinable thread (QURT_THREAD_ATTR_CREATE_JOINABLE), it is very important that some thread joins on it after it terminates, otherwise the resources of that thread are not reclaimed, causing memory leaks.

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*detachstate*	Thread detach state.

Returns

None.

Dependencies

None.

3.25 qurt_thread_set_timetest_id()

3.25.1 Function Documentation

3.25.1.1 void qurt_thread_set_timetest_id (unsigned short *tid*)

Sets the timetest identifier of the current thread. Timetest identifiers are used to identify a thread during debugging or profiling.

NOTE Timetest identifiers differ from the kernel-generated thread identifiers used to specify threads in the API thread operations.

Parameters

in	*tid*	Timetest identifier.

Returns

None.

Dependencies

None.

3.26 qurt_thread_stid_set()

3.26.1 Function Documentation

3.26.1.1 int qurt_thread_stid_set (char *stid*)

Sets the STID for a specified thread.

Associated data types

qurt_thread_t

Parameters

in	*stid*	Thread identifier.

Returns

QURT_EOK – STID set created.
QURT_EFAILED – STID not set.

Dependencies

None.

3.27 qurt_thread_get_running_ids()

3.27.1 Function Documentation

3.27.1.1 int qurt_thread_get_running_ids (qurt_thread_t *)

Returns the thread IDs of the running threads in the system; use only during fatal error handling.

Associated data types

qurt_thread_t

Parameters

in,out	*	Array of thread identifier of size QURT_MAX_HTHREAD_LIMIT + 1.

Returns

QURT_EINVALID – Incorrect argument
QURT_ENOTALLOWED – API not called during error handling
QURT_EOK – Success, returns a NULL-terminated array of thread_id

Dependencies

None.

3.28 qurt_thread_get_thread_id()

3.28.1 Function Documentation

3.28.1.1 int qurt_thread_get_thread_id (qurt_thread_t * *thread_id,* char * *name*)

Gets the thread identifier of the thread with the matching name in the same process of the caller.

Associated data types

qurt_thread_t

Parameters

out	*thread_id*	Pointer to the thread identifier.
in	*name*	Pointer to the name of the thread.

Returns

QURT_EINVALID – No thread with matching name in the process of the caller
QURT_EOK – Success

Dependencies

None.

3.29 qurt_sleep()

3.29.1 Function Documentation

3.29.1.1 void qurt_sleep (unsigned long long int *duration*)

Suspends the current thread for the specified amount of time.

NOTE Becuase QuRT timers are deferrable, this call is guaranteed to block at least for the specified amount of time. If power-collapse is enabled, the maximum amount of time this call can block depends on the earliest wakeup from power-collapse past the specified duration.

Parameters

in	*duration*	Duration (in microseconds) for which the thread is suspended.

Returns

None.

Dependencies

None.

3.30 qurt_system_set_priority_floor()

3.30.1 Function Documentation

3.30.1.1 int qurt_system_set_priority_floor (unsigned int *priority_floor*)

Sets a priority floor to move threads with thread priority lower than the floor out of the running state. Running threads with thread priority lower than the priority floor are moved into the kernel ready queue, and they are not scheduled to run when the thread priority is lower than the floor. Later the caller should reset the priority floor back to the default value of QURT_PRIORITY_FLOOR_DEFAULT. Threads in the kernel ready queue are scheduled to run when the thread priority is higher than the floor.

The priority floor is set and associated to the user process of the caller. When the caller gets into QuRTOS and sets a new floor, the new floor is associated to its original user process, not the QuRTOS process. The floor associated to the user process is reset when the user process exits or is killed, but not at the time when the user thread of the caller exits.

The priority floor cannot be set to a priority higher than the thread priority of the caller.

The priority floor cannot be set to a priority lower than the default QURT_PRIORITY_FLOOR_DEFAULT system floor.

This function is not supported in Island mode.

After the system floor is set above QURT_PRIORITY_FLOOR_DEFAULT, power collapse is skipped, and sleep task is not scheduled to run.

Parameters

in	*priority_floor*	Priority floor.

Returns

QURT_EOK – Success
QURT_ENOTALLOWED – Floor setting is not allowed

Dependencies

None.

3.31 qurt_thread_get_tls_base()

3.31.1 Function Documentation

3.31.1.1 void* qurt_thread_get_tls_base (qurt_tls_info * *info*)

Gets the base address of thread local storage (TLS) of a dynamically loaded module for the current thread.

Associated data types

qurt_tls_info

Parameters

in	*info*	Pointer to the TLS information for a module.

Returns

Pointer to the TLS object for the dynamically loaded module.
NULL – TLS information is invalid.

Dependencies

None.

3.32 qurt_busywait()

3.32.1 Function Documentation

3.32.1.1 void qurt_busywait (unsigned int *pause_time_us*)

Pauses the execution of a thread for a specified time.

Use for small microsecond delays.

NOTE The function does not return to the caller until the time duration has expired.

Parameters

in	*pause_time_us*	Time to pause in microseconds.

Returns

None.

Dependencies

None.

3.33 Data Types

Threads in QuRT are identified by values of type qurt_thread_t.

Thread priorities in QuRT are identified by values of type unsigned short.

Thread attributes in QuRT are stored in structures of type qurt_thread_attr_t.

3.33.1 Define Documentation

3.33.1.1 #define CCCC_PARTITION 0U

Use the CCCC page attribute bits to determine the main or auxiliary partition.

3.33.1.2 #define MAIN_PARTITION 1U

Use the main partition.

3.33.1.3 #define AUX_PARTITION 2U

Use the auxiliary partition.

3.33.1.4 #define MINIMUM_PARTITION 3U

Use the minimum. Allocates the least amount of cache (no-allocate policy possible) for this thread.

3.33.2 Data Structure Documentation

3.33.2.1 struct qurt_thread_attr_t

Thread attributes.

Data fields

Type	Parameter	Description
char	name	Thread name.
unsigned char	tcb_partition	Indicates whether the thread TCB resides in RAM or on chip memory (TCM).
unsigned char	stid	Software thread ID used to configure the stid register for profiling purposes.
unsigned short	priority	Thread priority.
unsigned char	autostack:1	Autostack v2 enabled thread.
unsigned char	reserved:7	Reserved bits.
unsigned char	bus_priority	Internal bus priority.
unsigned short	timetest_id	Timetest ID.
unsigned int	stack_size	Thread stack size.
void *	stack_addr	Pointer to the stack address base. The range of the stack is (stack_addr, stack_addr + stack_size - 1).
unsigned short	detach_state	Detach state of the thread.

3.33.2.2 struct qurt_tls_info

Dynamic TLS attributes.

Data fields

Type	Parameter	Description
unsigned int	module_id	Module ID of the loaded dynamic linked library.
unsigned int	tls_start	Start address of the TLS data.
unsigned int	tls_data_end	End address of the TLS RW data.
unsigned int	tls_end	End address of the TLS data.

3.33.3 Typedef Documentation

3.33.3.1 typedef unsigned int qurt_cache_partition_t

QuRT cache partition type.

3.33.3.2 typedef unsigned int qurt_thread_t

Thread ID type.

3.34 Constants and Macros

This section describes constants for thread services, and macros for thread configuration and QuRT thread attributes.

Bitmask configuration is for selecting DSP hardware threads. To select all the hardware threads, use QURT_THREAD_CFG_BITMASK_ALL.

3.34.1 Define Documentation

3.34.1.1 #define QURT_MAX_HTHREAD_LIMIT 8U

Limit on the maximum number of hardware threads supported by QuRT for any Hexagon version. Use this definition to define arrays, and so on, in target independent code.

3.34.1.2 #define QURT_THREAD_CFG_BITMASK_ALL 0x000000ffU

Select all the hardware threads.

3.34.1.3 #define QURT_THREAD_BUS_PRIO_DISABLED 0

Thread internal bus priority disabled.

3.34.1.4 #define QURT_THREAD_BUS_PRIO_ENABLED 1

Thread internal bus priority enabled.

3.34.1.5 #define QURT_THREAD_AUTOSTACK_DISABLED 0

Thread has autostack v2 feature disabled.

3.34.1.6 #define QURT_THREAD_AUTOSTACK_ENABLED 1

Thread has autostack v2 feature enabled.

3.34.1.7 #define QURT_THREAD_ATTR_CREATE_LEGACY 0U

Create a legacy QuRT thread by default. If a thread has this as a detach state, the thread can be joined on until it exits.

3.34.1.8 #define QURT_THREAD_ATTR_CREATE_JOINABLE 1U

Create a joinable thread.

3.34.1.9 #define QURT_THREAD_ATTR_CREATE_DETACHED 2U

Create a detached thread.

3.34.1.10 #define QURT_THREAD_ATTR_TCB_PARTITION_DEFAULT QURT_THREAD_ATTR_TCB_PARTITION_RAM

Backward compatibility.

3.34.1.11 #define QURT_THREAD_ATTR_PRIORITY_DEFAULT 254

Priority.

3.34.1.12 #define QURT_THREAD_ATTR_ASID_DEFAULT 0

ASID.

3.34.1.13 #define QURT_THREAD_ATTR_AFFINITY_DEFAULT (-1)

Affinity.

3.34.1.14 #define QURT_THREAD_ATTR_BUS_PRIO_DEFAULT 255

Bus priority.

3.34.1.15 #define QURT_THREAD_ATTR_AUTOSTACK_DEFAULT 0

Default autostack v2 disabled thread.

3.34.1.16 #define QURT_THREAD_ATTR_TIMETEST_ID_DEFAULT (-2)

Timetest ID.

3.34.1.17 #define QURT_THREAD_ATTR_STID_DEFAULT 0

STID.

3.34.1.18 #define QURT_PRIORITY_FLOOR_DEFAULT 255U

Default floor.

4 Processes

A process is a grouping of an executable program, an address space, and one or more threads. Each thread in a process shares the process memory area.

A process cannot access the memory in another process, except by using an OS-defined mechanism for resource sharing. QuRT uses the QDI framework to share resources across processes.

Processes are represented as shared objects in QuRT, and have the following attributes:

- Name - Character string identifier for a process object that is already loaded in memory as part of the QuRT system.
- Flags - Bit array used to specify properties of a newly created process. The properties are represented as defined symbols, which map into bits 0-31 of the 32-bit flag value. OR'ing together the individual property symbols specifies multiple properties.

When a process is created, it automatically starts running the code in the specified executable file. An identifier value that identifies the process is assigned to a newly created process.

Process objects support the following QuRT operations:

- qurt_process_attr_init()
- qurt_process_attr_set_executable()
- qurt_process_attr_set_binary_path
- qurt_process_attr_set_dtb_path()
- qurt_process_attr_set_flags()
- qurt_process_cmdline_get()
- qurt_process_create()
- qurt_process_get_id()
- qurt_process_attr_set_max_threads()
- qurt_process_attr_set_sw_id()
- qurt_process_attr_set_ceiling_prio()
- qurt_process_get_thread_count()
- qurt_process_get_thread_ids()
- qurt_process_attr_get()
- qurt_process_dump_register_cb()
- qurt_process_dump_deregister_cb()

- qurt_process_exit()
- qurt_process_kill()
- qurt_debugger_register_process()
- qurt_debugger_deregister_process()
- qurt_process_exec_callback()
- qurt_process_get_pid()
- qurt_process_get_dm_status()
- Data Types

4.1 qurt_process_attr_init()

4.1.1 Function Documentation

4.1.1.1 static void qurt_process_attr_init (qurt_process_attr_t * *attr*)

Initializes the structure that sets the process attributes when a thread is created.

After an attribute structure is initialized, the individual attributes in the structure can be explicitly set using the process attribute operations.

Table 4-1 lists the default attribute values set by the initialize operation.

Table 4-1 Process attribute defaults

Attribute	Default value
Name	Null string
Flags	0

Associated data types

qurt_process_attr_t

Parameters

`out`	*attr*	Pointer to the structure to initialize.

Returns

None.

Dependencies

None.

4.2 qurt_process_attr_set_executable()

4.2.1 Function Documentation

4.2.1.1 void qurt_process_attr_set_executable (qurt_process_attr_t * *attr,* const char * *name*)

Sets the process name in the specified process attribute structure.

Process names identify process objects that are already loaded in memory as part of the QuRT system.

NOTE Process objects are incorporated into the QuRT system at build time.

NOTE Maximum length of name string is limited to QURT_PROCESS_ATTR_NAME_MAXLEN - 1.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*name*	Pointer to the process name.

Returns

None.

Dependencies

None.

4.3 qurt_process_attr_set_binary_path

4.3.1 Function Documentation

4.3.1.1 void qurt_process_attr_set_binary_path (qurt_process_attr_t * *attr,* char * *path*)

Sets the binary path for the process loading in the specified process attribute structure.

Path specifies the binary to load for this process.

NOTE Max length of path string is limited to QURT_PROCESS_ATTR_BIN_PATH_MAXLEN-1.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*path*	Pointer to the binary path.

Returns

None.

Dependencies

None.

4.4 qurt_process_attr_set_dtb_path()

4.4.1 Function Documentation

4.4.1.1 void qurt_process_attr_set_dtb_path (qurt_process_attr_t * *attr,* char * *path*)

Sets the DTB binary path for the process loading in the specified process attribute structure.

Path specifies the DTB binary to load for this process.

NOTE Max length of path string is limited to QURT_PROCESS_ATTR_BIN_PATH_MAXLEN-1.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*path*	Pointer to the binary path.

Returns

None.

Dependencies

None.

4.5 qurt_process_attr_set_flags()

4.5.1 Function Documentation

4.5.1.1 static void qurt_process_attr_set_flags (qurt_process_attr_t * *attr,* int *flags*)

Sets the process properties in the specified process attribute structure. Process properties are represented as defined symbols that map into bits 0 through 31 of the 32-bit flag value. Multiple properties are specified by OR'ing together the individual property symbols.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*flags*	QURT_PROCESS_NON_SYSTEM_CRITICAL Process is considered as non system-critical. This attribute will be used by error services, to decide whether to kill user pd or whole subsystem. QURT_PROCESS_ISLAND_RESIDENT Process will be marked as island resident. QURT_PROCESS_RESTARTABLE Process will be marked as restartable. QURT_PROCESS_UNTRUSTED Process will be marked as unsigned process.

Returns

None.

Dependencies

None.

4.6 qurt_process_cmdline_get()

4.6.1 Function Documentation

4.6.1.1 void qurt_process_cmdline_get (char * *buf,* unsigned *buf_siz*)

Gets the command line string associated with the current process. The Hexagon simulator command line arguments are retrieved using this function as long as the call is made in the process of the QuRT installation, and with the requirement that the program runs in a simulation environment.

If the function modifies the provided buffer, it zero-terminates the string. It is possible that the function does not modify the provided buffer, so the caller must set buf[0] to a NULL byte before making the call. A truncated command line is returned when the command line is longer than the provided buffer.

Parameters

in	*buf*	Pointer to a character buffer that must be filled in.
in	*buf_siz*	Size (in bytes) of the buffer pointed to by the buf argument.

Returns

None.

Dependencies

None.

4.7 qurt_process_create()

4.7.1 Function Documentation

4.7.1.1 int qurt_process_create (qurt_process_attr_t * *attr*)

Creates a process with the specified attributes, and starts the process.

The process executes the code in the specified ELF.

Associated data types

qurt_process_attr_t

Parameters

out	*attr*	Accepts an initialized process attribute structure, which specifies the attributes of the created process.

Returns

In case of negative return value,error is returned: QURT_EPRIVILEGE – Caller does not have privilege for this operation
QURT_EMEM – Not enough memory to perform the operation
QURT_EFAILED – Operation failed
QURT_ENOTALLOWED – Operation not allowed
QURT_ENOREGISTERED – Not registered
QURT_ENORESOURCE – Resource exhaustion
QURT_EINVALID – Invalid argument value

In case of positive return value, Process ID is returned.

Dependencies

None.

4.8 qurt_process_get_id()

4.8.1 Function Documentation

4.8.1.1 int qurt_process_get_id (void)

Returns the process identifier for the current thread.

Returns

None.

Dependencies

Process identifier for the current thread.

4.9 qurt_process_attr_set_max_threads()

4.9.1 Function Documentation

4.9.1.1 static void qurt_process_attr_set_max_threads (qurt_process_attr_t * *attr,* unsigned *max_threads*)

Sets the maximum number of threads allowed in the specified process attribute structure.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*max_threads*	Maximum number of threads allowed for this process.

Returns

None.

Dependencies

None.

4.10 qurt_process_attr_set_sw_id()

4.10.1 Function Documentation

4.10.1.1 static void qurt_process_attr_set_sw_id (qurt_process_attr_t * *attr,* unsigned int *sw_id*)

Sets the software ID of the process to load in the specified process attribute structure.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*sw_id*	Software ID of the process.

Returns

None.

Dependencies

None.

4.11 qurt_process_attr_set_ceiling_prio()

4.11.1 Function Documentation

4.11.1.1 static void qurt_process_attr_set_ceiling_prio (qurt_process_attr_t * *attr,* unsigned short *prio*)

Sets the highest thread priority allowed in the specified process attribute structure. Refer qurt_thread.h for priority ranges.

Associated data types

qurt_process_attr_t

Parameters

in	*attr*	Pointer to the process attribute structure.
in	*prio*	Priority.

Returns

None.

Dependencies

None.

4.12 qurt_process_get_thread_count()

4.12.1 Function Documentation

4.12.1.1 int qurt_process_get_thread_count (unsigned int *pid*)

Gets the number of threads present in the process indicated by the PID.

Parameters

in	*pid*	PID of the process for which the information is required.

Returns

Number of threads in the process indicated by PID, if positive value is obtained Negative error code if failed include: QURT_EFATAL - Invalid PID -QURT_ENOTALLOWED - Current process doesnt have access to target process indicated by PID

Dependencies

None.

4.13 qurt_process_get_thread_ids()

4.13.1 Function Documentation

4.13.1.1 int qurt_process_get_thread_ids (unsigned int *pid,* unsigned int * *ptr,* unsigned *thread_num*)

Gets the thread IDs for a process indicated by PID.

Parameters

in	*pid*	PID of the process for which the information is required.
in	*ptr*	Pointer to a user passed buffer that must be filled in with thread IDs.
in	*thread_num*	Number of thread IDs requested.

Returns

None.

Dependencies

None.

4.14 qurt_process_attr_get()

4.14.1 Function Documentation

4.14.1.1 int qurt_process_attr_get (unsigned int *pid,* qurt_process_attr_t * *attr*)

Gets the attributes of the process with which it was created.

Associated data types

qurt_process_attr_t

Parameters

in	*pid*	PID of the process for which the information is required.
in,out	*attr*	Pointer to the user allocated attribute structure.

Returns

None.

Dependencies

None.

4.15 qurt_process_dump_register_cb()

4.15.1 Function Documentation

4.15.1.1 int qurt_process_dump_register_cb (qurt_cb_data_t * *cb_data,* qurt_process_dump_cb_type_t *type,* unsigned short *priority*)

Registers the process domain dump callback.

Associated data types

qurt_cb_data_t
qurt_process_dump_cb_type_t

Parameters

in	*cb_data*	Pointer to the callback information.
in	*type*	Callback type; these callbacks are called in the context of the user process domain: QURT_PROCESS_DUMP_CB_PRESTM – Before threads of the exiting process are frozen. QURT_PROCESS_DUMP_CB_ERROR – After threads are frozen and captured. QURT_PROCESS_DUMP_CB_ROOT – After threads are frozen and captured, and CB_ERROR type of callbacks are called.
in	*priority*	Priority.

Returns

QURT_EOK – Success
Other values – Failure

Dependencies

None.

4.16 qurt_process_dump_deregister_cb()

4.16.1 Function Documentation

4.16.1.1 int qurt_process_dump_deregister_cb (qurt_cb_data_t * *cb_data,* qurt_process_dump_cb_type_t *type*)

Deregisters the process domain dump callback.

Associated data types

qurt_cb_data_t
qurt_process_dump_cb_type_t

Parameters

in	*cb_data*	Pointer to the callback information to deregister.
in	*type*	Callback type.

Returns

QURT_EOK – Success.
Other values – Failure.

Dependencies

None.

4.17 qurt_process_exit()

4.17.1 Function Documentation

4.17.1.1 int qurt_process_exit (int *exitcode*)

Exits the xurrent user process with an exit code.

Parameters

in	*exitcode*	Exit code.

Returns

QURT_EFATAL – No client found with the specified PID value
QURT_EINVALID – Invalid client
QURT_ENOTALLOWED – User does not have permission to perform this operation
QURT_EOK – Success

Dependencies

None.

4.18 qurt_process_kill()

4.18.1 Function Documentation

4.18.1.1 int qurt_process_kill (int *pid*, int *exitcode*)

Kills the process represented by the PID with the exit code.

Parameters

in	*pid*	PID of the process to kill.
in	*exitcode*	Exit code.

Returns

QURT_EFATAL – No client found with the specified PID value
QURT_EINVALID – Invalid client
QURT_ENOTALLOWED – User does not have permission to perform this operation
QURT_EOK – Success

Dependencies

None.

4.19 qurt_debugger_register_process()

4.19.1 Function Documentation

4.19.1.1 int qurt_debugger_register_process (int *pid,* unsigned int *adr*)

Registers the process indicated by the PID with the debug monitor.

Parameters

in	*pid*	PID of the process.
in	*adr*	Address.

Returns

QURT_EOK – Success

Dependencies

None.

4.20 qurt_debugger_deregister_process()

4.20.1 Function Documentation

4.20.1.1 int qurt_debugger_deregister_process (int *pid*)

Deregister the process indicated by the PID with the debug monitor.

Parameters

in	*pid*	PID of the process.

Returns

QURT_EOK – Success

Dependencies

None.

4.21 qurt_process_exec_callback()

4.21.1 Function Documentation

4.21.1.1 int qurt_process_exec_callback (int *client_handle,* unsigned *callback_fn,* unsigned *stack_base,* unsigned *stack_size*)

Executes callbacks in the user process as indicated by the client_handle argument.

Parameters

in	*client_handle*	Client handle obtained from the current invocation function (Section 3.4.1).
in	*callback_fn*	Callback function to execute.
in	*stack_base*	Stack address to use.
in	*stack_size*	Stack size.

Returns

QURT_EOK – Success

Dependencies

None.

4.22 qurt_process_get_pid()

4.22.1 Function Documentation

4.22.1.1 int qurt_process_get_pid (int *client_handle,* int * *pid*)

Gets the process ID of the process that the client_handle argument represents.

Parameters

in	*client_handle*	Client handle obtained from the current invocation function (Section 3.4.1).
in	*pid*	Pointer to the address to store the PID.

Returns

QURT_EOK – Success

Dependencies

None.

4.23 qurt_process_get_dm_status()

4.23.1 Function Documentation

4.23.1.1 int qurt_process_get_dm_status (unsigned int *pid,* unsigned int * *status*)

Gets the debugging session status on the process represented by the pid argument.

Parameters

in	*pid*	Process ID
in,out	*status*	Address to store the status: QURT_DEBUG_NOT_START QURT_DEBUG_START

Returns

QURT_EOK - Success
QURT_EINVALID - Error

Dependencies

None.

4.24 Data Types

4.24.1 Define Documentation

4.24.1.1 #define QURT_PROCESS_ATTR_NAME_MAXLEN QURT_MAX_NAME_LEN

Maximum length of the process name.

4.24.1.2 #define QURT_PROCESS_ATTR_BIN_PATH_MAXLEN 128

Maximum length of the path of binary/ELF for this process.

4.24.1.3 #define QURT_PROCESS_DEFAULT_CEILING_PRIO 0

Default ceiling priority of the threads in the new process.

4.24.1.4 #define QURT_PROCESS_DEFAULT_MAX_THREADS -1

Default number of threads in the new process. -1 indicates that the limit is set to the maximum supported by the system.

4.24.1.5 #define QURT_PROCESS_NON_SYSTEM_CRITICAL (1u << 1)

Starts the new process as non system-critical.

4.24.1.6 #define QURT_PROCESS_ISLAND_RESIDENT (1u << 2)

Process is island resident.

4.24.1.7 #define QURT_PROCESS_RESTARTABLE (1u << 3)

Indicates that the process is restartable

4.24.1.8 #define QURT_PROCESS_UNTRUSTED (1u << 7)

Starts the new process as unsigned process.

4.24.1.9 #define QURT_DEBUG_NOT_START 0

Debug is not started.

4.24.1.10 #define QURT_DEBUG_START 1

Debug has started.

4.24.1.11 #define QURT_DEBUG_START 1

Debug has started.

4.24.1.12 #define QURT_PROCESS_SUSPEND_DEFAULT 0

Process Suspend Options

4.24.1.13 #define QURT_PROCESS_RESUME_DEFAULT 0

Process Resume Options

4.24.2 Data Structure Documentation

4.24.2.1 struct qurt_pd_dump_attr_t

QuRT process dump attributes.

4.24.2.2 struct qurt_process_attr_t

QuRT process attributes.

4.24.3 Enumeration Type Documentation

4.24.3.1 enum qurt_process_type_t

Enumerator:

QURT_PROCESS_TYPE_RESERVED Process type is reserved.
QURT_PROCESS_TYPE_KERNEL Kernel process.
QURT_PROCESS_TYPE_SRM SRM process.
QURT_PROCESS_TYPE_SECURE Secure process.
QURT_PROCESS_TYPE_ROOT Root process.
QURT_PROCESS_TYPE_USER User process.

4.24.3.2 enum qurt_process_dump_cb_type_t

QuRT process callback types.

Enumerator:

QURT_PROCESS_DUMP_CB_ROOT Register the callback that executes in the root process context.
QURT_PROCESS_DUMP_CB_ERROR Register the user process callback that is called after threads in the process are frozen.
QURT_PROCESS_DUMP_CB_PRESTM Register the user process callback that is called before threads in the process are frozen.
QURT_PROCESS_DUMP_CB_MAX Reserved for error checking.

5 Mutexes

Threads use mutexes (short for mutual exclusion) to synchronize their execution to ensure mutually exclusive access to shared resources.

If a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing.

If a thread performs a lock operation on a mutex that is already in use by another thread, the thread is suspended on the mutex. When the mutex becomes available (because the other thread has unlocked it), the suspended thread is awakened and gains access to the shared resource.

More than one thread can be suspended on a mutex. When the mutex is unlocked, only the highest-priority thread waiting on the mutex is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

Figure 5-1 shows an example of using mutexes.

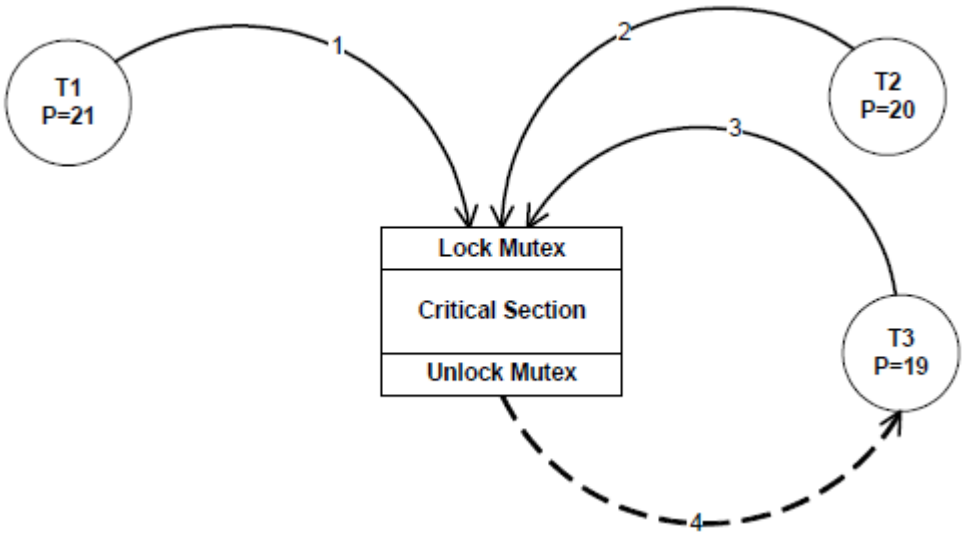

Figure 5-1 Mutex example

In figure 5-1 the following sequence of events occurs:

1. Thread T1 successfully locks the mutex and enters the critical section of code that is protected by the mutex.
2. Thread T2 attempts to lock the mutex but is blocked by T1. T2 is suspended on the mutex.
3. Thread T3 also tries to lock the mutex and it too is suspended. Because the thread priority of T3 is higher than the priority of T2, T3 is inserted into the mutex wait queue ahead of T2.
4. T1 exits the critical section and unlocks the mutex. T3 is selected from the mutex wait queue and awakened (because it is the highest-priority thread waiting on the mutex). Because T3 has higher priority than T1 (19 versus 21 respectively), T1 is suspended and T3 resumes execution, locking the mutex and entering the critical section.

The try lock operation enables a thread to try locking a mutex without the risk of getting suspended if the mutex is already locked:

- If the mutex is unlocked, try lock is identical to the regular lock operation.
- If the mutex is locked, try lock returns with a value indicating the locked state.

Mutexes are shared objects which support the following operations:

- qurt_mutex_destroy()
- qurt_mutex_init()
- qurt_mutex_lock()
- qurt_mutex_try_lock()
- qurt_mutex_unlock()
- qurt_mutex_lock_timed()
- Data Types

5.1 qurt_mutex_destroy()

5.1.1 Function Documentation

5.1.1.1 void qurt_mutex_destroy (qurt_mutex_t * *lock*)

Destroys the specified mutex.

NOTE Mutexes must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Mutexes must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the mutex object to destroy.

Returns

None.

Dependencies

None.

5.2 qurt_mutex_init()

5.2.1 Function Documentation

5.2.1.1 void qurt_mutex_init (qurt_mutex_t * *lock*)

Initializes a mutex object. The mutex is initially unlocked.

NOTE Each mutex-based object has one or more kernel resources associated with it; to prevent resource leaks, call qurt_mutex_destroy() when this object is not used anymore

Associated data types

qurt_mutex_t

Parameters

out	*lock*	Pointer to the mutex object. Returns the initialized object.

Returns

None.

Dependencies

None.

5.3 qurt_mutex_lock()

5.3.1 Function Documentation

5.3.1.1 void qurt_mutex_lock (qurt_mutex_t * *lock*)

Locks the specified mutex. If a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing.

If a thread performs a lock operation on a mutex that is already in use by another thread, the thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the thread is awakened and given access to the shared resource.

NOTE A thread is suspended indefinitely if it locks a mutex that it has already locked. Avoid this by using recursive mutexes (Section 6).

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the mutex object. Specifies the mutex to lock.

Returns

None.

Dependencies

None.

5.4 qurt_mutex_try_lock()

5.4.1 Function Documentation

5.4.1.1 int qurt_mutex_try_lock (qurt_mutex_t * *lock*)

Attempts to lock the specified mutex. If a thread performs a try_lock operation on a mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing.

NOTE If a thread performs a try_lock operation on a mutex that it has already locked or is in use by another thread, qurt_mutex_try_lock immediately returns with a nonzero result value.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the mutex object. Specifies the mutex to lock.

Returns

0 – Success.
Nonzero – Failure.

Dependencies

None.

5.5 qurt_mutex_unlock()

5.5.1 Function Documentation

5.5.1.1 void qurt_mutex_unlock (qurt_mutex_t * *lock*)

Unlocks the specified mutex.

More than one thread can be suspended on a mutex. When the mutex is unlocked, only the highest-priority thread waiting on the mutex is awakened. If the awakened thread has higher priority than the current thread, a context switch occurs.

NOTE The behavior of QuRT is undefined if a thread unlocks a mutex it did not first lock.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the mutex object. Specifies the mutex to unlock.

Returns

None.

Dependencies

None.

5.6 qurt_mutex_lock_timed()

5.6.1 Function Documentation

5.6.1.1 int qurt_mutex_lock_timed (qurt_mutex_t * *lock*, unsigned long long int *duration*)

Locks the specified mutex. When a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing.

When a thread performs a lock operation on a mutex that is already in use by another thread, the thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the thread is awakened and given access to the shared resource. If the duration of suspension exceeds the timeout duration, wait is terminated and no access to mutex is granted.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the mutex object; specifies the mutex to lock.
in	*duration*	Interval (in microseconds) that the duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION

Returns

QURT_EOK – Success
QURT_ETIMEDOUT – Timeout

Dependencies

None.

5.7 Data Types

Mutexes are represented in QuRT as objects of type qurt_mutex_t.

5.7.1 Data Structure Documentation

5.7.1.1 union qurt_mutex_t

QuRT mutex type.

Both non-recursive mutex lock and unlock, and recursive mutex lock and unlock can be applied to this type.

6 Recursive Mutexes

QuRT supports a variant of mutexes known as recursive mutexes. Recursive mutexes are functionally equivalent to regular mutexes (Section 5), except that they enable a thread to perform nested locking on a mutex:

- If a thread performs a lock operation on a recursive mutex that is already in use by another thread, the thread is suspended.
- If a thread performs a lock on a recursive mutex that is already in use by itself, the operation is treated as a nested lock and the thread continues executing as if the mutex is unlocked. However, the recursive mutex does not become available again until the thread performs a balanced number of unlocks on it.

The regular and recursive mutex operations are identical except for the change within the function names from mutex to rmutex.

Note: With recursive mutexes, the try lock operation handles a nested lock as if the mutex is unlocked.

Recursive mutexes are shared objects that support the following operations:

- qurt_rmutex_destroy()
- qurt_rmutex_init()
- qurt_rmutex_lock()
- qurt_rmutex_try_lock()
- qurt_rmutex_unlock()
- qurt_rmutex_lock_timed()
- qurt_rmutex_try_lock_block_once()

6.1 qurt_rmutex_destroy()

6.1.1 Function Documentation

6.1.1.1 void qurt_rmutex_destroy (qurt_mutex_t * *lock*)

Destroys the specified recursive mutex.

NOTE Recursive mutexes must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the recursive mutex object to destroy.

Returns

None.

Dependencies

None.

6.2 qurt_rmutex_init()

6.2.1 Function Documentation

6.2.1.1 void qurt_rmutex_init (qurt_mutex_t * *lock*)

Initializes a recursive mutex object. The recursive mutex is initialized in unlocked state.

Associated data types

qurt_mutex_t

Parameters

out	*lock*	Pointer to the recursive mutex object.

Returns

None.

Dependencies

None.

6.3 qurt_rmutex_lock()

6.3.1 Function Documentation

6.3.1.1 void qurt_rmutex_lock (qurt_mutex_t * *lock*)

Locks the specified recursive mutex.

If a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that the mutex protects, and continues executing.

If a thread performs a lock operation on a mutex that is already use by another thread, the thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the thread is awakened and given access to the shared resource.

NOTE A thread is not suspended if it locks a recursive mutex that it has already locked. However, the mutex does not become available to other threads until the thread performs a balanced number of unlocks on the mutex.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the recursive mutex object to lock.

Returns

None.

Dependencies

None.

6.4 qurt_rmutex_try_lock()

6.4.1 Function Documentation

6.4.1.1 int qurt_rmutex_try_lock (qurt_mutex_t * *lock*)

Attempts to lock the specified recursive mutex.

If a thread performs a try_lock operation on a recursive mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing.

If a thread performs a try_lock operation on a recursive mutex that another thread has already locked, qurt_rmutex_try_lock immediately returns with a nonzero result value.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the recursive mutex object to lock.

Returns

0 – Success.
Nonzero – Failure.

6.5 qurt_rmutex_unlock()

6.5.1 Function Documentation

6.5.1.1 void qurt_rmutex_unlock (qurt_mutex_t * *lock*)

Unlocks the specified recursive mutex.

More than one thread can be suspended on a mutex. When the mutex is unlocked, the thread waiting on the mutex awakens. If the awakened thread has higher priority than the current thread, a context switch occurs.

NOTE When a thread unlocks a recursive mutex, the mutex is not available until the balanced number of locks and unlocks has been performed on the mutex.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the recursive mutex object to unlock.

Returns

None.

Dependencies

None.

6.6 qurt_rmutex_lock_timed()

6.6.1 Function Documentation

6.6.1.1 int qurt_rmutex_lock_timed (qurt_mutex_t * *lock*, unsigned long long int *duration*)

Locks the specified recursive mutex. The wait must be terminated when the specified timeout expires.

If a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that the mutex is protecting, and continues executing.

If a thread performs a lock operation on a mutex that is already in use by another thread, the thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the thread is awakened and given access to the shared resource.

NOTE A thread is not suspended if it locks a recursive mutex that it has already locked by itself. However, the mutex does not become available to other threads until the thread performs a balanced number of unlocks on the mutex. If timeout expires, this wait must be terminated and no access to the mutex is granted.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the recursive mutex object to lock.
in	*duration*	Interval (in microseconds) duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION

Returns

QURT_EOK – Success
QURT_ETIMEDOUT – Timeout

Dependencies

None.

6.7 qurt_rmutex_try_lock_block_once()

6.7.1 Function Documentation

6.7.1.1 int qurt_rmutex_try_lock_block_once (qurt_mutex_t * *lock*)

Attempts to lock a mutex object recursively. If the mutex is available, it locks the mutex. If the mutex is held by the current thread, it increases the internal counter and returns 0. If not, it returns a nonzero value. If the mutex is already locked by another thread, the caller thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the caller thread is awakened and tries to lock the mutex; and if it fails, this function returns failure with a nonzero value. If it succeeds, this function returns success with zero.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the qurt_mutex_t object.

Returns

0 – Success.
Nonzero – Failure.

Dependencies

None.

7 Priority Inheritance Mutexes

QuRT supports a variant of recursive mutexes known as priority inheritance mutexes.

Priority inheritance mutexes are functionally equivalent to recursive mutexes (Section 6), except that they enable a thread to perform priority inheritance after locking a mutex:

- If a thread has locked a priority inheritance mutex, and another thread with higher priority (Section 3) becomes suspended on the mutex, the thread with the lock acquires the higher priority of the suspended thread.
- If multiple threads are suspended on a priority inheritance mutex, the thread with the lock acquires the priority of the highest-priority suspended thread (if it is higher than the thread's original priority.

The change in priority is temporary – when a thread unlocks a priority inheritance mutex, its thread priority is restored to its original value.

The regular and priority inheritance mutex operations are identical except for the change within the function names from mutex to pimutex.

Note: With priority inheritance mutexes, the try lock operation handles a nested lock as if the mutex is unlocked.

Priority inheritance mutexes are shared objects that support the following operations:

- qurt_pimutex_init()
- qurt_pimutex_destroy()
- qurt_pimutex_lock
- qurt_pimutex_try_lock()
- qurt_pimutex_unlock()

7.1 qurt_pimutex_init()

7.1.1 Function Documentation

7.1.1.1 void qurt_pimutex_init (qurt_mutex_t * *lock*)

Initializes a priority inheritance mutex object. The priority inheritance mutex is initially unlocked.

This function works the same as qurt_mutex_init().

NOTE Each pimutex-based object has one or more kernel resources associated with it; to prevent resource leaks, call qurt_pimutex_destroy() when this object is not used anymore

Associated data types

qurt_mutex_t

Parameters

out	*lock*	Pointer to the priority inheritance mutex object.

Returns

None.

Dependencies

None.

7.2 qurt_pimutex_destroy()

7.2.1 Function Documentation

7.2.1.1 void qurt_pimutex_destroy (qurt_mutex_t * *lock*)

Destroys the specified priority inheritance mutex.

NOTE Priority inheritance mutexes must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Priority inheritance mutexes must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the priority inheritance mutex object to destroy.

Returns

None.

Dependencies

None.

7.3 qurt_pimutex_lock

7.3.1 Function Documentation

7.3.1.1 void qurt_pimutex_lock (qurt_mutex_t * *lock*)

Requests access to a shared resources. If a thread performs a lock operation on a mutex that is not in use, the thread gains access to the shared resource that the mutex protects, and continues executing.

If a thread performs a lock operation on a mutex that is already in use by another thread, the thread is suspended. When the mutex becomes available again (because the other thread has unlocked it), the thread is awakened and given access to the shared resource.

If a thread is suspended on a priority inheritance mutex, and the priority of the suspended thread is higher than the priority of the thread that has locked the mutex, the thread with the mutex acquires the higher priority of the suspended thread. The locker thread blocks until the lock is available.

NOTE A thread is not suspended if it locks a priority inheritance mutex that it has already locked . However, the mutex does not become available to other threads until the thread performs a balanced number of unlocks on the mutex.

When multiple threads compete for a mutex, the lock operation for a priority inheritance mutex is slower than it is for a recursive mutex. In particular, it is about 10 times slower when the mutex is available for locking, and slower (with greatly varying times) when the mutex is already locked.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the priority inheritance mutex object to lock.

Returns

None.

Dependencies

None.

7.4 qurt_pimutex_try_lock()

7.4.1 Function Documentation

7.4.1.1 int qurt_pimutex_try_lock (qurt_mutex_t * *lock*)

Request access to a shared resource (without suspend). Attempts to lock the specified priority inheritance mutex.

If a thread performs a try_lock operation on a priority inheritance mutex that is not in use, the thread gains access to the shared resource that is protected by the mutex, and continues executing. If a thread performs a try_lock operation on a priority inheritance mutex that is already in use by another thread, qurt_pimutex_try_lock immediately returns with a nonzero result value.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the priority inheritance mutex object to lock.

Returns

0 – Success.
Nonzero – Failure.

Dependencies

None.

7.5 qurt_pimutex_unlock()

7.5.1 Function Documentation

7.5.1.1 void qurt_pimutex_unlock (qurt_mutex_t * *lock*)

Releases access to a shared resource; unlocks the specified priority inheritance mutex.

More than one thread can be suspended on a priority inheritance mutex. When the mutex is unlocked, only the highest-priority thread waiting on the mutex is awakened. If the awakened thread has higher priority than the current thread, a context switch occurs.

When a thread unlocks a priority inheritance mutex, its thread priority is restored to its original value from any higher priority value that it acquired from another thread suspended on the mutex.

Associated data types

qurt_mutex_t

Parameters

in	*lock*	Pointer to the priority inheritance mutex object to unlock.

Returns

None.

Dependencies

None.

8 Signals

Threads use signals to synchronize their execution based on the occurrence of one or more internal events.

If a thread waits on a signal object for any of the specified set of signals to be set, and one or more of those signals is set in the signal object, the thread is awakened.

If a thread waits on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

A signal object contains 32 signals, which are represented as bits 0 through 31 in a 32-bit value. The bit value 1 indicates that a signal is set, and 0 indicates that it is cleared.

Note: At most, one thread can wait on a signal object at any given time.

The qurt_signal_wait() and qurt_signal_wait_cancellable() functions wait for any or all signals, depending on its wait type argument. Signals are stored in shared objects that support the following operations:

- qurt_signal_clear()
- qurt_signal_destroy()
- qurt_signal_get()
- qurt_signal_init()
- qurt_signal_set()
- qurt_signal_wait()
- qurt_signal_wait_all()
- qurt_signal_wait_any()
- qurt_signal_wait_cancellable()
- qurt_signal_wait_timed()
- qurt_signal_64_init()
- qurt_signal_64_destroy()
- qurt_signal_64_wait()
- qurt_signal_64_set()
- qurt_signal_64_get()
- qurt_signal_64_clear()
- Data Types

8.1 qurt_signal_clear()

8.1.1 Function Documentation

8.1.1.1 void qurt_signal_clear (qurt_signal_t * *signal,* unsigned int *mask*)

Clear signals in the specified signal object.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be cleared, and 0 indicates not to clear it.

NOTE Signals must be explicitly cleared by a thread when it is awakened – the wait operations do not automatically clear them.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to modify.
in	*mask*	Mask value identifying the individual signals to clear in the signal object.

Returns

None.

Dependencies

None.

8.2 qurt_signal_destroy()

8.2.1 Function Documentation

8.2.1.1 void qurt_signal_destroy (qurt_signal_t * *signal*)

Destroys the specified signal object.

NOTE Signal objects must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Signal objects must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_signal_t

Parameters

in	*signal	Pointer to the signal object to destroy.

Returns

None.

Dependencies

None.

8.3 qurt_signal_get()

8.3.1 Function Documentation

8.3.1.1 unsigned int qurt_signal_get (qurt_signal_t * *signal*)

Gets a signal from a signal object.

Returns the current signal values of the specified signal object.

Associated data types

qurt_signal_t

Parameters

in	*signal	Pointer to the signal object to access.

Returns

A 32-bit word with current signals

Dependencies

None.

8.4 qurt_signal_init()

8.4.1 Function Documentation

8.4.1.1 void qurt_signal_init (qurt_signal_t * *signal*)

Initializes a signal object. Signal returns the initialized object. The signal object is initially cleared.

NOTE Each signal-based object has one or more kernel resources associated with it; to prevent resource leaks, call qurt_signal_destroy() when this object is not used anymore

Associated data types

qurt_signal_t

Parameters

in	*signal	Pointer to the initialized object.

Returns

None.

Dependencies

None.

8.5 qurt_signal_set()

8.5.1 Function Documentation

8.5.1.1 void qurt_signal_set (qurt_signal_t * *signal,* unsigned int *mask*)

Sets signals in the specified signal object.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates to set the signal, and 0 indicates not to set it.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to modify.
in	*mask*	Mask value identifying the individual signals to set in the signal object.

Returns

None.

Dependencies

None.

8.6 qurt_signal_wait()

8.6.1 Function Documentation

8.6.1.1 unsigned int qurt_signal_wait (qurt_signal_t * *signal*, unsigned int *mask*, unsigned int *attribute*)

Suspends the current thread until the specified signals are set.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates waiting on a signal, and 0 indicates not waiting on the signal.

If a thread waits on a signal object for any of the specified set of signals to be set, and one or more of those signals is set in the signal object, the thread is awakened.

If a thread is waiting on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

The specified set of signals can be cleared when the signal is set.

NOTE At most, one thread can wait on a signal object at any given time.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value identifying the individual signals in the signal object to wait on.
in	*attribute*	Indicates whether the thread waits to set any of the signals, or to set all of them. **NOTE** The wait-any and wait-all types are mutually exclusive. Values: • QURT_SIGNAL_ATTR_WAIT_ANY • QURT_SIGNAL_ATTR_WAIT_ALL

Returns

A 32-bit word with current signals.

Dependencies

None.

8.7 qurt_signal_wait_all()

8.7.1 Function Documentation

8.7.1.1 static unsigned int qurt_signal_wait_all (qurt_signal_t * *signal,* unsigned int *mask*)

Suspends the current thread until all of the specified signals are set.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates to wait on a signal, and 0 indicates not to wait on it.

If a thread is waiting on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

NOTE At most, one thread can wait on a signal object at any given time.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value identifying the individual signals in the signal object to wait on.

Returns

A 32-bit word with current signals.

Dependencies

None.

8.8 qurt_signal_wait_any()

8.8.1 Function Documentation

8.8.1.1 static unsigned int qurt_signal_wait_any (qurt_signal_t * *signal,* unsigned int *mask*)

Suspends the current thread until any of the specified signals are set.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates to wait on a signal, and 0 indicates not to wait on the thread.

If a thread is waiting on a signal object for any of the specified set of signals to be set, and one or more of those signals is set in the signal object, the thread is awakened.

NOTE At most, one thread can wait on a signal object at any given time.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value identifying the individual signals in the signal object to wait on.

Returns

32-bit word with current signals.

Dependencies

None.

8.9 qurt_signal_wait_cancellable()

8.9.1 Function Documentation

8.9.1.1 int qurt_signal_wait_cancellable (qurt_signal_t * *signal,* unsigned int *mask,* unsigned int *attribute,* unsigned int * *return_mask*)

Suspends the current thread until either the specified signals are set or the wait operation is cancelled. The operation is cancelled if the user process of the calling thread is killed, or if the calling thread must finish its current QDI invocation and return to user space.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not to wait on it.

If a thread is waiting on a signal object for any of the specified set of signals to be set, and one or more of those signals is set in the signal object, the thread is awakened.

If a thread is waiting on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

NOTE At most, one thread can wait on a signal object at any given time.

When the operation is cancelled, the caller must assume that the signal is never set.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value identifying the individual signals in the signal object to wait on.
in	*attribute*	Indicates whether the thread must wait until any of the signals are set, or until all of them are set. Values: • QURT_SIGNAL_ATTR_WAIT_ANY • QURT_SIGNAL_ATTR_WAIT_ALL
out	*return_mask*	Pointer to the 32-bit mask value that was originally passed to the function.

Returns

QURT_EOK – Wait completed.
QURT_ECANCEL – Wait cancelled.

Dependencies

None.

8.10 qurt_signal_wait_timed()

8.10.1 Function Documentation

8.10.1.1 int qurt_signal_wait_timed (qurt_signal_t * *signal*, unsigned int *mask*, unsigned int *attribute*, unsigned int * *signals*, unsigned long long int *duration*)

Suspends the current thread until the specified signals are set or until timeout.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates waiting on a signal, and 0 indicates not waiting.

If a thread is waiting on a signal object for any of the specified set of signals to be set, and one or more of those signals is set in the signal object, the thread is awakened.

If a thread is waiting on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

The specified set of signals can be cleared after the signal is set.

NOTE At most, one thread can wait on a signal object at any given time.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value that identifies the individual signals in the signal object to wait on.
in	*attribute*	Indicates whether the thread must wait until any of the signals are set, or until all of them are set. **NOTE** The wait-any and wait-all types are mutually exclusive. Values: • QURT_SIGNAL_ATTR_WAIT_ANY • QURT_SIGNAL_ATTR_WAIT_ALL
out	*signals*	Bitmask of signals that are set
in	*duration*	Duration (microseconds) to wait. Must be in the range [QURT_TIMER_MIN_DURATION ... QURT_TIMER_MAX_DURATION]

Returns

QURT_EOK – Success; one or more signals were set
QURT_ETIMEDOUT – Timed-out
QURT_EINVALID – Duration out of range

Dependencies

Timed-waiting support in the kernel.

8.11 qurt_signal_64_init()

8.11.1 Function Documentation

8.11.1.1 void qurt_signal_64_init (qurt_signal_64_t * *signal*)

Initializes a 64-bit signal object.

The signal argument returns the initialized object. The signal object is initially cleared.

NOTE Each signal-based object has one or more kernel resources associated with it; to prevent resource leaks, call qurt_signal_destroy() when this object is not used anymore.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the initialized object.

Returns

None.

Dependencies

None.

8.12 qurt_signal_64_destroy()

8.12.1 Function Documentation

8.12.1.1 void qurt_signal_64_destroy (qurt_signal_64_t * *signal*)

Destroys the specified signal object.

NOTE 64-bit signal objects must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Signal objects must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the signal object to destroy.

Returns

None.

Dependencies

None.

8.13 qurt_signal_64_wait()

8.13.1 Function Documentation

8.13.1.1 unsigned long long qurt_signal_64_wait (qurt_signal_64_t * *signal*, unsigned long long *mask*, unsigned int *attribute*)

Suspends the current thread until all of the specified signals are set.

Signals are represented as bits 0 through 63 in the 64-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not wait on it.

If a thread is waiting on a signal object for all of the specified set of signals to be set, and all of those signals are set in the signal object, the thread is awakened.

NOTE At most, one thread can wait on a signal object at any given time.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the signal object to wait on.
in	*mask*	Mask value, which identifies the individual signals in the signal object to wait on.
in	*attribute*	Indicates whether the thread must wait until any of the signals are set, or until all of them are set. **NOTE** The wait-any and wait-all types are mutually exclusive. Values: • QURT_SIGNAL_ATTR_WAIT_ANY • QURT_SIGNAL_ATTR_WAIT_ALL

Returns

A 32-bit word with current signals.

Dependencies

None.

8.14 qurt_signal_64_set()

8.14.1 Function Documentation

8.14.1.1 void qurt_signal_64_set (qurt_signal_64_t * *signal*, unsigned long long *mask*)

Sets signals in the specified signal object.

Signals are represented as bits 0 through 63 in the 64-bit mask value. A mask bit value of 1 indicates that a signal must be set, and 0 indicates not to set it.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the signal object to modify.
in	*mask*	Mask value identifiying the individual signals to set in the signal object.

Returns

None.

Dependencies

None.

8.15 qurt_signal_64_get()

8.15.1 Function Documentation

8.15.1.1 unsigned long long qurt_signal_64_get (qurt_signal_64_t * *signal*)

Gets a signal from a signal object.

Returns the current signal values of the specified signal object.

Associated data types

qurt_signal_64_t

Parameters

in	*signal	Pointer to the signal object to access.

Returns

A 64-bit double word with current signals.

Dependencies

None.

8.16 qurt_signal_64_clear()

8.16.1 Function Documentation

8.16.1.1 void qurt_signal_64_clear (qurt_signal_64_t * *signal,* unsigned long long *mask*)

Clears signals in the specified signal object.

Signals are represented as bits 0 through 63 in the 64-bit mask value. A mask bit value of 1 indicates that a signal must be cleared, and 0 indicates not to clear it.

NOTE Signals must be explicitly cleared by a thread when it is awakened – the wait operations do not automatically clear them.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the signal object to modify.
in	*mask*	Mask value identifying the individual signals to clear in the signal object.

Returns

None.

Dependencies

None.

8.17 Data Types

Any-signals are represented in QuRT as objects of type qurt_signal_t.

8.17.1 Define Documentation

8.17.1.1 #define QURT_SIGNAL_ATTR_WAIT_ANY 0x00000000

Wait any.

8.17.1.2 #define QURT_SIGNAL_ATTR_WAIT_ALL 0x00000001

Wait all.

8.17.2 Data Structure Documentation

8.17.2.1 union qurt_signal_t

QuRT signal type.

8.17.2.2 struct qurt_signal_64_t

QuRT 64-bit signal type.

9 Any-signals

Threads use any-signals to synchronize their execution based on the occurrence of internal events.

If a signal is set in an any-signal object, and a thread is waiting on the any-signal object for that signal, the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

Threads are responsible for explicitly clearing any set signals in an any-signal object before waiting on them again. If a thread waits on a signal that has already been set, the thread continues executing.

An any-signal object contains 32 signals, which are represented as bits 0-31 in a 32-bit value. The bit value 1 indicates that a signal is set, and 0 indicates that it is cleared.

Note: At most, one thread can wait on an any-signal object at any given time.

Any-signals are stored in shared objects that support the following operations:

- qurt_anysignal_clear()
- qurt_anysignal_destroy()
- qurt_anysignal_get()
- qurt_anysignal_init()
- qurt_anysignal_set()
- qurt_anysignal_wait()
- qurt_anysignal_wait_timed()
- Data Types

9.1 qurt_anysignal_clear()

9.1.1 Function Documentation

9.1.1.1 unsigned int qurt_anysignal_clear (qurt_anysignal_t * *signal,* unsigned int *mask*)

Clears signals in the specified any-signal object.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be cleared, and 0 indicates not to clear the signal.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object, which specifies the any-signal object to modify.
in	*mask*	Signal mask value identifying the individual signals to clear in the any-signal object.

Returns

Bitmask – Old signal values (before clear).

Dependencies

None.

9.2 qurt_anysignal_destroy()

9.2.1 Function Documentation

9.2.1.1 static void qurt_anysignal_destroy (qurt_anysignal_t * *signal*)

Destroys the specified any-signal object.

NOTE Any-signal objects must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Any-signal objects must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to destroy.

Returns

None.

Dependencies

None.

9.3 qurt_anysignal_get()

9.3.1 Function Documentation

9.3.1.1 static unsigned int qurt_anysignal_get (qurt_anysignal_t * *signal*)

Gets signal values from the any-signal object.

Returns the current signal values of the specified any-signal object.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to access.

Returns

A bitmask with the current signal values of the specified any-signal object.

Dependencies

None.

9.4 qurt_anysignal_init()

9.4.1 Function Documentation

9.4.1.1 static void qurt_anysignal_init (qurt_anysignal_t * *signal*)

Initializes an any-signal object.

The any-signal object is initially cleared.

Associated data types

qurt_anysignal_t

Parameters

out	*signal*	Pointer to the initialized any-signal object.

Returns

None.

Dependencies

None.

9.5 qurt_anysignal_set()

9.5.1 Function Documentation

9.5.1.1 unsigned int qurt_anysignal_set (qurt_anysignal_t * *signal,* unsigned int *mask*)

Sets signals in the specified any-signal object.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be set, and 0 indicates not to set the sigmal.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to modify.
in	*mask*	Signal mask value identifying the individual signals to set in the any-signal object.

Returns

Bitmask of old signal values (before set).

Dependencies

None.

9.6 qurt_anysignal_wait()

9.6.1 Function Documentation

9.6.1.1 static unsigned int qurt_anysignal_wait (qurt_anysignal_t * *signal,* unsigned int *mask*)

Wait on the any-signal object.

Suspends the current thread until any one of the specified signals is set.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not to wait on the signal. If a signal is set in an any-signal object, and a thread is waiting on the any-signal object for that signal, the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

NOTE At most, one thread can wait on an any-signal object at any given time.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to wait on.
in	*mask*	Signal mask value, which specifies the individual signals in the any-signal object to wait on.

Returns

Bitmask of current signal values.

Dependencies

None.

9.7 qurt_anysignal_wait_timed()

9.7.1 Function Documentation

9.7.1.1 int qurt_anysignal_wait_timed (qurt_anysignal_t * *signal,* unsigned int *mask,* unsigned int * *signals,* unsigned long long int *duration*)

Waits on the any-signal object.

Suspends the current thread until any of the specified signals is set or timeout expires.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not to wait on the signal. If a signal is set in an any-signal object, and a thread waits on the any-signal object for that signal, the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

NOTE At most, one thread can wait on an any-signal object at any given time.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to wait on.
in	*mask*	Signal mask value, which specifies the individual signals in the any-signal object to wait on.
out	*signals*	Bitmask of current signal values.
in	*duration*	Interval (in microseconds) duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION.

Returns

QURT_EOK – Success
QURT_ETIMEDOUT – timeout

Dependencies

None.

9.8 Data Types

Any-signals are represented in QuRT as objects of type qurt_anysignal_t.

9.8.1 Typedef Documentation

9.8.1.1 typedef qurt_signal_t qurt_anysignal_t

qurt_signal_t supersedes qurt_anysignal_t. This type definition was added for backwards compatibility.

10 All-signals

Threads use all-signals to synchronize their execution based on the occurrence of one or more internal events. All-signals are stored in shared objects that support the following operations:

If one or more signals is set in an all-signal object, and a thread is waiting on the all-signal object for that particular set of signals to be set, the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

Unlike any-signals, all-signals do not need to explicitly clear any set signals in an all-signal object before waiting on them again – clearing is done automatically by the wait operation.

An all-signal object contains 32 signals, which are represented as bits 0-31 in a 32-bit value. The bit value 0 indicates that a signal is set, and 1 indicates that it is cleared (which is the opposite definition of any-signals).

Note: At most, one thread can wait on an all-signal object at any given time.

Because signal clearing is done by the wait operation, no clear operation is defined for all-signals.

All-signal services are accessed with the following QuRT functions.

- qurt_allsignal_destroy()
- qurt_allsignal_get()
- qurt_allsignal_init()
- qurt_allsignal_set()
- qurt_allsignal_wait()
- Data Types

10.1 qurt_allsignal_destroy()

10.1.1 Function Documentation

10.1.1.1 void qurt_allsignal_destroy (qurt_allsignal_t * *signal*)

Destroys the specified all-signal object.

NOTE All-signal objects must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

All-signal objects must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_allsignal_t

Parameters

in	*signal*	Pointer to the all-signal object to destroy.

Returns

None.

Dependencies

None.

10.2 qurt_allsignal_get()

10.2.1 Function Documentation

10.2.1.1 static unsigned int qurt_allsignal_get (qurt_allsignal_t * *signal*)

Gets signal values from the all-signal object.

Returns the current signal values of the specified all-signal object.

Associated data types

qurt_allsignal_t

Parameters

in	*signal*	Pointer to the all-signal object to access.

Returns

Bitmask with current signal values.

Dependencies

None.

10.3 qurt_allsignal_init()

10.3.1 Function Documentation

10.3.1.1 void qurt_allsignal_init (qurt_allsignal_t * *signal*)

Initializes an all-signal object.

The all-signal object is initially cleared.

Associated data types

qurt_allsignal_t

Parameters

out	*signal*	Pointer to the all-signal object to initialize.

Returns

None.

Dependencies

None.

10.4 qurt_allsignal_set()

10.4.1 Function Documentation

10.4.1.1 void qurt_allsignal_set (qurt_allsignal_t * *signal,* unsigned int *mask*)

Set signals in the specified all-signal object.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be set, and 0 indicates not to set the signal.

Associated data types

qurt_allsignal_t

Parameters

in	*signal*	Pointer to the all-signal object to modify.
in	*mask*	Signal mask value identifying the individual signals to set in the all-signal object.

Returns

None.

Dependencies

None.

10.5 qurt_allsignal_wait()

10.5.1 Function Documentation

10.5.1.1 void qurt_allsignal_wait (qurt_allsignal_t * *signal,* unsigned int *mask*)

Waits on the all-signal object.

Suspends the current thread until all of the specified signals are set. Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 that it is not to be waited on.

If a signal is set in an all-signal object, and a thread is waiting on the all-signal object for that signal, the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch can occur.

Unlike any-signals, all-signals do not need to explicitly clear any set signals in an all-signal object before waiting on them again – clearing is done automatically by the wait operation.

NOTE At most, one thread can wait on an all-signal object at any given time. Because signal clearing is done by the wait operation, no clear operation is defined for all-signals.

Associated data types

qurt_allsignal_t

Parameters

in	*signal*	Pointer to the all-signal object to wait on.
in	*mask*	Signal mask value, which identifies the individual signals in the all-signal object to wait on.

Returns

None.

Dependencies

None.

10.6 Data Types

All-signals are represented in QuRT as objects of type qurt_allsignal_t.

10.6.1 Data Structure Documentation

10.6.1.1 union qurt_allsignal_t

qurt_signal_t supersedes qurt_allsignal_t. This type definition was added for backwards compatibility.

11 Semaphores

Threads use semaphores to synchronize their access to shared resources. When a semaphore is initialized, it is assigned an integer count value. This value indicates the number of threads that can simultaneously access a shared resource through the semaphore. The default value is 1.

When a thread performs a down operation on a semaphore, the result depends on the semaphore count value:

- If the count value is nonzero it is decremented, and the thread gains access to the shared resource and continues executing.
- If the count value is zero it is not decremented, and the thread is suspended on the semaphore. When the count value becomes nonzero (because another thread released the semaphore) it is decremented, and the suspended thread is awakened and gains access to the shared resource.

When a thread performs an up operation on a semaphore, the semaphore count value is incremented. The result depends on the number of threads waiting on the semaphore:

- If no threads are waiting the current thread releases access to the shared resource and continues executing.
- If one or more threads are waiting and the semaphore count value is nonzero, the kernel awakens the highest-priority waiting thread and decrements the semaphore count value. If the awakened thread has higher priority than the current thread, a context switch can occur.

The add operation is similar to up, but can increment the semaphore count value by an amount greater than one. As a result, add has the potential to awaken multiple waiting threads in a single operation.

The try down operation enables a thread to try accessing a shared resource without the risk of getting suspended if its semaphore has a count value of zero:

- If the count is nonzero, try down is identical to the regular down operation.
- If the count is zero, try down returns with a value indicating the zero-count state.

Semaphores are shared objects that support the following operations:

- qurt_sem_add()
- qurt_sem_destroy()
- qurt_sem_down()
- qurt_sem_get_val()
- qurt_sem_init()
- qurt_sem_init_val()
- qurt_sem_try_down()

- qurt_sem_up()
- qurt_sem_down_timed()
- Data Types

11.1 qurt_sem_add()

11.1.1 Function Documentation

11.1.1.1 int qurt_sem_add (qurt_sem_t * *sem*, unsigned int *amt*)

Releases access to a shared resource (the specified amount increments the semaphore count value).

When a thread performs an add operation on a semaphore, the specified value increments the semaphore count. The result depends on the number of threads waiting on the semaphore:

- When no threads are waiting, the current thread releases access to the shared resource and continues executing.
- When one or more threads are waiting and the semaphore count value is nonzero, the kernel repeatedly awakens the highest-priority waiting thread and decrements the semaphore count value until either no waiting threads remain or the semaphore count value is zero. If any of the awakened threads has higher priority than the current thread, a context switch can occur.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.
in	*amt*	Amount to increment the semaphore count value.

Returns

Unused integer value.

Dependencies

None.

11.2 qurt_sem_destroy()

11.2.1 Function Documentation

11.2.1.1 void qurt_sem_destroy (qurt_sem_t * *sem*)

Destroys the specified semaphore.

NOTE Semaphores must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Semaphores must not be destroyed while they are still in use. If this occur, the behavior of QuRT is undefined.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to destroy.

Returns

None.

Dependencies

None.

11.3 qurt_sem_down()

11.3.1 Function Documentation

11.3.1.1 int qurt_sem_down (qurt_sem_t * *sem*)

Requests access to a shared resource. When a thread performs a down operation on a semaphore, the result depends on the semaphore count value:

- When the count value is nonzero, it is decremented, and the thread gains access to the shared resource and continues executing.
- When the count value is zero, it is not decremented, and the thread is suspended on the semaphore. When the count value becomes nonzero (because another thread released the semaphore) it is decremented, and the suspended thread is awakened and gains access to the shared resource.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.

Returns

Unused integer value.

Dependencies

None.

11.4 qurt_sem_get_val()

11.4.1 Function Documentation

11.4.1.1 static unsigned short qurt_sem_get_val (qurt_sem_t * *sem*)

Gets the semaphore count value.

Returns the current count value of the specified semaphore.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.

Returns

Integer semaphore count value

Dependencies

None.

11.5 qurt_sem_init()

11.5.1 Function Documentation

11.5.1.1 void qurt_sem_init (qurt_sem_t * *sem*)

Initializes a semaphore object. The default initial value of the semaphore count value is 1.

Parameters

out	*sem*	Pointer to the initialized semaphore object.

Returns

None.

Dependencies

None.

11.6 qurt_sem_init_val()

11.6.1 Function Documentation

11.6.1.1 void qurt_sem_init_val (qurt_sem_t * *sem*, unsigned short *val*)

Initializes a semaphore object with the specified value.

Associated data types

qurt_sem_t

Parameters

out	*sem*	Pointer to the initialized semaphore object.
in	*val*	Initial value of the semaphore count value.

Returns

None.

Dependencies

None.

11.7 qurt_sem_try_down()

11.7.1 Function Documentation

11.7.1.1 int qurt_sem_try_down (qurt_sem_t * *sem*)

Requests access to a shared resource (without suspend). When a thread performs a try down operation on a semaphore, the result depends on the semaphore count value:

- The count value is decremented when it is nonzero. The down operation returns 0 as the function result, and the thread gains access to the shared resource and is free to continue executing.
- The count value is not decremented when it is zero. The down operation returns -1 as the function result, and the thread does not gain access to the shared resource and should not continue executing.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.

Returns

0 – Success.
-1 – Failure.

Dependencies

None.

11.8 qurt_sem_up()

11.8.1 Function Documentation

11.8.1.1 static int qurt_sem_up (qurt_sem_t * *sem*)

Releases access to a shared resource. When a thread performs an up operation on a semaphore, the semaphore count value increments. The result depends on the number of threads waiting on the semaphore:

- When no threads are waiting, the current thread releases access to the shared resource and continues executing.
- When one or more threads are waiting and the semaphore count value is nonzero, the kernel awakens the highest-priority waiting thread and decrements the semaphore count value. If the awakened thread has higher priority than the current thread, a context switch can occur.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.

Returns

Unused integer value.

Dependencies

None.

11.9 qurt_sem_down_timed()

11.9.1 Function Documentation

11.9.1.1 int qurt_sem_down_timed (qurt_sem_t * *sem*, unsigned long long int *duration*)

When a thread performs a down operation on a semaphore, the result depends on the semaphore count value:

- When the count value is nonzero, it is decremented, and the thread gains access to the shared resource and continues executing.
- When the count value is zero, it is not decremented, and the thread is suspended on the semaphore. When the count value becomes nonzero (because another thread released the semaphore) it is decremented, and the suspended thread is awakened and gains access to the shared resource. Terminate the wait when the specified timeout expires. If timeout expires, terminate this wait and grant no access to the shared resource.

Associated data types

qurt_sem_t

Parameters

in	*sem*	Pointer to the semaphore object to access.
in	*duration*	Interval (in microseconds) duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION

Returns

QURT_EOK – Success
QURT_ETIMEDOUT – Timeout

Dependencies

None.

11.10 Data Types

Semaphores are represented in QuRT as objects of type qurt_sem_t.

11.10.1 Data Structure Documentation

11.10.1.1 union qurt_sem_t

QuRT semaphore type.

12 Barriers

Threads use barriers to synchronize their execution at a specific point in a program.

When a barrier is initialized it is assigned a user-specified integer value. This value indicates the number of threads to synchronize on the barrier.

When a thread waits on a barrier, it is suspended on the barrier:

- If the total number of threads waiting on the barrier is less than the barrier's assigned value, no other action occurs.
- If the total number of threads waiting on the barrier equals the barrier's assigned value, all threads currently waiting on the barrier are awakened, allowing them to execute past the barrier.

After its waiting threads are awakened, a barrier is automatically reset and can be used again in the program without the need for re-initialization.

Barriers are shared objects that support the following operations:

- qurt_barrier_destroy()
- qurt_barrier_init()
- qurt_barrier_wait()
- Data Types

12.1 qurt_barrier_destroy()

12.1.1 Function Documentation

12.1.1.1 int qurt_barrier_destroy (qurt_barrier_t * *barrier*)

Destroys the specified barrier.

NOTE Barriers must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Barriers must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_barrier_t

Parameters

in	*barrier*	Pointer to the barrier object to destroy.

Returns

Unused integer value.

Dependencies

None.

12.2 qurt_barrier_init()

12.2.1 Function Documentation

12.2.1.1 int qurt_barrier_init (qurt_barrier_t * *barrier,* unsigned int *threads_total*)

Initializes a barrier object.

Associated data types

qurt_barrier_t

Parameters

out	*barrier*	Pointer to the barrier object to initialize.
in	*threads_total*	Total number of threads to synchronize on the barrier.

Returns

Unused integer value.

Dependencies

None.

12.3 qurt_barrier_wait()

12.3.1 Function Documentation

12.3.1.1 int qurt_barrier_wait (qurt_barrier_t * *barrier*)

Waits on the barrier.

Suspends the current thread on the specified barrier.

The function return value indicates whether the thread was the last one to synchronize on the barrier. When a thread waits on a barrier, it is suspended on the barrier:

- If the total number of threads waiting on the barrier is less than the assigned value of the barrier, no other action occurs.
- If the total number of threads waiting on the barrier equals the assigned value of the barrier, all threads currently waiting on the barrier are awakened, allowing them to execute past the barrier.

NOTE After its waiting threads are awakened, a barrier is automatically reset and can be used again in the program without the need for re-initialization.

Associated data types

qurt_barrier_t

Parameters

in	*barrier*	Pointer to the barrier object to wait on.

Returns

QURT_BARRIER_OTHER – Current thread awakened from barrier.
QURT_BARRIER_SERIAL_THREAD – Current thread is last caller of barrier.

Dependencies

None.

12.4 Data Types

Barriers are represented in QuRT as objects of type qurt_barrier_t.

12.4.1 Define Documentation

12.4.1.1 #define QURT_BARRIER_SERIAL_THREAD 1

Serial thread.

12.4.1.2 #define QURT_BARRIER_OTHER 0

Other.

12.4.2 Data Structure Documentation

12.4.2.1 union qurt_barrier_t

QuRT barrier type.

13 Condition Variables

Threads use condition variables to synchronize their execution based on the value in a shared data item. Condition variables are useful in cases where a thread would normally have to continuously poll a data item until it contained a specific value – using a condition variable the thread can accomplish the same task without the need for polling.

A condition variable is always used with an associated mutex (Section 5) to ensure that the shared data item is checked and updated without thread contention.

For a thread to wait for a specific condition on a shared data item, it must first lock the mutex that controls access to the data item. If the condition is not satisfied, the thread then performs the wait condition operation on the condition variable (which suspends the thread and unlocks the mutex).

For a thread to signal that a condition is true on a shared data item, it must first lock the mutex that controls access to the data item, then perform the signal condition operation, and finally explicitly unlock the mutex.

The signal condition operation is used to awaken a single waiting thread. If multiple threads are waiting on a condition variable, they can all be awakened by using the broadcast condition operation.

Note: Failure to properly lock and unlock mutexes with condition variables can cause the threads to never be suspended (or suspended but never awakened).

Because QuRT allows threads to be awakened by spurious conditions, threads should always verify the target condition on being awakened.

Condition variables are shared objects that support the following operations:

- qurt_cond_broadcast()
- qurt_cond_destroy()
- qurt_cond_init()
- qurt_cond_signal()
- qurt_cond_wait()
- qurt_cond_wait2()
- Data Types

13.1 qurt_cond_broadcast()

13.1.1 Function Documentation

13.1.1.1 void qurt_cond_broadcast (qurt_cond_t * *cond*)

Signals multiple waiting threads that the specified condition is true.

When a thread wishes to broadcast that a condition is true on a shared data item, it must perform the following procedure:

1. Lock the mutex that controls access to the data item.
2. Perform the broadcast condition operation.
3. Unlock the mutex.

NOTE Failure to properly lock and unlock the mutex of a condition variable can cause the threads to never be suspended (or suspended but never awakened).

Use condition variables only with regular mutexes – attempting to use recursive mutexes or priority inheritance mutexes results in undefined behavior.

Associated data types

qurt_cond_t

Parameters

in	*cond*	Pointer to the condition variable object to signal.

Returns

None.

Dependencies

None.

13.2 qurt_cond_destroy()

13.2.1 Function Documentation

13.2.1.1 void qurt_cond_destroy (qurt_cond_t * *cond*)

Destroys the specified condition variable.

NOTE Conditions must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Conditions must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_cond_t

Parameters

in	*cond*	Pointer to the condition variable object to destroy.

Returns

None.

13.3 qurt_cond_init()

13.3.1 Function Documentation

13.3.1.1 void qurt_cond_init (qurt_cond_t * *cond*)

Initializes a conditional variable object.

Associated data types

qurt_cond_t

Parameters

out	*cond*	Pointer to the initialized condition variable object.

Returns

None.

Dependencies

None.

13.4 qurt_cond_signal()

13.4.1 Function Documentation

13.4.1.1 void qurt_cond_signal (qurt_cond_t * *cond*)

Signals a waiting thread that the specified condition is true.

When a thread wishes to signal that a condition is true on a shared data item, it must perform the following procedure:

1. Lock the mutex that controls access to the data item.
2. Perform the signal condition operation.
3. Unlock the mutex.

NOTE Failure to properly lock and unlock a mutex of a condition variable can cause the threads to never be suspended (or suspended but never awakened).

Use condition variables only with regular mutexes – attempting to use recursive mutexes or priority inheritance mutexes results in undefined behavior.

Associated data types

qurt_cond_t

Parameters

in	*cond*	Pointer to the condition variable object to signal.

Returns

None.

Dependencies

None.

13.5 qurt_cond_wait()

13.5.1 Function Documentation

13.5.1.1 void qurt_cond_wait (qurt_cond_t * *cond*, qurt_mutex_t * *mutex*)

Suspends the current thread until the specified condition is true. When a thread wishes to wait for a specific condition on a shared data item, it must perform the following procedure:

1. Lock the mutex that controls access to the data item.
2. If the condition is not satisfied, perform the wait condition operation on the condition variable (suspends the thread and unlocks the mutex).

NOTE Failure to properly lock and unlock the mutex of a condition variable can cause the threads to never be suspended (or suspended but never awakened).

Use condition variables only with regular mutexes – attempting to use recursive mutexes or priority inheritance mutexes results in undefined behavior.

Associated data types

qurt_cond_t
qurt_mutex_t

Parameters

in	*cond*	Pointer to the condition variable object to wait on.
in	*mutex*	Pointer to the mutex associated with condition variable to wait on.

Returns

None.

Dependencies

None.

13.6 qurt_cond_wait2()

13.6.1 Function Documentation

13.6.1.1 void qurt_cond_wait2 (qurt_cond_t * *cond*, qurt_rmutex2_t * *mutex*)

Suspends the current thread until the specified condition is true. When a thread wishes to wait for a specific condition on a shared data item, it must perform the following procedure:

1. Lock the mutex that controls access to the data item.
2. If the condition is not satisfied, perform the wait condition operation on the condition variable, which suspends the thread and unlocks the mutex.

NOTE Failure to properly lock and unlock the mutex of a condition variable can cause the threads to never be suspended (or suspended but never awakened).

Use condition variables only with regular mutexes – attempting to use recursive mutexes or priority inheritance mutexes results in undefined behavior.

This is the same API as qurt_cond_wait(), use this version when using mutexes of type qurt_rmutex2_t.

Associated data types

qurt_cond_t
qurt_rmutex2_t

Parameters

in	*cond*	Pointer to the condition variable object to wait on.
in	*mutex*	Pointer to the mutex associated with the condition variable to wait on.

Returns

None.

Dependencies

None.

13.7 Data Types

Condition variables are represented in QuRT as objects of type qurt_cond_t.

13.7.1 Data Structure Documentation

13.7.1.1 union qurt_cond_t

QuRT condition variable type.

13.7.1.2 struct qurt_rmutex2_t

QuRT rmutex2 type. Mutex type used with rmutex2 APIs.

14 Pipes

Threads use pipes to perform synchronized exchange of data streams.

When a pipe object is initialized, it uses a user-allocated FIFO buffer to store one or more elements of pipe data. The pipe buffer address and length are specified as parameters.

When creating a pipe object, the pipe buffer is allocated as part of the create operation. In this case, only the pipe buffer length is specified as a parameter.

If a thread reads from an empty pipe, it is suspended on the pipe. When another thread writes to the pipe, the suspended thread is awakened and can then read data from the pipe.

If a thread writes to a full pipe, it is suspended on the pipe. When another thread reads from the pipe, the suspended thread is awakened and can then write data to the pipe.

The try operations enable a thread to try reading or writing from a pipe without the risk of getting suspended if the pipe is empty (on a read) or full (on a write). If the operation cannot be performed, it returns with a value indicating the state of the pipe.

The cancellable operations automatically return if a system-level event interrupts the calling thread: in particular, if the user process of a thread is killed, or if the thread must finish its current QDI invocation and return to the user space.

Pipe data items are defined as 64-bit values. Pipe reads and writes are limited to transferring a single 64-bit data item per operation. Data items larger than 64 bits can be transferred by reading and writing pointers to the data (rather than the data itself), or by transferring the data in consecutive 64-bit chunks.

Note: Multiple threads can read from or write to a single pipe.

Pipes have the following attributes:

- Buffer – The pipe buffer address specifies the byte address of the start of the pipe data buffer.
- Elements – The pipe buffer length specifies the length of the pipe data buffer; expressed in terms of the number of 64-bit data elements that can be stored in the buffer.
- Buffer partition – Pipe buffer allocated in either RAM or TCM/LPM.

The qurt_pipe_attr_init() and qurt_pipe_attr_set functions set the pipe attributes before a pipe is created.

Note: The pipe attribute structure stores the pipe buffer address and buffer length. The pipe create operation ignores the buffer address attribute– for create operations only the buffer length must be set.

Pipes are shared objects that support the following operations:

- qurt_pipe_attr_init()
- qurt_pipe_attr_set_buffer()
- qurt_pipe_attr_set_buffer_partition()

- qurt_pipe_attr_set_elements()
- qurt_pipe_create()
- qurt_pipe_delete()
- qurt_pipe_destroy()
- qurt_pipe_init()
- qurt_pipe_is_empty()
- qurt_pipe_receive()
- qurt_pipe_receive_cancellable()
- qurt_pipe_send()
- qurt_pipe_send_cancellable()
- qurt_pipe_try_receive()
- qurt_pipe_try_send()
- Data Types

14.1 qurt_pipe_attr_init()

14.1.1 Function Documentation

14.1.1.1 static void qurt_pipe_attr_init (qurt_pipe_attr_t * *attr*)

Initializes the structure that sets the pipe attributes when a pipe is created.

After an attribute structure is initialized, the individual attributes in the structure are explicitly set using the pipe attribute operations.

The attribute structure is assigned the following default values:

- buffer – 0
- elements – 0
- mem_partition – QURT_PIPE_ATTR_MEM_PARTITION_RAM

Associated data types

qurt_pipe_attr_t

Parameters

in,out	*attr*	Pointer to the pipe attribute structure.

Returns

None.

Dependencies

None.

14.2 qurt_pipe_attr_set_buffer()

14.2.1 Function Documentation

14.2.1.1 static void qurt_pipe_attr_set_buffer (qurt_pipe_attr_t * *attr,* qurt_pipe_data_t * *buffer*)

Sets the pipe buffer address attribute.

Specifies the base address of the memory area to use for the data buffer of a pipe.

The base address and size (Section 14.4.1.1) specify the memory area used as a pipe data buffer. The user is responsible for allocating the memory area used for the buffer.

Associated data types

qurt_pipe_attr_t
qurt_pipe_data_t

Parameters

in,out	*attr*	Pointer to the pipe attribute structure.
in	*buffer*	Pointer to the buffer base address.

Returns

None.

Dependencies

None.

14.3 qurt_pipe_attr_set_buffer_partition()

14.3.1 Function Documentation

14.3.1.1 static void qurt_pipe_attr_set_buffer_partition (qurt_pipe_attr_t * *attr,* unsigned char *mem_partition*)

Specifies the memory type where a pipe's buffer is allocated. Allocate pipes in RAM or TCM/LPM.

NOTE If a pipe is specified as allocated in TCM/LPM, it must be created with the qurt_pipe_init() operation. The qurt_pipe_create() operation results in an error.

Associated data types

qurt_pipe_attr_t

Parameters

in,out	*attr*	Pointer to the pipe attribute structure.
in	*mem_partition*	Pipe memory partition. Values: • QURT_PIPE_ATTR_MEM_PARTITION_RAM – Pipe resides in RAM • QURT_PIPE_ATTR_MEM_PARTITION_TCM – Pipe resides in TCM/LCM

Returns

None.

Dependencies

None.

14.4 qurt_pipe_attr_set_elements()

14.4.1 Function Documentation

14.4.1.1 static void qurt_pipe_attr_set_elements (qurt_pipe_attr_t * *attr,* unsigned int *elements*)

Specifies the length of the memory area to use for the data buffer of a pipe.

The length is expressed in terms of the number of 64-bit data elements that can be stored in the buffer.

The base address (Section 14.2.1.1) and size specify the memory area used as a pipe data buffer. The user is responsible for allocating the memory area used for the buffer.

Associated data types

qurt_pipe_attr_t

Parameters

in,out	*attr*	Pointer to the pipe attribute structure.
in	*elements*	Pipe length (64-bit elements).

Returns

None.

Dependencies

None.

14.5 qurt_pipe_create()

14.5.1 Function Documentation

14.5.1.1 int qurt_pipe_create (qurt_pipe_t ** *pipe*, qurt_pipe_attr_t * *attr*)

Creates a pipe.

Allocates a pipe object and its associated data buffer, and initializes the pipe object.

NOTE The buffer address and size stored in the attribute structure specify how the pipe data buffer is allocated.

If a pipe is specified as allocated in TCM/LPM, it must be created using the qurt_pipe_init() operation. The qurt_pipe_create() operation results in an error.

Associated data types

qurt_pipe_t
qurt_pipe_attr_t

Parameters

out	*pipe*	Pointer to the created pipe object.
in	*attr*	Pointer to the attribute structure used to create the pipe.

Returns

QURT_EOK – Pipe created.
QURT_EFAILED – Pipe not created.
QURT_ENOTALLOWED – Pipe cannot be created in TCM/LPM.

Dependencies

None.

14.6 qurt_pipe_delete()

14.6.1 Function Documentation

14.6.1.1 void qurt_pipe_delete (qurt_pipe_t * *pipe*)

Deletes the pipe.

Destroys the specified pipe (Section 14.7.1.1) and deallocates the pipe object and its associated data buffer.

NOTE Delete pipes only if they were created using qurt_pipe_create (and not qurt_pipe_init). Otherwise the behavior of QuRT is undefined.

Pipes must be deleted when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Pipes must not be deleted while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_pipe_t

Parameters

in	*pipe*	Pointer to the pipe object to destroy.

Returns

None.

Dependencies

None.

14.7 qurt_pipe_destroy()

14.7.1 Function Documentation

14.7.1.1 void qurt_pipe_destroy (qurt_pipe_t * *pipe*)

Destroys the specified pipe.

NOTE Pipes must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel. Pipes must not be destroyed while they are still in use. If this occurs, the behavior of QuRT is undefined.

Associated data types

qurt_pipe_t

Parameters

in	*pipe*	Pointer to the pipe object to destroy.

Returns

None.

Dependencies

None.

14.8 qurt_pipe_init()

14.8.1 Function Documentation

14.8.1.1 int qurt_pipe_init (qurt_pipe_t * *pipe*, qurt_pipe_attr_t * *attr*)

Initializes a pipe object using an existing data buffer.

NOTE The buffer address and size stored in the attribute structure must specify a data buffer that the user has already allocated.

Associated data types

qurt_pipe_t
qurt_pipe_attr_t

Parameters

out	*pipe*	Pointer to the pipe object to initialize.
in	*attr*	Pointer to the pipe attribute structure used to initialize the pipe.

Returns

QURT_EOK – Success.
QURT_EFAILED – Failure.

Dependencies

None.

14.9 qurt_pipe_is_empty()

14.9.1 Function Documentation

14.9.1.1 int qurt_pipe_is_empty (qurt_pipe_t * *pipe*)

Returns a value indicating whether the specified pipe contains any data.

Associated data types

qurt_pipe_t

Parameters

in	*pipe*	Pointer to the pipe object to read from.

Returns

1 – Pipe contains no data.
0 – Pipe contains data.

Dependencies

None.

14.10 qurt_pipe_receive()

14.10.1 Function Documentation

14.10.1.1 qurt_pipe_data_t qurt_pipe_receive (qurt_pipe_t * *pipe*)

Reads a data item from the specified pipe.

If a thread reads from an empty pipe, it is suspended on the pipe. When another thread writes to the pipe, the suspended thread is awakened and can then read data from the pipe. Pipe data items are defined as 64-bit values. Pipe reads are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t

Parameters

in	*pipe*	Pointer to the pipe object to read from.

Returns

Integer containing the 64-bit data item from pipe.

Dependencies

None.

14.11 qurt_pipe_receive_cancellable()

14.11.1 Function Documentation

14.11.1.1 int qurt_pipe_receive_cancellable (qurt_pipe_t * *pipe*, qurt_pipe_data_t * *result*)

Reads a data item from the specified pipe (with suspend), cancellable.

If a thread reads from an empty pipe, it is suspended on the pipe. When another thread writes to the pipe, the suspended thread is awakened and can then read data from the pipe. The operation is cancelled if the user process of the calling thread is killed, or if the calling thread must finish its current QDI invocation and return to user space.

Pipe data items are defined as 64-bit values. Pipe reads are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t
qurt_pipe_data_t

Parameters

in	*pipe*	Pointer to the pipe object to read from.
in	*result*	Pointer to the integer containing the 64-bit data item from pipe.

Returns

QURT_EOK – Receive completed.
QURT_ECANCEL – Receive cancelled.

Dependencies

None.

14.12 qurt_pipe_send()

14.12.1 Function Documentation

14.12.1.1 void qurt_pipe_send (qurt_pipe_t * *pipe*, qurt_pipe_data_t *data*)

Writes a data item to the specified pipe.

If a thread writes to a full pipe, it is suspended on the pipe. When another thread reads from the pipe, the suspended thread is awakened and can then write data to the pipe.

Pipe data items are defined as 64-bit values. Pipe writes are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t
qurt_pipe_data_t

Parameters

in	*pipe*	Pointer to the pipe object to write to.
in	*data*	Data item to write.

Returns

None.

Dependencies

None.

14.13 qurt_pipe_send_cancellable()

14.13.1 Function Documentation

14.13.1.1 int qurt_pipe_send_cancellable (qurt_pipe_t * *pipe*, qurt_pipe_data_t *data*)

Writes a data item to the specified pipe (with suspend), cancellable.

If a thread writes to a full pipe, it is suspended on the pipe. When another thread reads from the pipe, the suspended thread is awakened and can then write data to the pipe. The operation is canceled if the user process of the calling thread is killed, or if the calling thread must finish its current QDI invocation and return to the user space.

Pipe data items are defined as 64-bit values. Pipe writes are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t
qurt_pipe_data_t

Parameters

in	*pipe*	Pointer to the pipe object to read from.
in	*data*	Data item to write.

Returns

QURT_EOK – Send complete.
QURT_ECANCEL – Send canceled.

Dependencies

None.

14.14 qurt_pipe_try_receive()

14.14.1 Function Documentation

14.14.1.1 qurt_pipe_data_t qurt_pipe_try_receive (qurt_pipe_t * *pipe*, int * *success*)

Reads a data item from the specified pipe (without suspending the thread if the pipe is empty).

If a thread reads from an empty pipe, the operation returns immediately with success set to -1. Otherwise, success is always set to 0 to indicate a successful read operation.

Pipe data items are defined as 64-bit values. Pipe reads are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t

Parameters

in	*pipe*	Pointer to the pipe object to read from.
out	*success*	Pointer to the operation status result.

Returns

Integer containing a 64-bit data item from pipe.

Dependencies

None.

14.15 qurt_pipe_try_send()

14.15.1 Function Documentation

14.15.1.1 int qurt_pipe_try_send (qurt_pipe_t * *pipe*, qurt_pipe_data_t *data*)

Writes a data item to the specified pipe (without suspending the thread if the pipe is full).

If a thread writes to a full pipe, the operation returns immediately with success set to -1. Otherwise, success is always set to 0 to indicate a successful write operation.

Pipe data items are defined as 64-bit values. Pipe writes are limited to transferring a single 64-bit data item per operation.

NOTE Transfer data items larger than 64 bits by reading and writing pointers to the data, or by transferring the data in consecutive 64-bit chunks.

Associated data types

qurt_pipe_t
qurt_pipe_data_t

Parameters

in	*pipe*	Pointer to the pipe object to write to.
in	*data*	Data item to write.

Returns

0 – Success.
-1 – Failure (pipe full).

Dependencies

None.

14.16 Data Types

Pipes are represented in QuRT as objects of type qurt_pipe_t.

Pipe data values are represented as objects of type qurt_pipe_data_t.

Pipe attributes in QuRT are stored in structures of type qurt_pipe_attr_t.

14.16.1 Define Documentation

14.16.1.1 #define QURT_PIPE_MAGIC 0xF1FEF1FE

Magic.

14.16.1.2 #define QURT_PIPE_ATTR_MEM_PARTITION_RAM 0

RAM.

14.16.1.3 #define QURT_PIPE_ATTR_MEM_PARTITION_TCM 1

TCM.

14.16.2 Data Structure Documentation

14.16.2.1 struct qurt_pipe_t

QuRT pipe type.

14.16.2.2 struct qurt_pipe_attr_t

QuRT pipe attributes type.

14.16.3 Typedef Documentation

14.16.3.1 typedef unsigned long long int qurt_pipe_data_t

QuRT pipe data values type.

15 Timers

Threads use timers to perform actions that must occur at specific intervals. A timer waits for the specified period of time and then generates a timer event.

When a timer object is created, it is both started and associated with the specified signal object and signal mask. Whenever the timer expires, the signal specified in the signal mask is set in the signal object. A timer event handler must be implemented by the user program to wait on that signal to handle the timer event.

Stop a running timer by calling the timer stop operation. Restart a stopped (or expired) timer with a specified duration by calling the timer restart operation.

A thread can suspend itself (Section 3) for a specific amount of time by calling the timer sleep operation. The sleep duration specifies the interval (in microseconds) between when the thread is suspended and when it is re-awakened.

Timers can be assigned to groups that make it possible to enable or disable one or more timers with a single operation. A timer state is saved across disabling and subsequent re-enabling.

Access the static attributes of a running timer with the get timer attributes operation.

Note: Timers can run for up to 36 hours, and have a worst-case error margin of 60 microseconds.

Timers have the following attributes:

- Duration is the interval between timer events; specifies the interval (in microseconds) between the creation of the timer object and the generation of the corresponding timer event.
- Type is the timer functional behavior (one-shot or periodic):
  - A one-shot timer (QURT_TIMER_ONESHOT) waits for the specified timer duration and then generates a single timer event. After this the timer is nonfunctional.
  - A periodic timer (QURT_TIMER_PERIODIC) repeatedly waits for the specified timer duration and then generates a timer event. The result is a series of timer events with interval equal to the timer duration.
- Group is the timer group that timer is assigned to; timer groups are used to enable or disable one or more timers with a single operation.
- Remaining returns the time remaining (in microseconds) before the generation of the next timer event on the timer (read-only).
- Expiry is the absolute time (in microseconds) when the timer expires. Absolute time is defined as the time elapsed since the previous hardware reset of the Hexagon processor. This attribute applies only to one-shot timers.

The qurt_timer_attr_init() and qurt_timer_attr_set() functions set the timer attributes before a timer is created.

The timer type must be set on all timers. Depending on the type, either the timer duration or expiry is set – expiry applies only to one-shot timers. The timer group is optional.

The qurt_timer_get_attr() and qurt_timer_attr_get functions retrieve timer attributes from a created timer.

Of the attributes retrieved from a timer, the timer remaining is the only dynamic attribute – it returns the time remaining before the next event occurs on the timer. The other returned attributes are static, and remain unchanged from when they were set.
Timer objects are assigned to specific threads. They support the following operations:

- qurt_timer_attr_get_duration()
- qurt_timer_attr_get_group()
- qurt_timer_attr_get_remaining()
- qurt_timer_attr_get_type()
- qurt_timer_attr_init()
- qurt_timer_attr_set_duration()
- qurt_timer_attr_set_expiry()
- qurt_timer_attr_set_group()
- qurt_timer_attr_set_type()
- qurt_timer_create()
- qurt_timer_delete()
- qurt_timer_get_attr()
- qurt_timer_group_disable()
- qurt_timer_group_enable()
- qurt_timer_restart()
- qurt_timer_sleep()
- qurt_timer_stop()
- Timer Data Types

15.1 qurt_timer_attr_get_duration()

15.1.1 Function Documentation

15.1.1.1 void qurt_timer_attr_get_duration (qurt_timer_attr_t * *attr,* qurt_timer_duration_t * *duration*)

Gets the timer duration from the specified timer attribute structure. The value returned is the duration that was originally set for the timer.

NOTE This function does not return the remaining time of an active timer; use qurt_timer_attr_get_remaining() to get the remaining time.

Associated data types

qurt_timer_attr_t
qurt_timer_duration_t

Parameters

in	*attr*	Pointer to the timer attributes object
out	*duration*	Pointer to the destination variable for timer duration.

Returns

None.

Dependencies

None.

15.2 qurt_timer_attr_get_group()

15.2.1 Function Documentation

15.2.1.1 void qurt_timer_attr_get_group (qurt_timer_attr_t * *attr,* unsigned int * *group*)

Gets the timer group identifier from the specified timer attribute structure.

Associated data types

qurt_timer_attr_t

Parameters

in	*attr*	Pointer to the timer attribute structure.
out	*group*	Pointer to the destination variable for the timer group identifier.

Returns

None.

Dependencies

None.

15.3 qurt_timer_attr_get_remaining()

15.3.1 Function Documentation

15.3.1.1 void qurt_timer_attr_get_remaining (qurt_timer_attr_t * *attr,* qurt_timer_-duration_t * *remaining*)

Gets the timer remaining duration from the specified timer attribute structure.

The timer remaining duration indicates (in microseconds) how much time remains before the generation of the next timer event on the corresponding timer. In most cases this function assumes that the timer attribute structure was obtained by calling qurt_timer_get_attr().

NOTE This attribute is read-only and thus has no set operation defined for it.

Associated data types

qurt_timer_attr_t
qurt_timer_duration_t

Parameters

in	*attr*	Pointer to the timer attribute object.
out	*remaining*	Pointer to the destination variable for remaining time.

Returns

None.

Dependencies

None.

15.4 qurt_timer_attr_get_type()

15.4.1 Function Documentation

15.4.1.1 void qurt_timer_attr_get_type (qurt_timer_attr_t * *attr,* qurt_timer_type_t * *type*)

Gets the timer type from the specified timer attribute structure.

Associated data types

qurt_timer_attr_t
qurt_timer_type_t

Parameters

in	*attr*	Pointer to the timer attribute structure.
out	*type*	Pointer to the destination variable for the timer type.

Returns

None.

Dependencies

None.

15.5 qurt_timer_attr_init()

15.5.1 Function Documentation

15.5.1.1 void qurt_timer_attr_init (qurt_timer_attr_t * *attr*)

Initializes the specified timer attribute structure with default attribute values:

- Timer duration – QURT_TIMER_DEFAULT_DURATION (Section 15)
- Timer type – QURT_TIMER_ONESHOT
- Timer group – QURT_TIMER_DEFAULT_GROUP

Associated data types

qurt_timer_attr_t

Parameters

in,out	*attr*	Pointer to the destination structure for the timer attributes.

Returns

None.

Dependencies

None.

15.6 qurt_timer_attr_set_duration()

15.6.1 Function Documentation

15.6.1.1 void qurt_timer_attr_set_duration (qurt_timer_attr_t * *attr,* qurt_timer_duration_t *duration*)

Sets the timer duration in the specified timer attribute structure.

The timer duration specifies the interval (in microseconds) between the creation of the timer object and the generation of the corresponding timer event.

The timer duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION (Section 15). Otherwise, the set operation is ignored.

Associated data types

qurt_timer_attr_t
qurt_timer_duration_t

Parameters

in,out	*attr*	Pointer to the timer attribute structure.
in	*duration*	Timer duration (in microseconds). Valid range is QURT_TIMER_MIN_DURATION to QURT_TIMER_MAX_DURATION.

Returns

None.

Dependencies

None.

15.7 qurt_timer_attr_set_expiry()

15.7.1 Function Documentation

15.7.1.1 void qurt_timer_attr_set_expiry (qurt_timer_attr_t * *attr,* qurt_timer_time_t *time*)

Sets the absolute expiry time in the specified timer attribute structure.

The timer expiry specifies the absolute time (in microseconds) of the generation of the corresponding timer event.

Timer expiries are relative to when the system first began executing.

Associated data types

qurt_timer_attr_t
qurt_timer_time_t

Parameters

in,out	*attr*	Pointer to the timer attribute structure.
in	*time*	Timer expiry.

Returns

None.

Dependencies

None.

15.8 qurt_timer_attr_set_group()

15.8.1 Function Documentation

15.8.1.1 void qurt_timer_attr_set_group (qurt_timer_attr_t * *attr,* unsigned int *group*)

Sets the timer group identifier in the specified timer attribute structure.

The timer group identifier specifies the group that the timer belongs to. Timer groups are used to enable or disable one or more timers in a single operation.

The timer group identifier value must be between 0 and (QURT_TIMER_MAX_GROUPS - 1). See Section 15.

Associated data types

qurt_timer_attr_t

Parameters

in,out	*attr*	Pointer to the timer attribute object.
in	*group*	Timer group identifier; Valid range is 0 to (QURT_TIMER_MAX_GROUPS - 1).

Returns

None.

Dependencies

None.

15.9 qurt_timer_attr_set_type()

15.9.1 Function Documentation

15.9.1.1 void qurt_timer_attr_set_type (qurt_timer_attr_t * *attr*, qurt_timer_type_t *type*)

Sets the timer type in the specified timer attribute structure.

The timer type specifies the functional behavior of the timer:

- A one-shot timer (QURT_TIMER_ONESHOT) waits for the specified timer duration and then generates a single timer event. After this the timer is nonfunctional.
- A periodic timer (QURT_TIMER_PERIODIC) repeatedly waits for the specified timer duration and then generates a timer event. The result is a series of timer events with interval equal to the timer duration.

Associated data types

qurt_timer_attr_t
qurt_timer_type_t

Parameters

in,out	*attr*	Pointer to the timer attribute structure.
in	*type*	Timer type. Values are: • QURT_TIMER_ONESHOT – One-shot timer. • QURT_TIMER_PERIODIC – Periodic timer.

Returns

None.

Dependencies

None.

15.10 qurt_timer_create()

15.10.1 Function Documentation

15.10.1.1 int qurt_timer_create (qurt_timer_t * *timer,* const qurt_timer_attr_t * *attr,* const qurt_anysignal_t * *signal,* unsigned int *mask*)

Creates a timer.

Allocates and initializes a timer object, and starts the timer.

NOTE A timer event handler must be defined to wait on the specified signal to handle the timer event.

Associated data types

qurt_timer_t
qurt_timer_attr_t
qurt_anysignal_t

Parameters

out	*timer*	Pointer to the created timer object.
in	*attr*	Pointer to the timer attribute structure.
in	*signal*	Pointer to the signal object set when timer expires.
in	*mask*	Signal mask, which specifies the signal to set in the signal object when the time expires.

Returns

QURT_EOK – Success.
QURT_EMEM – Not enough memory to create the timer.
QURT_EINVALID – One of the arguments in the attr field is invalid.
Other error code – Operation failed.

Dependencies

None.

15.11 qurt_timer_delete()

15.11.1 Function Documentation

15.11.1.1 int qurt_timer_delete (qurt_timer_t *timer*)

Deletes the timer.

Destroys the specified timer and deallocates the timer object.

Associated data types

qurt_timer_t

Parameters

in	*timer*	Timer object.

Returns

QURT_EOK – Success.
QURT_EVAL – Argument passed is not a valid timer.

Dependencies

None.

15.12 qurt_timer_get_attr()

15.12.1 Function Documentation

15.12.1.1 int qurt_timer_get_attr (qurt_timer_t *timer,* qurt_timer_attr_t * *attr*)

Gets the timer attributes of the specified timer when it was created.

Associated data types

qurt_timer_t
qurt_timer_attr_t

Parameters

in	*timer*	Timer object.
out	*attr*	Pointer to the destination structure for timer attributes.

Returns

QURT_EOK – Success.
QURT_EVAL – Argument passed is not a valid timer.

Dependencies

None.

15.13 qurt_timer_group_disable()

15.13.1 Function Documentation

15.13.1.1 int qurt_timer_group_disable (unsigned int *group*)

Disables all timers that are assigned to the specified timer group. If a specified timer is already disabled, ignore it. If a specified timer is expired, do not process it. If the specified timer group is empty, do nothing.

NOTE When a timer is disabled its remaining time does not change, thus it cannot generate a timer event.

Parameters

in	*group*	Timer group identifier.

Returns

QURT_EOK – Success.

Dependencies

None.

15.14 qurt_timer_group_enable()

15.14.1 Function Documentation

15.14.1.1 int qurt_timer_group_enable (unsigned int *group*)

Enables all timers that are assigned to the specified timer group. If a specified timer is already enabled, ignore it. If a specified timer is expired, process it. If the specified timer group is empty, do nothing.

Parameters

in	*group*	Timer group identifier.

Returns

QURT_EOK – Success.

Dependencies

None.

15.15 qurt_timer_restart()

15.15.1 Function Documentation

15.15.1.1 int qurt_timer_restart (qurt_timer_t *timer,* qurt_timer_duration_t *duration*)

Restarts a stopped timer with the specified duration. The timer must be a one-shot timer. Timers stop after they have expired or after they are explicitly stopped with qurt_timer_stop(). A restarted timer expires after the specified duration, the starting time is when the function is called.

NOTE Timers stop after they have expired or after they are explicitly stopped with the timer stop operation, see Section 15.17.1.1.

Associated data types

qurt_timer_t
qurt_timer_duration_t

Parameters

in	*timer*	Timer object.
in	*duration*	Timer duration (in microseconds) before the restarted timer expires again. The valid range is QURT_TIMER_MIN_DURATION to QURT_TIMER_MAX_DURATION.

Returns

QURT_EOK – Success.
QURT_EINVALID – Invalid timer ID or duration value.
QURT_ENOTALLOWED – Timer is not a one-shot timer.
QURT_EMEM – Out-of-memory error.

Dependencies

None.

15.16 qurt_timer_sleep()

15.16.1 Function Documentation

15.16.1.1 int qurt_timer_sleep (qurt_timer_duration_t *duration*)

Suspends the current thread for the specified amount of time. The sleep duration value must be between QURT_TIMER_MIN_DURATION and QURT_TIMER_MAX_DURATION (Section 15).

Associated data types

qurt_timer_duration_t

Parameters

in	*duration*	Interval (in microseconds) between when the thread is suspended and when it is re-awakened.

Returns

QURT_EOK – Success.
QURT_EMEM – Not enough memory to perform the operation.

Dependencies

None.

15.17 qurt_timer_stop()

15.17.1 Function Documentation

15.17.1.1 int qurt_timer_stop (qurt_timer_t *timer*)

Stops a running timer. The timer must be a one-shot timer.

NOTE Restart stopped timers with the timer restart operation, see Section 15.15.1.1.

Associated data types

qurt_timer_t

Parameters

in	*timer*	Timer object.

Returns

QURT_EOK – Success.
QURT_EINVALID – Invalid timer ID or duration value.
QURT_ENOTALLOWED – Timer is not a one shot timer.
QURT_EMEM – Out of memory error.

Dependencies

None.

15.18 Timer Data Types

This section describes data types for timer services.

- Timers are represented in QuRT as objects of type qurt_timer_t.
- Timer attributes are stored in structures of type qurt_timer_attr_t.
- Timer durations are specified as values of type qurt_timer_duration_t.
- Timer times are specified as values of type qurt_timer_time_t.
- Timer types are specified as values of type qurt_timer_type_t.

15.18.1 Data Structure Documentation

15.18.1.1 struct qurt_timer_attr_t

QuRT timer attribute type.

15.18.2 Typedef Documentation

15.18.2.1 typedef unsigned int qurt_timer_t

QuRT timer type.

15.18.2.2 typedef unsigned long long qurt_timer_duration_t

QuRT timer duration type.

15.18.2.3 typedef unsigned long long qurt_timer_time_t

QuRT timer time type.

15.18.3 Enumeration Type Documentation

15.18.3.1 enum qurt_timer_type_t

QuRT timer types.

Enumerator:

QURT_TIMER_ONESHOT One shot.
QURT_TIMER_PERIODIC Periodic.

15.19 Timer Constants and Macros

15.19.1 Define Documentation

15.19.1.1 #define QURT_TIMER_DEFAULT_TYPE QURT_TIMER_ONESHOT

Default values. One shot.

15.19.1.2 #define QURT_TIMER_DEFAULT_DURATION 1000uL

Default duration.

15.19.1.3 #define QURT_TIMER_DEFAULT_EXPIRY 0uL

Default expiration.

15.19.1.4 #define QURT_TIMER_TIMETICK_FROM_US(*us*) QURT_SYSCLOCK_TIMETICK_FROM_US(us)

Conversion from microseconds to timer ticks.

15.19.1.5 #define QURT_TIMER_TIMETICK_TO_US(*ticks*) qurt_timer_timetick_to_us(ticks)

Conversion from timer ticks to microseconds at the nominal frequency.

15.19.1.6 #define QURT_TIMER_MIN_DURATION 100uL

Minimum microseconds value is 100 microseconds (sleep timer).

15.19.1.7 #define QURT_TIMER_MAX_DURATION QURT_SYSCLOCK_MAX_DURATION

Maximum microseconds value for Qtimer is 1,042,499 hours.

15.19.1.8 #define QURT_TIMER_MAX_DURATION_TICKS QURT_SYSCLOCK_MAX_DURATION_TICKS

Timer clock for Qtimer is 19.2 MHz.

15.19.1.9 #define QURT_TIMETICK_ERROR_MARGIN QURT_SYSCLOCK_ERROR_MARGIN

Sleep timer error margin for Qtimer is 1,000 ticks ∼52 us.

15.19.1.10 #define QURT_TIMER_MAX_GROUPS 5U

Maximum groups.

15.19.1.11 #define QURT_TIMER_DEFAULT_GROUP 0U

Default groups.

16 System Clock

Threads use the QuRT system clock to create alarms and timers, access the current system time, or determine when the next timer event occurs on any active timer.

The system clock time indicates how long (in terms of system ticks) the QuRT application system has been executing. A system tick is defined as one cycle of the Hexagon processor's 19.2 MHz QTIMER clock.

Unlike regular timers (Section 15), system clock alarms and timers are global resources, which can notify multiple client threads that a clock event has occurred. When a client thread registers for a system clock event, it specifies a signal object and signal mask.

System clock alarms expire at a specified time, while system clock timers expire after a specified duration. In both cases, when the event occurs, for each registered client thread the signal specified in the registered signal mask is set in the registered signal object.

The system clock supports the following operations:

- qurt_sysclock_get_hw_ticks()
- qurt_sysclock_get_hw_ticks_32()
- qurt_sysclock_get_hw_ticks_16()

16.1 qurt_sysclock_get_hw_ticks()

16.1.1 Function Documentation

16.1.1.1 unsigned long long qurt_sysclock_get_hw_ticks (void)

Gets the hardware tick count.

Returns the current value of a 64-bit hardware counter. The value wraps around to zero when it exceeds the maximum value.

NOTE This operation must be used with care because of the wrap-around behavior.

Returns

Integer – Current value of 64-bit hardware counter.

Dependencies

None.

16.2 qurt_sysclock_get_hw_ticks_32()

16.2.1 Function Documentation

16.2.1.1 static unsigned long qurt_sysclock_get_hw_ticks_32 (void)

Gets the hardware tick count in 32 bits.

Returns the current value of a 32-bit hardware counter. The value wraps around to zero when it exceeds the maximum value.

NOTE This operation is implemented as an inline C function, and should be called from a C/C++ program. The returned 32 bits are the lower 32 bits of the Qtimer counter.

Returns

Integer – Current value of the 32-bit timer counter.

Dependencies

None.

16.3 qurt_sysclock_get_hw_ticks_16()

16.3.1 Function Documentation

16.3.1.1 static unsigned short qurt_sysclock_get_hw_ticks_16 (void)

Gets the hardware tick count in 16 bits.

Returns the current value of a 16-bit timer counter. The value wraps around to zero when it exceeds the maximum value.

NOTE This operation is implemented as an inline C function, and should be called from a C/C++ program. The returned 16 bits are based on the value of the lower 32 bits in Qtimer counter, right shifted by 16 bits.

Returns

Integer – Current value of the 16-bit timer counter, calculated from the lower 32 bits in the Qtimer counter, right shifted by 16 bits.

Dependencies

None.

17 Interrupts

Threads use interrupts to respond to external events.

When registering an interrupt, it is both enabled and associated with the specified signal object and signal mask. When an interrupt occurs, the signal specified in the signal mask is set in the signal object. To handle the interrupt, an interrupt service thread (IST) conventionally waits on that signal.

Interrupts are automatically disabled after they occur. To re-enable an interrupt, an IST performs the acknowledge interrupt operation after it has finished processing the interrupt and just before suspending itself (for example, by waiting on the interrupt signal). When an interrupt is deregistered, it is disabled and no longer associated with any signal.

Up to 31 separate interrupts can be registered to a single signal object, as determined by the number of individual signals the object can store. Signal 31 is reserved by QuRT. Thus a single IST can handle several different interrupts.

Note: Only one signal object can be registered to a specific interrupt. Registering multiple signal objects on an interrupt raises an exception (Section 19).

Threads that serve as ISTs must not call the exit thread operation.

Interrupts do not support init and destroy operations because no objects (Section 2.4) are created for them.

Explicity clear a pending interrupt with the clear interrupt operation.

Note: This operation is intended for system-level use, and must be used with care.

All interrupts are based on the L2VIC interrupt controller. Specify all interrupts using the L2VIC interrupt numbers.

L2VIC interrupts can be configured dynamically to have different types (edge-triggered or level-triggered) or polarities (active-low or active-high).

Note: L2VIC interrupts must be deregistered before they can be reconfigured.

Interrupts are processor resources, which support the following operations:

- qurt_interrupt_acknowledge()
- qurt_interrupt_clear()
- qurt_interrupt_deregister()
- qurt_interrupt_disable()
- qurt_interrupt_enable()
- qurt_interrupt_get_config()
- qurt_interrupt_raise()

- qurt_interrupt_register()
- qurt_interrupt_register2()
- qurt_interrupt_set_config()
- qurt_interrupt_set_config2()
- qurt_interrupt_set_config3()
- qurt_interrupt_status()
- qurt_interrupt_get_status()
- qurt_interrupt_raise2()
- Constants
- Interrupt types

17.1 qurt_interrupt_acknowledge()

17.1.1 Function Documentation

17.1.1.1 int qurt_interrupt_acknowledge (int *int_num*)

Acknowledges an interrupt after it has been processed.

Re-enables an interrupt and clears its pending status. This is done after an interrupt is processed by an IST.

Interrupts are automatically disabled after they occur. To re-enable an interrupt, an IST performs the acknowledge operation after it has finished processing the interrupt and just before suspending itself (such as by waiting on the interrupt signal).

NOTE To prevent losinhg or reprocessing subsequent occurrences of the interrupt, an IST must clear the interrupt signal (Section 9.1.1.1) before acknowledging the interrupt.

Parameters

in	*int_num*	Interrupt that is being re-enabled.

Returns

QURT_EOK – Interrupt acknowledge was successful.
QURT_EDEREGISTERED – Interrupt is already de-registered.

Dependencies

None.

17.2 qurt_interrupt_clear()

17.2.1 Function Documentation

17.2.1.1 unsigned int qurt_interrupt_clear (int *int_num*)

Clears the pending status of the specified interrupt.

NOTE This operation is intended for system-level use, and must be used with care.

Parameters

in	*int_num*	Interrupt that is being re-enabled.

Returns

QURT_EOK – Success.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

17.3 qurt_interrupt_deregister()

17.3.1 Function Documentation

17.3.1.1 unsigned int qurt_interrupt_deregister (int *int_num*)

Disables the specified interrupt and disassociates it from a QuRT signal object. If the specified interrupt was never registered (Section 17.8.1.1), the deregister operation returns the status value QURT_EINT.

NOTE If an interrupt is deregistered while an IST waits to receive it, the IST might wait indefinitely for the interrupt to occur. To avoid this problem, the QuRT kernel sends the signal SIG_INT_ABORT to awaken an IST after determining that it has no interrupts registered.

Parameters

in	*int_num*	L2VIC to deregister; valid range is 0 to 1023.

Returns

QURT_EOK – Success.
QURT_EINT – Invalid interrupt number (not registered).

Dependencies

None.

17.4 qurt_interrupt_disable()

17.4.1 Function Documentation

17.4.1.1 unsigned int qurt_interrupt_disable (int *int_num*)

Disables an interrupt with its interrupt number.

The interrupt must be registered prior to calling this function. After qurt_interrupt_disable() returns, the Hexagon subsystem can no longer send the corresponding interrupt to the Hexagon core, until qurt_interrupt_enable() is called for the same interrupt.

Avoid calling qurt_interrupt_disable() and qurt_interrupt_enable() frequently within a short period of time.

- A pending interrupt can already be in the Hexagon core when qurt_interrupt_disable() is called. Therefore, some time later, the pending interrupt is received on a Hexagon hardware thread.
- After the Hexagon subsystem sends an interrupt to the Hexagon core, the Hexagon hardware automatically disables the interrupt until kernel software re-enables the interrupt at the interrupt acknowledgement stage. If qurt_interrupt_enable() is called from a certain thread at an ealier time, the interrupt is re-enabled earlier and can trigger sending a new interrupt to the Hexagon core while kernel software is still processing the previous interrupt.

Parameters

in	*int_num*	Interrupt number.

Returns

QURT_EOK – Interrupt successfully disabled.
QURT_EINT – Invalid interrupt number.
QURT_ENOTALLOWED – Interrupt is locked.
QURT_EVAL – Interrupt is not registered.

Dependencies

None.

17.5 qurt_interrupt_enable()

17.5.1 Function Documentation

17.5.1.1 unsigned int qurt_interrupt_enable (int *int_num*)

Enables an interrupt with its interrupt number.

The interrupt must be registered prior to calling this function.

Parameters

in	*int_num*	Interrupt number.

Returns

QURT_EOK – Interrupt successfully enabled.
QURT_EINT – Invalid interrupt number.
QURT_ENOTALLOWED – Interrupt is locked.
QURT_EVAL – Interrupt is not registered.

Dependencies

None.

17.6 qurt_interrupt_get_config()

17.6.1 Function Documentation

17.6.1.1 unsigned int qurt_interrupt_get_config (unsigned int *int_num,* unsigned int * *int_type,* unsigned int * *int_polarity*)

Gets the L2VIC interrupt configuration.

This function returns the type and polarity of the specified L2VIC interrupt.

Parameters

in	*int_num*	L2VIC interrupt that is being re-enabled.
out	*int_type*	Pointer to an interrupt type. 0 – Level-triggered interrupt 1 – Eedge-triggered interrupt
out	*int_polarity*	Pointer to interrupt polarity. 0 – Active-high interrupt 1 – Active-low interrupt.

Returns

QURT_EOK – Configuration successfully returned.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

17.7 qurt_interrupt_raise()

17.7.1 Function Documentation

17.7.1.1 int qurt_interrupt_raise (unsigned int *interrupt_num*)

Raises the interrupt.

This function triggers a level-triggered L2VIC interrupt, and accepts interrupt numbers in the range of 0 to 1023.

Parameters

in	*interrupt_num*	Interrupt number.

Returns

QURT_EOK – Success
-1 – Failure; the interrupt is not supported.

Dependencies

None.

17.8 qurt_interrupt_register()

17.8.1 Function Documentation

17.8.1.1 unsigned int qurt_interrupt_register (int *int_num,* qurt_anysignal_t * *int_signal,* int *signal_mask*)

Registers the interrupt.

Enables the specified interrupt and associates it with the specified QuRT signal object and signal mask.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not to wait.

When the interrupt occurs, the signal specified in the signal mask is set in the signal object. An IST conventionally waits on that signal to handle the interrupt. The thread that registers the interrupt is set as the IST.

Up to 31 separate interrupts can be registered to a single signal object, as determined by the number of individual signals the object can store. QuRT reserves signal 31. Thus a single IST can handle several different interrupts.

QuRT reserves some interrupts for internal use – the remainder are available for use by applications, and thus are valid interrupt numbers. If the specified interrupt number is outside the valid range, the register operation returns the status value QURT_EINT.

Only one thread can be registered at a time to a specific interrupt. Attempting to register an already-registered interrupt returns the status value QURT_EVAL.

Only one signal bit in a signal object can be registered at a time to a specific interrupt. Attempting to register multiple signal bits to an interrupt returns the status value QURT_ESIG.

When the signal registers an interrupt, QuRT can only set its signal bits when receiving the interrupt. The QuRT signal API from another software thread cannot set the signal even for unused signal bits.

NOTE The valid range for an interrupt number can differ on target execution environments other than the simulator. For more information, see the appropriate hardware document.

Associated data types

qurt_anysignal_t

Parameters

in	*int_num*	L2VIC interrupt to deregister; valid range is 0 to 1023.
in	*int_signal*	Any-signal object to wait on (Section 9).
in	*signal_mask*	Signal mask value indicating signal to receive the interrupt.

Returns

QURT_EOK – Interrupt successfully registered.
QURT_EINT – Invalid interrupt number.
QURT_ESIG – Invalid signal bitmask (cannot set more than one signal at a time).
QURT_EVAL – Interrupt already registered.

Dependencies

None.

17.9 qurt_interrupt_register2()

17.9.1 Function Documentation

17.9.1.1 unsigned int qurt_interrupt_register2 (int *int_num,* qurt_anysignal_t * *int_signal,* int *signal_mask,* unsigned int *flags*)

Registers the interrupt.

Enables the specified interrupt, associates it with the specified QuRT signal object and signal mask, and sets interrupt flags.

Signals are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that a signal must be waited on, and 0 indicates not to wait.

When the interrupt occurs, the signal specified in the signal mask is set in the signal object. An IST conventionally waits on that signal to handle the interrupt. The thread that registers the interrupt is set as the IST.

Up to 31 separate interrupts can be registered to a single signal object, as determined by the number of individual signals that the object can store. QuRT reserves signal 31. Thus a single IST can handle several different interrupts.

QuRT reserves some interrupts for internal use – the remainder are available for use by applications, and thus are valid interrupt numbers. If the specified interrupt number is outside the valid range, the register operation returns the status value QURT_EINT.

Only one thread can be registered at a time to a specific interrupt. Attempting to register an already-registered interrupt returns the status value QURT_EVAL.

Only one signal bit in a signal object can be registered at a time to a specific interrupt. Attempting to register multiple signal bits to an interrupt returns the status value QURT_ESIG.

When the signal registers an interrupt, QuRT can only set its signal bits when receiving the interrupt. The QuRT signal API from another software thread cannot set the signal even for unused signal bits.

NOTE The valid range for an interrupt number can differ on target execution environments other than the simulator. For more information, see the appropriate hardware document.

Associated data types

qurt_anysignal_t

Parameters

in	*int_num*	L2VIC interrupt to deregister; valid range is 0 to 1023.
in	*int_signal*	Any-signal object to wait on (Section 9).
in	*signal_mask*	Signal mask value indicating signal to receive the interrupt.
in	*flags*	Defines interrupt property, supported property is interrupt lock enable/disable. Possible values for flags: • QURT_INT_LOCK_ENABLE • QURT_INT_LOCK_DISABLE

Returns

QURT_EOK – Interrupt successfully registered.
QURT_EINT – Invalid interrupt number.
QURT_ESIG – Invalid signal bitmask (cannot set more than one signal at a time).
QURT_EVAL – Interrupt already registered.

Dependencies

None.

17.10 qurt_interrupt_set_config()

17.10.1 Function Documentation

17.10.1.1 unsigned int qurt_interrupt_set_config (unsigned int *int_num,* unsigned int *int_type,* unsigned int *int_polarity*)

Sets the type and polarity of the specified L2VIC interrupt.

NOTE Deregister L2VIC interrupts before reconfiguring them.

Parameters

in	*int_num*	L2VIC interrupt that is being re-enabled.
in	*int_type*	Interrupt type. 0 – Level-triggered interrupt 1 – Edge-triggered interrupt
in	*int_polarity*	Interrupt polarity. 0 – Active-high interrupt 1 – Active-low interrupt

Returns

QURT_EOK – Success.
QURT_ENOTALLOWED – Not allowed; the interrupt is being registered.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

17.11 qurt_interrupt_set_config2()

17.11.1 Function Documentation

17.11.1.1 unsigned int qurt_interrupt_set_config2 (unsigned int *int_num,* unsigned int *int_type*)

Sets the type and polarity of the specified L2VIC interrupt.

NOTE L2VIC interrupts must be deregistered before they can be reconfigured.

Parameters

in	*int_num*	L2VIC interrupt that is being re-enabled.
in	*int_type*	Notified to the hardware configuration callback function and used to modify the L2VIC type. Possible values: • QURT_INT_TRIGGER_USE_DEFAULT • QURT_INT_TRIGGER_LEVEL_HIGH • QURT_INT_TRIGGER_LEVEL_LOW • QURT_INT_TRIGGER_RISING_EDGE • QURT_INT_TRIGGER_FALLING_EDGE • QURT_INT_TRIGGER_DUAL_EDGE

Returns

QURT_EOK – Success.
QURT_ENOTALLOWED – Not allowed; the interrupt is being registered.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

17.12 qurt_interrupt_set_config3()

17.12.1 Function Documentation

17.12.1.1 unsigned int qurt_interrupt_set_config3 (unsigned int *int_num,* unsigned int *config_id,* unsigned int *config_val*)

Sets the specified configuration value for the specified property of the specified L2VIC interrupt.

NOTE L2VIC interrupts must be deregistered before they can be reconfigured for polarity.

Parameters

in	*int_num*	L2VIC interrupt to re-enable.
in	*config_id*	Property to configure: • QURT_INT_CONFIGID_POLARITY • QURT_INT_CONFIGID_LOCK
in	*config_val*	Dependent on the second argument config_id, specifies the value to set. Values for QURT_INT_CONFIGID_POLARITY: • QURT_INT_TRIGGER_USE_DEFAULT • QURT_INT_TRIGGER_LEVEL_HIGH • QURT_INT_TRIGGER_LEVEL_LOW • QURT_INT_TRIGGER_RISING_EDGE • QURT_INT_TRIGGER_FALLING_EDGE • QURT_INT_TRIGGER_DUAL_EDGE

Values for QURT_INT_CONFIGID_LOCK:

- QURT_INT_LOCK_ENABLE
- QURT_INT_LOCK_DISABLE

Returns

QURT_EOK – Success.
QURT_ENOTALLOWED – Not allowed; the interrupt is being registered or is locked for enable/disable.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

17.13 qurt_interrupt_status()

17.13.1 Function Documentation

17.13.1.1 unsigned int qurt_interrupt_status (int *int_num,* int * *status*)

Returns a value that indicates the pending status of the specified interrupt.

Parameters

in	*int_num*	Interrupt number that is being checked.
out	*status*	Interrupt status; 1 indicates that an interrupt is pending, 0 indicates that an interrupt is not pending.

Returns

QURT_EOK – Success.
QURT_EINT – Failure; invalid interrupt number.

Dependencies

None.

17.14 qurt_interrupt_get_status()

17.14.1 Function Documentation

17.14.1.1 unsigned int qurt_interrupt_get_status (int *int_num,* int *status_type,* int * *status*)

Gets the status of the specified interrupt in L2VIC.

Parameters

in	*int_num*	Interrupt number that is being checked.
in	*status_type*	0 – interrupt pending status 1 – interrupt enabling status
out	*status*	0 – OFF 1 – ON

Returns

QURT_EOK – Success.
QURT_EINT – Failure; invalid interrupt number.

Dependencies

None.

17.15 qurt_interrupt_raise2()

17.15.1 Function Documentation

17.15.1.1 unsigned long long qurt_interrupt_raise2 (unsigned int *interrupt_num*)

Raises the interrupt and returns the current pcycle value.

Parameters

in	*interrupt_num*	Interrupt number.

Returns

0xFFFFFFFFFFFFFFFF – Failure; the interrupt is not supported.
Other value – pcycle count at the time the interrupt is raised.

Dependencies

None.

17.16 Constants

17.16.1 Define Documentation

17.16.1.1 #define SIG_INT_ABORT 0x80000000

17.16.1.2 #define QURT_INT_CONFIGID_POLARITY 0x1U

QuRT interrupt property.

17.16.1.3 #define QURT_INT_CONFIGID_LOCK 0x2U

17.16.1.4 #define QURT_INT_LOCK_DEFAULT 0x0

QuRT interrupt lock. Default.

17.16.1.5 #define QURT_INT_LOCK_DISABLE 0x0

Interrupt can be enabled or disabled or deregistered.

17.16.1.6 #define QURT_INT_LOCK_ENABLE 0x1

Interrupt is locked and cannot be enabled, disabled, or deregistered.

17.17 Interrupt types

17.17.1 Define Documentation

17.17.1.1 #define QURT_INT_TRIGGER_TYPE_SET(*pol, edge*) ((((pol) & 0x01U) << 2) | ((edge) & 0x03U))

Trigger type bit fields for a PDC interrupt:

```
Polarity  Edge  Output\n
0         00    Level sensitive active low
0         01    Rising edge sensitive
0         10    Falling edge sensitive
0         11    Dual edge sensitive
1         00    Level sensitive active high
1         01    Falling edge sensitive
1         10    Rising edge sensitive
1         11    Dual edge sensitive
```

17.17.1.2 #define QURT_INT_TRIGGER_LEVEL_LOW QURT_INT_TRIGGER_TYPE_SET(0U, 0x00U)

17.17.1.3 #define QURT_INT_TRIGGER_LEVEL_HIGH QURT_INT_TRIGGER_TYPE_SET(1U, 0x00U)

17.17.1.4 #define QURT_INT_TRIGGER_RISING_EDGE QURT_INT_TRIGGER_TYPE_SET(1U, 0x02U)

17.17.1.5 #define QURT_INT_TRIGGER_FALLING_EDGE QURT_INT_TRIGGER_TYPE_SET(0U, 0x02U)

17.17.1.6 #define QURT_INT_TRIGGER_DUAL_EDGE QURT_INT_TRIGGER_TYPE_SET(0U, 0x03U)

17.17.1.7 #define QURT_INT_TRIGGER_USE_DEFAULT 0xffU

18 Thread Local Storage

Threads use thread local storage to allocate global storage, which is private to specific threads.

Data items stored in thread local storage can be accessed by any function in a thread (but not by any function outside the thread). As with global storage, the stored data items persist for as long as the thread exists. Destructor functions can be defined that process the stored data items when a thread terminates.

Note: Deleting a key does not run any destructor function that is associated with it.

Memory used for thread local storage is automatically allocated by QuRT. The QuRT thread local storage service is POSIX-compatible.

Thread local storage keys in QuRT are identified by values of type int.

Thread local storage supports the following operations:

- qurt_tls_create_key()
- qurt_tls_delete_key()
- qurt_tls_get_specific()
- qurt_tls_set_specific()

18.1 qurt_tls_create_key()

18.1.1 Function Documentation

18.1.1.1 int qurt_tls_create_key (int * *key*, void(*)(void *) *destructor*)

Creates a key for accessing a thread local storage data item.

Subsequent get and set operations use the key value.

NOTE The destructor function performs any clean-up operations needed by a thread local storage item when its containing thread is deleted (Section 3.14.1.1).

Parameters

out	*key*	Pointer to the newly created thread local storage key value.
in	*destructor*	Pointer to the key-specific destructor function. Passing NULL specifies that no destructor function is defined for the key.

Returns

QURT_EOK – Key successfully created.
QURT_ETLSAVAIL – No free TLS key available.

Dependencies

None.

18.2 qurt_tls_delete_key()

18.2.1 Function Documentation

18.2.1.1 int qurt_tls_delete_key (int *key*)

Deletes the specified key from thread local storage.

NOTE Explicitly deleting a key does not execute any destructor function that is associated with the key (Section 18.1.1.1).

Parameters

in	*key*	Thread local storage key value to delete.

Returns

QURT_EOK – Key successfully deleted.
QURT_ETLSENTRY – Key already free.

Dependencies

None.

18.3 qurt_tls_get_specific()

18.3.1 Function Documentation

18.3.1.1 void* qurt_tls_get_specific (int *key*)

Loads the data item from thread local storage.

Returns the data item that is stored in thread local storage with the specified key. The data item is always a pointer to user data.

Parameters

in	*key*	Thread local storage key value.

Returns

Pointer – Data item indexed by key in thread local storage.
0 (NULL) – Key out of range.

Dependencies

None.

18.4 qurt_tls_set_specific()

18.4.1 Function Documentation

18.4.1.1 int qurt_tls_set_specific (int *key,* const void * *value*)

Stores a data item to thread local storage along with the specified key.

Parameters

in	*key*	Thread local storage key value.
in	*value*	Pointer to user data value to store.

Returns

QURT_EOK – Data item successfully stored.
QURT_EINVALID – Invalid key.
QURT_EFAILED – Invoked from a non-thread context.

19 Exception Handling

QuRT supports exception handling for software errors and processor-detected hardware exceptions. Exceptions are treated as either fatal or nonfatal, and handled accordingly.

QuRT handles program exceptions (fatal or nonfatal), kernel exceptions, and imprecise exceptions.

QuRT program code raises program exceptions – they include cases such as page faults, misaligned load/store operations, and other Hexagon processor exceptions. The QuRT API can also explicitly raise program exceptions.

A thread (Section 3) registered as the program exception handler handles the program exceptions.

Nonfatal program exceptions cause QuRT to take the following actions:

- Save the context of the relevant hardware thread in the task control block (TCB).
- Schedule the registered program exception handler thread (if any), with the error information assigned to the parameters of the wait for exception operation.

A program exception handler can handle a nonfatal exception either by reloading the QuRT program (if it has the ability), or by terminating the execution of the QuRT program system.

Note: If no program exception handler is registered, or if the registered handler calls raise nonfatal exception, QuRT raises a kernel exception.

If a thread runs in Supervisor mode, errors are treated as kernel exceptions.

If multiple program exceptions occur, exceptions are forwarded to the program exception handler in the order that the exceptions occur. The exception handler must make repeated calls to qurt_exception_wait() to process the error information from the queued exceptions.

Fatal program exceptions terminate the execution of the QuRT program system without invoking the program exception handler. Use fatal program exceptions where the program handles the system shutdown operations.

Fatal exceptions are raised by calling qurt_exception_raise_fatal(), which masks the Hexagon processor interrupts and stops the other hardware threads in the Hexagon processor. This operation returns so the program can then perform the necessary program-level shutdown operations (data logging, and so on).

When the program is ready to shut down the system, it calls the fatal shutdown operation to performs the following actions:

1. If the raise fatal exception operation was not already called, mask the processor interrupts and stop the other hardware threads.
2. Save the contexts of all hardware threads.
3. Save the contents of TCM.
4. Save all TLB entries.

5. Flush the caches and update cache flush status.
6. Call the registered fatal notification handler.
7. Execute an infinite loop in the current hardware thread.

The QuRT kernel raises kernel exceptions – they include Supervisor mode exceptions along with page faults and other Hexagon processor exceptions.

Kernel exceptions cause QuRT to terminate the execution of the program system and shut down the system processor, while saving the processor state to assist with investigations of the problem that caused the exception.

A kernel exception causes QuRT to perform the following actions:

1. Save the context of the current hardware thread to the kernel error data structure.
2. Save the contexts of other active hardware threads to their respective TCBs.
3. Stop the other hardware threads.
4. Wait until the other hardware threads stop.
5. Flush the Hexagon processor cache.
6. Mask the Hexagon processor interrupts.
7. Call the registered fatal notification handler.
8. Execute an infinite loop in the current hardware thread.

Note: Kernel exceptions do not invoke the program exception handler.

Imprecise exceptions are serious and unrecoverable error conditions that can be raised in either the QuRT kernel or the program code – they include cases such as stores to bad addresses, hardware parity errors, or other imprecise slave error conditions, and non-maskable interrupt (NMI) exceptions raised from outside the Hexagon processor.

QuRT does not forward imprecise exceptions to the program exception handler. Instead the kernel terminates the execution of the current hardware thread while saving the processor state.

When an imprecise exception occurs, QuRT performs the same procedure used for a kernel exception, except that the thread contexts for all hardware threads are stored in the kernel error data structure.

Note: The imprecise exception handler overwrites Hexagon processor register R23. This does not occur with program or kernel exceptions.

For floating point exceptions, User programs can selectively enable specific floating point events (inexact, underflow, overflow, divide by zero, and invalid) to generate QuRT program exceptions.

Program exception handling supports the following operations:

- qurt_exception_enable_fp_exceptions()
- qurt_exception_raise_fatal()
- qurt_exception_raise_nonfatal()
- qurt_exception_wait()
- qurt_exception_wait3()
- qurt_assert_error()

19.1 qurt_exception_enable_fp_exceptions()

19.1.1 Function Documentation

19.1.1.1 static unsigned int qurt_exception_enable_fp_exceptions (unsigned int *mask*)

Enables the specified floating point exceptions as QuRT program exceptions.

The exceptions are enabled by setting the corresponding bits in the Hexagon control user status register (USR).

The mask argument specifies a mask value identifying the individual floating point exceptions to set. The exceptions are represented as defined symbols that map into bits 0 through 31 of the 32-bit flag value. Multiple floating point exceptions are specified by OR'ing together the individual exception symbols.

NOTE This function must be called before performing any floating point operations.

Parameters

in	*mask*	Floating point exception types. Values: • QURT_FP_EXCEPTION_ALL • QURT_FP_EXCEPTION_INEXACT • QURT_FP_EXCEPTION_UNDERFLOW • QURT_FP_EXCEPTION_OVERFLOW • QURT_FP_EXCEPTION_DIVIDE0 • QURT_FP_EXCEPTION_INVALID

Returns

Updated contents of the USR.

Dependencies

None.

19.2 qurt_exception_raise_fatal()

19.2.1 Function Documentation

19.2.1.1 void qurt_exception_raise_fatal (void)

Raises a fatal program exception in the QuRT system.

Fatal program exceptions terminate the execution of the QuRT system without invoking the program exception handler.

For more information on fatal program exceptions, see Section 19.

This operation always returns, so the calling program can perform the necessary shutdown operations (data logging, on so on).

NOTE Context switches do not work after this operation has been called.

Returns

None.

Dependencies

None.

19.3 qurt_exception_raise_nonfatal()

19.3.1 Function Documentation

19.3.1.1 int qurt_exception_raise_nonfatal (int *error*)

Raises a nonfatal program exception in the QuRT program system.

For more information on program exceptions, see Section 19.

This operation never returns – the program exception handler is assumed to perform all exception handling before terminating or reloading the QuRT program system.

NOTE The C library function abort() calls this operation to indicate software errors.

Parameters

in	*error*	QuRT error result code (Section 26).

Returns

Integer – Unused.

Dependencies

None.

19.4 qurt_exception_wait()

19.4.1 Function Documentation

19.4.1.1 unsigned int qurt_exception_wait (unsigned int * *ip*, unsigned int * *sp*, unsigned int * *badva*, unsigned int * *cause*)

Registers the program exception handler. This function assigns the current thread as the QuRT program exception handler and suspends the thread until a program exception occurs.

When a program exception occurs, the thread is awakened with error information assigned to the parameters of this operation.

NOTE If no program exception handler is registered, or if the registered handler calls exit, QuRT raises a kernel exception. If a thread runs in Supervisor mode, any errors are treated as kernel exceptions.

Parameters

out	*ip*	Pointer to the instruction memory address where the exception occurred.
out	*sp*	Stack pointer.
out	*badva*	Pointer to the virtual data address where the exception occurred.
out	*cause*	Pointer to the QuRT error result code.

Returns

Registry status:
Thread identifier – Handler successfully registered.
QURT_EFATAL – Registration failed.

Dependencies

None.

19.5 qurt_exception_wait3()

19.5.1 Function Documentation

19.5.1.1 unsigned int qurt_exception_wait3 (void * *sys_err,* unsigned int *sys_err_size*)

Registers the current thread as the QuRT program exception handler, and suspends the thread until a program exception occurs. When a program exception occurs, the thread is awakened with error information assigned to the specified error event record. If a program exception is raised when no handler is registered (or when a handler is registered, but it calls exit), the exception is treated as fatal.

NOTE If a thread runs in Monitor mode, all exceptions are treated as kernel exceptions.

This function differs from qurt_exception_wait() by returning the error information in a data structure rather than as individual variables. It also returns additional information (for example, SSR, FP, and LR).

Parameters

in	*sys_err*	Pointer to the qurt_sysevent_error_1_t type structure.
in	*sys_err_size*	qurt_sysevent_error_1_t type structure.

Returns

Registry status:

- QURT_EFATAL – Failure.
- Thread ID – Success.

Dependencies

None.

19.6 qurt_assert_error()

19.6.1 Function Documentation

19.6.1.1 void qurt_assert_error (const char * *filename,* int *lineno*)

Writes diagnostic information to the debug buffer, and raises an error to the QuRT kernel.

Associated data types

None.

Parameters

in	*filename*	Pointer to the file name string.
in	*lineno*	Line number.

Returns

None.

Dependencies

None.

20 Memory Allocation

QuRT user programs are assigned a default global heap, which is accessed by the standard C functions malloc and free (Section 2.1).

Threads use memory allocation to create additional heap-based storage allocators within user programs.

Note: Memory allocation cannot allocate memory outside the thread assigned memory area (Section 2.1). This is done using the QuRT memory management services (Section 21).

Memory allocation supports the following operations:

- qurt_calloc()
- qurt_free()
- qurt_malloc()
- qurt_realloc()

20.1 qurt_calloc()

20.1.1 Function Documentation

20.1.1.1 void* qurt_calloc (unsigned int *elsize,* unsigned int *num*)

Dynamically allocates the specified array on the QuRT system heap. The return value is the address of the allocated array.

NOTE The allocated memory area is automatically initialized to zero.

Parameters

in	*elsize*	Size (in bytes) of each array element.
in	*num*	Number of array elements.

Returns

Nonzero – Pointer to allocated array.
Zero – Not enough memory in heap to allocate array.

Dependencies

None.

20.2 qurt_free()

20.2.1 Function Documentation

20.2.1.1 void qurt_free (void * *ptr*)

Frees allocated memory from the heap.

Deallocates the specified memory from the QuRT system heap.

Parameters

in	*ptr	Pointer to the address of the memory to deallocate.

Returns

None.

Dependencies

The memory item that the ptr value specifies must have been previously allocated using one of the qurt_calloc(), qurt_malloc(), or qurt_realloc() memory allocation functions. Otherwise the behavior of QuRT is undefined.

20.3 qurt_malloc()

20.3.1 Function Documentation

20.3.1.1 void* qurt_malloc (unsigned int *size*)

Dynamically allocates the specified array on the QuRT system heap. The return value is the address of the allocated memory area.

NOTE The allocated memory area is automatically initialized to zero.

Parameters

in	*size*	Size (in bytes) of the memory area.

Returns

Nonzero – Pointer to the allocated memory area.
0 – Not enough memory in heap to allocate memory area.

Dependencies

None.

20.4 qurt_realloc()

20.4.1 Function Documentation

20.4.1.1 void* qurt_realloc (void * *ptr,* int *newsize*)

Reallocates memory on the heap.

Changes the size of a memory area that is already allocated on the QuRT system heap. The reallocate memory operation is functionally similar to realloc. It accepts a pointer to an existing memory area on the heap, and resizes the memory area to the specified size while preserving the original contents of the memory area.

NOTE This function might change the address of the memory area. If the value of ptr is NULL, this function is equivalent to qurt_malloc(). If the value of new_size is 0, it is equivalent to qurt_free(). If the memory area is expanded, the added memory is not initialized.

Parameters

in	*ptr	Pointer to the address of the memory area.
in	*newsize*	Size (in bytes) of the reallocated memory area.

Returns

Nonzero – Pointer to reallocated memory area.
0 – Not enough memory in heap to reallocate the memory area.

Dependencies

None.

21 Memory Management

Threads use memory management to dynamically allocate user program memory, share memory with other user programs, and manage virtual memory.

To dynamically allocate memory outside its assigned memory area (Section 2.1), a thread initializes a memory pool by attaching it to a predefined pool. It then creates one or more memory regions with the pool specified as one of the region attributes. The thread can access the memory in the newly allocated memory regions.

Note: A user program cannot share its original assigned memory with another user program – it can only share dynamically-allocated memory regions.

Memory pools assign memory regions to different types of physical (not virtual) memory. For example, the Hexagon processor can access SMI, TCM, and EBI memory. To allocate regions in each of these memories, define a separate memory pool for each memory unit (for example,an SMI pool or TCM pool). Requests to create memory regions always specify a memory pool object as a region attribute.

QuRT predefines the memory pool object qurt_mem_default_pool(), which is preattached to the default memory pool in the system configuration file. It is defined to allocate memory regions in SMI memory. Memory pools are predefined in the system configuration file (Section 2.2), and are specified by their assigned pool name in memory pool attach operations. qurt_mem_pool_create() can create memory pools at runtime. All user programs in the QuRT user program system can access memory pools.

The add and remove pages operations are used to directly manipulate the memory pools.

Memory regions are used to define memory areas with a fixed set of attributes that specify virtual memory mapping and cache type of an area. A core set of regions is predefined in the system configuration file (Section 2.2), with additional regions created or deleted at run time to support dynamic memory management.

Memory regions have the following attributes:

- Size: Memory region size (in bytes).
- Pool: Memory pool that the region belongs to; each region must have a corresponding pool.
- Mapping: Memory mapping indicates how the memory region is mapped in virtual memory:
  - Virtual mapped regions have their virtual address (VA) range mapped to an available contiguous area of physical memory. This makes the most efficient use of virtual memory, and works for most memory use cases.
  - Physical contiguous mapped regions have their virtual address range mapped to a specific contiguous area of physical memory. This is necessary when the memory region is accessed by external devices that bypass Hexagon virtual memory addressing.
- Physical address: The physical base address of the memory region; it is set only when using physical-contiguous-mapped memory regions.

- Virtual address: Memory region virtual address; a read-only attribute that returns the base address of the memory region.
- Cache mode: Cache type indicating whether the memory region uses the instruction or data cache..
- Bus: Bus attributes indicate the (A1, A0) bus attribute bits.
- Type: Memory region type (local/shared); indicates whether the memory region is local to a user program or shared between user programs.

Note: The memory region size and pool attributes are set directly as parameters in the memory region create operation.

Memory region attributes are set both before a region is created (using the qurt_mem_region_attr_init() and the qurt_mem_region_attr_set functions) and when a region is created (by directly passing the attributes as arguments to qurt_mem_region_create()).

The memory region size and memory region pool are set when a region is created – other memory region attributes are set before the create operation.

Memory region attributes can be retrieved from a created memory region using qurt_mem_region_attr_get() and the other qurt_mem_region_attr_get functions.

The only attribute that cannot be retrieved from a memory region is the memory pool.

Memory maps specify the mapping between virtual memory and physical memory in the Hexagon processor.

The create mapping and remove mapping operations directly manipulate the memory maps.

The memory map static query operation indicates whether a memory page is statically mapped. If the specified page is statically mapped, the operation returns the virtual address of the page. If the page is not statically mapped (or if it does not exist), the operation returns -1 as the virtual address value.

The lookup physical address operation performs virtual to physical address translation. It returns the physical memory address of the specified virtual address.

Note: Memory maps operate directly on the page table – therefore, changing the map can affect any memory region defined for the affected memory area.

Memory ordering

Some devices require synchronization of stores and loads when they are accessed. This synchronization can be done via qurt_mem_barrier() and qurt_mem_syncht().

The barrier operation ensures that previous memory transactions are globally observable before any future memory transactions are globally observable.

The syncht operation does not return until previous memory transactions (such as cached and uncached load, and store) that originated from the current thread are completed and globally observable.

Memory management services are accessed with the following QuRT functions.

- qurt_lookup_physaddr()
- qurt_lookup_physaddr2()
- qurt_lookup_physaddr_64()
- qurt_mapping_create()

- qurt_mapping_create_64()
- qurt_mapping_remove()
- qurt_mapping_remove_64()
- qurt_mem_barrier()
- qurt_mem_cache_clean()
- qurt_mem_cache_clean2()
- qurt_mem_cache_phys_clean()
- qurt_mem_configure_cache_partition()
- qurt_mem_l2cache_line_lock()
- qurt_mem_l2cache_line_unlock()
- qurt_mem_map_static_query()
- qurt_mem_map_static_query_64()
- qurt_mem_pool_add_pages()
- qurt_mem_pool_attach()
- qurt_mem_pool_attach2()
- qurt_mem_pool_attr_get()
- qurt_mem_pool_attr_get_addr()
- qurt_mem_pool_is_available()
- qurt_mem_pool_attr_get_size()
- qurt_mem_pool_create()
- qurt_mem_pool_remove_pages()
- qurt_mem_region_attr_get()
- qurt_mem_region_attr_get_bus_attr()
- qurt_mem_region_attr_get_cache_mode()
- qurt_mem_region_attr_get_mapping()
- qurt_mem_region_attr_get_physaddr()
- qurt_mem_region_attr_get_size()
- qurt_mem_region_attr_get_type()
- qurt_mem_region_attr_get_virtaddr()
- qurt_mem_region_attr_get_physaddr_64()
- qurt_mem_region_attr_init()
- qurt_mem_region_attr_set_bus_attr()
- qurt_mem_region_attr_set_cache_mode()

- qurt_mem_region_attr_set_mapping()
- qurt_mem_region_attr_set_physaddr()
- qurt_mem_region_attr_set_physaddr_64()
- qurt_mem_region_attr_set_type()
- qurt_mem_region_attr_set_virtaddr()
- qurt_mem_region_create()
- qurt_mem_region_delete()
- qurt_mem_region_query()
- qurt_mem_region_query_64()
- qurt_mem_syncht()
- Memory Management Data Types
- Memory Management Macros

21.1 qurt_lookup_physaddr()

21.1.1 Function Documentation

21.1.1.1 qurt_paddr_t qurt_lookup_physaddr (qurt_addr_t *vaddr*)

Translates a virtual memory address to the physical memory address to which it maps.

The lookup occurs in the process of the caller. Use qurt_lookup_physaddr2() to lookup the physical address of another process.

Associated data types

qurt_addr_t
qurt_paddr_t

Parameters

in	*vaddr*	Virtual address.

Returns

Nonzero – Physical address to which the virtual address is mapped.
0 – Virtual address not mapped.

Dependencies

None.

21.2 qurt_lookup_physaddr2()

21.2.1 Function Documentation

21.2.1.1 qurt_paddr_64_t qurt_lookup_physaddr2 (qurt_addr_t *vaddr,* unsigned int *pid*)

Translates the virtual memory address of the specified process to the 64-bit physical memory address to which it is mapped.

Associated data types

qurt_addr_t
qurt_paddr_64_t

Parameters

in	*vaddr*	Virtual address.
in	*pid*	PID.

Returns

Nonzero – 64-bit physical address to which the virtual address is mapped.
0 – Virtual address is not mapped.

Dependencies

None.

21.3 qurt_lookup_physaddr_64()

21.3.1 Function Documentation

21.3.1.1 qurt_paddr_64_t qurt_lookup_physaddr_64 (qurt_addr_t *vaddr*)

Translates a virtual memory address to the 64-bit physical memory address it is mapped to.

The lookup occurs in the process of the caller. Use qurt_lookup_physaddr2() to lookup the physical address of another process.

Associated data types

qurt_paddr_64_t
qurt_addr_t

Parameters

in	*vaddr*	Virtual address.

Returns

Nonzero – 64-bit physical address to which the virtual address is mapped.
0 – Virtual address has not been mapped.

Dependencies

None.

21.4 qurt_mapping_create()

21.4.1 Function Documentation

21.4.1.1 int qurt_mapping_create (qurt_addr_t *vaddr,* qurt_addr_t *paddr,* qurt_size_t *size,* qurt_mem_cache_mode_t *cache_attribs,* qurt_perm_t *perm*)

Creates a memory mapping in the page table. Not supported if called from a user process, always returns QURT_EMEM.

Associated data types

qurt_addr_t
qurt_size_t
qurt_mem_cache_mode_t
qurt_perm_t

Parameters

in	*vaddr*	Virtual address.
in	*paddr*	Physical address.
in	*size*	Size (4K-aligned) of the mapped memory page.
in	*cache_attribs*	Cache mode (writeback, and so on).
in	*perm*	Access permissions.

Returns

QURT_EOK – Mapping created.
QURT_EMEM – Failed to create mapping.

Dependencies

None.

21.5 qurt_mapping_create_64()

21.5.1 Function Documentation

21.5.1.1 int qurt_mapping_create_64 (qurt_addr_t *vaddr,* qurt_paddr_64_t *paddr_64,* qurt_size_t *size,* qurt_mem_cache_mode_t *cache_attribs,* qurt_perm_t *perm*)

Creates a memory mapping in the page table. Not supported if called from a user process, always returns QURT_EMEM.

Associated data types

qurt_addr_t
qurt_paddr_64_t
qurt_size_t
qurt_mem_cache_mode_t
qurt_perm_t

Parameters

in	*vaddr*	Virtual address.
in	*paddr_64*	64-bit physical address.
in	*size*	Size (4K-aligned) of the mapped memory page.
in	*cache_attribs*	Cache mode (writeback, and so on).
in	*perm*	Access permissions.

Returns

QURT_EOK – Success.
QURT_EMEM – Failure.

Dependencies

None.

21.6 qurt_mapping_remove()

21.6.1 Function Documentation

21.6.1.1 int qurt_mapping_remove (qurt_addr_t *vaddr,* qurt_addr_t *paddr,* qurt_size_t *size*)

Deletes the specified memory mapping from the page table.

Associated data types

qurt_addr_t
qurt_size_t

Parameters

in	*vaddr*	Virtual address.
in	*paddr*	Physical address.
in	*size*	Size of the mapped memory page (4K-aligned).

Returns

QURT_EOK – Mapping created.

Dependencies

None.

21.7 qurt_mapping_remove_64()

21.7.1 Function Documentation

21.7.1.1 int qurt_mapping_remove_64 (qurt_addr_t *vaddr,* qurt_paddr_64_t *paddr_64,* qurt_size_t *size*)

Deletes the specified memory mapping from the page table.

Associated data types

qurt_addr_t
qurt_paddr_64_t
qurt_size_t

Parameters

in	*vaddr*	Virtual address.
in	*paddr_64*	64-bit physical address.
in	*size*	Size of the mapped memory page (4K-aligned).

Returns

QURT_EOK – Success.

Dependencies

None.

21.8 qurt_mem_barrier()

21.8.1 Function Documentation

21.8.1.1 static void qurt_mem_barrier (void)

Creates a barrier for memory transactions.

This operation ensures that all previous memory transactions are globally observable before any future memory transactions are globally observable.

NOTE This operation is implemented as a wrapper for the Hexagon barrier instruction.

Returns

None

Dependencies

None.

21.9 qurt_mem_cache_clean()

21.9.1 Function Documentation

21.9.1.1 int qurt_mem_cache_clean (qurt_addr_t *addr*, qurt_size_t *size*, qurt_mem_cache_op_t *opcode*, qurt_mem_cache_type_t *type*)

Performs a cache clean operation on the data stored in the specified memory area. Peforms a syncht on all the data cache operations when the Hexagon processor version is V60 or greater.

NOTE Perform the flush all operation only on the data cache.

This operation flushes and invalidates the contents of all cache lines from start address to end address (start address + size). The contents of the adjoining buffer can be flushed and invalidated if it falls in any of the cache line.

Associated data types

qurt_addr_t
qurt_size_t
qurt_mem_cache_op_t
qurt_mem_cache_type_t

Parameters

in	*addr*	Address of data to flush.
in	*size*	Size (in bytes) of data to flush.
in	*opcode*	Type of cache clean operation. Values: • QURT_MEM_CACHE_FLUSH • QURT_MEM_CACHE_INVALIDATE • QURT_MEM_CACHE_FLUSH_INVALIDATE • QURT_MEM_CACHE_FLUSH_ALL **NOTE** QURT_MEM_CACHE_FLUSH_ALL is valid only when the type is QURT_MEM_DCACHE
in	*type*	Cache type. Values: • QURT_MEM_ICACHE • QURT_MEM_DCACHE

Returns

QURT_EOK – Cache operation performed successfully.
QURT_EVAL – Invalid cache type.

Dependencies

None.

21.10 qurt_mem_cache_clean2()

21.10.1 Function Documentation

21.10.1.1 int qurt_mem_cache_clean2 (qurt_addr_t *addr*, qurt_size_t *size*, qurt_mem_cache_op_t *opcode*, qurt_mem_cache_type_t *type*)

Performs a data cache clean operation on the data stored in the specified memory area.

This API only performs the following data cache operations:

- QURT_MEM_CACHE_FLUSH
- QURT_MEM_CACHE_INVALIDATE
- QURT_MEM_CACHE_FLUSH_INVALIDATE – flushes/invalidates the contents of all cache lines from start address to end address (start address + size). The contents of the adjoining buffer can be flushed/invalidated if it falls in any of the cache line.

Associated data types

qurt_addr_t
qurt_size_t
qurt_mem_cache_op_t
qurt_mem_cache_type_t

Parameters

in	*addr*	Address of data to flush.
in	*size*	Size (in bytes) of data to flush.
in	*opcode*	Type of cache clean operation. Values: QURT_MEM_CACHE_FLUSH QURT_MEM_CACHE_INVALIDATE QURT_MEM_CACHE_FLUSH_INVALIDATE
in	*type*	Cache type. Values: QURT_MEM_DCACHE

Returns

QURT_EOK – Cache operation performed successfully.
QURT_EVAL – Invalid cache type.

Dependencies

None.

21.11 qurt_mem_cache_phys_clean()

21.11.1 Function Documentation

21.11.1.1 int qurt_mem_cache_phys_clean (unsigned int *mask*, unsigned int *addrmatch*, qurt_mem_cache_op_t *opcode*)

Performs a cache clean operation on the data stored in the specified memory area based on address match and mask. Operate on a cache line when (LINE.PhysicalPageNumber & mask) == addrmatch.

NOTE The addrmatch value should be the upper 24-bit physical address to match against.

Associated data types

qurt_mem_cache_op_t

Parameters

in	*mask*	24-bit address mask.
in	*addrmatch*	Physical page number (24 bits) of memory to use as an address match.
in	*opcode*	Type of cache clean operation. Values: • QURT_MEM_CACHE_FLUSH • QURT_MEM_CACHE_INVALIDATE

Returns

QURT_EOK – Cache operation performed successfully.
QURT_EVAL – Invalid operation

Dependencies

None.

21.12 qurt_mem_configure_cache_partition()

21.12.1 Function Documentation

21.12.1.1 int qurt_mem_configure_cache_partition (qurt_cache_type_t *cache_type*, qurt_cache_partition_size_t *partition_size*)

Configures the Hexagon cache partition at the system level.

A partition size value of SEVEN_EIGHTHS_SIZE is applicable only to the L2 cache.

The L1 cache partition is not supported in Hexagon processor version V60 or greater.

NOTE Call this operation only with QuRT OS privilege.

Associated data types

qurt_cache_type_t
qurt_cache_partition_size_t

Parameters

in	*cache_type*	Cache type for partition configuration. Values: • HEXAGON_L1_I_CACHE • HEXAGON_L1_D_CACHE • HEXAGON_L2_CACHE
in	*partition_size*	Cache partition size. Values: • FULL_SIZE • HALF_SIZE • THREE_QUARTER_SIZE • SEVEN_EIGHTHS_SIZE

Returns

QURT_EOK – Success.
QURT_EVAL – Error.

Dependencies

None.

21.13 qurt_mem_l2cache_line_lock()

21.13.1 Function Documentation

21.13.1.1 int qurt_mem_l2cache_line_lock (qurt_addr_t *addr,* qurt_size_t *size*)

Performs an L2 cache line locking operation. This function locks selective lines in the L2 cache memory.

NOTE Perform the line lock operation only on the 32-byte aligned size and address.

Associated data types

qurt_addr_t
qurt_size_t

Parameters

in	*addr*	Address of the L2 cache memory line to lock; the address must be 32-byte aligned.
in	*size*	Size (in bytes) of L2 cache memory to line lock; size must be a multiple of 32 bytes.

Returns

QURT_EOK – Success.
QURT_EALIGN – Data alignment or address failure.

Dependencies

None.

21.14 qurt_mem_l2cache_line_unlock()

21.14.1 Function Documentation

21.14.1.1 int qurt_mem_l2cache_line_unlock (qurt_addr_t *addr,* qurt_size_t *size*)

Performs an L2 cache line unlocking operation. This function unlocks selective lines in the L2 cache memory.

NOTE Perform the line unlock operation only on a 32-byte aligned size and address.

Associated data types

qurt_addr_t
qurt_size_t

Parameters

in	*addr*	Address of the L2 cache memory line to unlock; the address must be 32-byte aligned.
in	*size*	Size (in bytes) of the L2 cache memory line to unlock; size must be a multiple of 32 bytes.

Returns

QURT_EOK – Success.
QURT_EALIGN – Aligning data or address failure.
QURT_EFAILED – Operation failed, cannot find the matching tag.

Dependencies

None.

21.15 qurt_mem_map_static_query()

21.15.1 Function Documentation

21.15.1.1 int qurt_mem_map_static_query (qurt_addr_t * *vaddr,* qurt_addr_t *paddr,* unsigned int *page_size,* qurt_mem_cache_mode_t *cache_attribs,* qurt_perm_t *perm*)

Determines whether a memory page is statically mapped. Pages are specified by the following attributes: physical address, page size, cache mode, and memory permissions.

- If the specified page is statically mapped, vaddr returns the virtual address of the page.
- If the page is not statically mapped (or if it does not exist as specified), vaddr returns -1 as the virtual address value.

  The system configuration file defines QuRT memory maps.

Associated data types

qurt_addr_t
qurt_mem_cache_mode_t
qurt_perm_t

Parameters

out	*vaddr*	Virtual address corresponding to paddr.
in	*paddr*	Physical address.
in	*page_size*	Size of the mapped memory page.
in	*cache_attribs*	Cache mode (writeback, and so on).
in	*perm*	Access permissions.

Returns

QURT_EOK – Specified page is statically mapped, vaddr returns the virtual address.
QURT_EMEM – Specified page is not statically mapped, vaddr returns -1.
QURT_EVAL – Specified page does not exist.

Dependencies

None.

21.16 qurt_mem_map_static_query_64()

21.16.1 Function Documentation

21.16.1.1 int qurt_mem_map_static_query_64 (qurt_addr_t * *vaddr,* qurt_paddr_64_t *paddr_64,* unsigned int *page_size,* qurt_mem_cache_mode_t *cache_attribs,* qurt_perm_t *perm*)

Determines if a memory page is statically mapped. The following attributes specify pages: 64-bit physical address, page size, cache mode, and memory permissions.

If the specified page is statically mapped, vaddr returns the virtual address of the page. If the page is not statically mapped (or if it does not exist as specified), vaddr returns -1 as the virtual address value.

QuRT memory maps are defined in the system configuration file.

Associated data types

qurt_addr_t
qurt_paddr_64_t
qurt_mem_cache_mode_t
qurt_perm_t

Parameters

out	*vaddr*	Virtual address corresponding to paddr.
in	*paddr_64*	64-bit physical address.
in	*page_size*	Size of the mapped memory page.
in	*cache_attribs*	Cache mode (writeback, and so on).
in	*perm*	Access permissions.

Returns

QURT_EOK – Specified page is statically mapped; a virtual address is returned in vaddr.
QURT_EMEM – Specified page is not statically mapped; -1 is returned in vaddr.
QURT_EVAL – Specified page does not exist.

Dependencies

None.

21.17 qurt_mem_pool_add_pages()

21.17.1 Function Documentation

21.17.1.1 int qurt_mem_pool_add_pages (qurt_mem_pool_t *pool*, unsigned *first_pageno*, unsigned *size_in_pages*)

Adds a physical address range to the specified memory pool object.

NOTE Call this operation only with root privileges (guest OS mode).

Associated data types

qurt_mem_pool_t

Parameters

in	*pool*	Memory pool object.
in	*first_pageno*	First page number of the physical address range (equivalent to address >> 12)
in	*size_in_pages*	Number of pages in the physical address range (equivalent to size >> 12)

Returns

QURT_EOK – Pages successfully added.

Dependencies

None.

21.18 qurt_mem_pool_attach()

21.18.1 Function Documentation

21.18.1.1 int qurt_mem_pool_attach (char * *name*, qurt_mem_pool_t * *pool*)

Initializes a memory pool object to attach to a pool predefined in the system configuration file.

Memory pool objects assign memory regions to physical memory in different Hexagon memory units. They are specified in memory region create operations (Section 21.43.1.1).

NOTE QuRT predefines the memory pool object qurt_mem_default_pool (Section 21) for allocation memory regions in SMI memory. The pool attach operation is necessary only when allocating memory regions in nonstandard memory units such as TCM.

Associated data types

qurt_mem_pool_t

Parameters

in	*name*	Pointer to the memory pool name.
out	*pool*	Pointer to the memory pool object.

Returns

QURT_EOK – Attach operation successful.

Dependencies

None.

21.19 qurt_mem_pool_attach2()

21.19.1 Function Documentation

21.19.1.1 int qurt_mem_pool_attach2 (int *client_handle,* char * *name,* qurt_mem_pool_t * *pool*)

Gets the identifier that corresponds to a pool object created specifically for a client, for example, HLOS_PHYSPOOL. The client_handle is used to look up the client specific pool.

Memory pool objects assign memory regions to physical memory in different Hexagon memory units. Memory pool objects are specified during map creation operations (qurt_mem_mmap() and qurt_mem_region_create()).

NOTE QuRT predefines the memory pool object qurt_mem_default_pool (Section 21) for allocation memory regions in SMI memory. The pool_attach2 operation is necessary only when allocating memory regions in memory units specific to the client.

Associated data types

qurt_mem_pool_t

Parameters

in	*client_handle*	Client identifier used by the OS to lookup the identifier for client specific pool
in	*name*	Pointer to the memory pool name.
out	*pool*	Pointer to the memory pool object.

Returns

QURT_EOK – Attach operation successful.

Dependencies

None.

21.20 qurt_mem_pool_attr_get()

21.20.1 Function Documentation

21.20.1.1 int qurt_mem_pool_attr_get (qurt_mem_pool_t *pool,* qurt_mem_pool_attr_t ∗ *attr*)

Gets the memory pool attributes.

Retrieves pool configurations based on the pool handle, and fills in the attribute structure with configuration values.

Associated data types

qurt_mem_pool_t
qurt_mem_pool_attr_t

Parameters

in	*pool*	Pool handle obtained from qurt_mem_pool_attach().
out	*attr*	Pointer to the memory region attribute structure.

Returns

0 – Success.
QURT_EINVALID – Corrupt handle; pool handle is invalid.

21.21 qurt_mem_pool_attr_get_addr()

21.21.1 Function Documentation

21.21.1.1 static int qurt_mem_pool_attr_get_addr (qurt_mem_pool_attr_t * *attr,* int *range_id,* qurt_addr_t * *addr*)

Gets the start address of the specified memory pool range.

Associated data types

qurt_mem_pool_attr_t
qurt_addr_t

Parameters

in	*attr*	Pointer to the memory pool attribute structure.
in	*range_id*	Memory pool range key.
out	*addr*	Pointer to the destination variable for range start address.

Returns

0 – Success.
QURT_EINVALID – Range is invalid.

Dependencies

None.

21.22 qurt_mem_pool_is_available()

21.22.1 Function Documentation

21.22.1.1 int qurt_mem_pool_is_available (qurt_mem_pool_t *pool*, int *page_count*, qurt_mem_mapping_t *mapping_type*)

Checks whether the number of pages that the page_count argument indicates can be allocated from the specified pool.

Associated data types

qurt_mem_pool_attr_t
qurt_mem_mapping_t

Parameters

in	*pool*	Pool handle obtained from qurt_mem_pool_attach().
in	*page_count*	Number of 4K pages.
in	*mapping_type*	Variable of type qurt_mem_mapping_t.

Returns

0 – Success.
QURT_EINVALID – Mapping_type is invalid.
QURT_EMEM – Specified pages cannot be allocated from the pool.

Dependencies

None.

21.23 qurt_mem_pool_attr_get_size()

21.23.1 Function Documentation

21.23.1.1 static int qurt_mem_pool_attr_get_size (qurt_mem_pool_attr_t * *attr,* int *range_id,* qurt_size_t * *size*)

Gets the size of the specified memory pool range.

Associated data types

qurt_mem_pool_attr_t
qurt_size_t

Parameters

in	*attr*	Pointer to the memory pool attribute structure.
in	*range_id*	Memory pool range key.
out	*size*	Pointer to the destination variable for the range size.

Returns

0 – Success.
QURT_EINVALID – Range is invalid.

Dependencies

None.

21.24 qurt_mem_pool_create()

21.24.1 Function Documentation

21.24.1.1 int qurt_mem_pool_create (char * *name*, unsigned *base*, unsigned *size*, qurt_mem_pool_t * *pool*)

Dynamically creates a memory pool object from a physical address range.

The pool is assigned a single memory region with the specified base address and size.

The base address and size values passed to this function must be aligned to 4K byte boundaries, and must be expressed as the actual base address and size values divided by 4K.

For example, the function call:

```
qurt_mem_pool_create ("TCM_PHYSPOOL", 0xd8020, 0x20, &pool)
```

... is equivalent to the following static pool definition in the QuRT system configuration file:

```
<physical_pool name="TCM_PHYSPOOL">
    <region base="0xd8020000" size="0x20000" />
</physical_pool>
```

NOTE Dynamically created pools are not identical to static pools. In particular, qurt_mem_pool_attr_get() is not valid with dynamically created pools.

Dynamic pool creation permanently consumes system resources, and cannot be undone.

Associated data types

qurt_mem_pool_t

Parameters

in	*name*	Pointer to the memory pool name.
in	*base*	Base address of the memory region (divided by 4K).
in	*size*	Size (in bytes) of the memory region (divided by 4K).
out	*pool*	Pointer to the memory pool object.

Returns

QURT_EOK – Success.

Dependencies

None.

21.25 qurt_mem_pool_remove_pages()

21.25.1 Function Documentation

21.25.1.1 int qurt_mem_pool_remove_pages (qurt_mem_pool_t *pool*, unsigned *first_pageno*, unsigned *size_in_pages*, unsigned *flags*, void(∗)(void ∗) *callback*, void ∗ *arg*)

Removes a physical address range from the specified memory pool object.

If any part of the address range is in use, this operation returns an error without changing the state.

NOTE Call this operation only with root privileges (guest-OS mode).

In the future, this operation will support (via the flags parameter) the removal of a physical address range when part of the range is in use.

Associated data types

qurt_mem_pool_t

Parameters

in	*pool*	Memory pool object.
in	*first_pageno*	First page number of the physical address range (equivalent to address >> 12)
in	*size_in_pages*	Number of pages in the physical address range (equivalent to size >> 12)
in	*flags*	Remove options. Values: • 0 – Skip holes in the range that are not part of the pool (default) • QURT_POOL_REMOVE_ALL_OR_NONE – Pages are removed only if the specified physical address range is entirely contained (with no holes) in the pool free space.
in	*callback*	Callback procedure called when pages were successfully removed. Not called if the operation failed. Passing 0 as the parameter value causes the callback to not be called.
in	*arg*	Value passed as an argument to the callback procedure.

Returns

QURT_EOK – Pages successfully removed.

Dependencies

None.

21.26 qurt_mem_region_attr_get()

21.26.1 Function Documentation

21.26.1.1 int qurt_mem_region_attr_get (qurt_mem_region_t *region,* qurt_mem_region_attr_t * *attr*)

Gets the memory attributes of the specified message region. After a memory region is created, its attributes cannot be changed.

Associated data types

qurt_mem_region_t
qurt_mem_region_attr_t

Parameters

in	*region*	Memory region object.
out	*attr*	Pointer to the destination structure for memory region attributes.

Returns

QURT_EOK – Operation successfully performed.
Error code – Failure.

Dependencies

None.

21.27 qurt_mem_region_attr_get_bus_attr()

21.27.1 Function Documentation

21.27.1.1 static void qurt_mem_region_attr_get_bus_attr (qurt_mem_region_attr_t * *attr,* unsigned * *pbits*)

Gets the (A1, A0) bus attribute bits from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*pbits*	Pointer to an unsigned integer that is filled in with the (A1, A0) bits from the memory region attribute structure, expressed as a 2-bit binary number.

Returns

None.

Dependencies

None.

21.28 qurt_mem_region_attr_get_cache_mode()

21.28.1 Function Documentation

21.28.1.1 static void qurt_mem_region_attr_get_cache_mode (qurt_mem_region_attr_t * *attr*, qurt_mem_cache_mode_t * *mode*)

Gets the cache operation mode from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_mem_cache_mode_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*mode*	Pointer to the destination variable for cache mode.

Returns

None.

Dependencies

None.

21.29 qurt_mem_region_attr_get_mapping()

21.29.1 Function Documentation

21.29.1.1 static void qurt_mem_region_attr_get_mapping (qurt_mem_region_attr_t * *attr,* qurt_mem_mapping_t * *mapping*)

Gets the memory mapping from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_mem_mapping_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*mapping*	Pointer to the destination variable for memory mapping.

Returns

None.

Dependencies

None.

21.30 qurt_mem_region_attr_get_physaddr()

21.30.1 Function Documentation

21.30.1.1 static void qurt_mem_region_attr_get_physaddr (qurt_mem_region_attr_t * *attr,* unsigned int * *addr*)

Gets the memory region physical address from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*addr*	Pointer to the destination variable for memory region physical address.

Returns

None.

Dependencies

None.

21.31 qurt_mem_region_attr_get_size()

21.31.1 Function Documentation

21.31.1.1 static void qurt_mem_region_attr_get_size (qurt_mem_region_attr_t * *attr*, qurt_size_t * *size*)

Gets the memory region size from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_size_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*size*	Pointer to the destination variable for memory region size.

Returns

None.

Dependencies

None.

21.32 qurt_mem_region_attr_get_type()

21.32.1 Function Documentation

21.32.1.1 static void qurt_mem_region_attr_get_type (qurt_mem_region_attr_t * *attr,* qurt_mem_region_type_t * *type*)

Gets the memory type from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_mem_region_type_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*type*	Pointer to the destination variable for the memory type.

Returns

None.

Dependencies

None.

21.33 qurt_mem_region_attr_get_virtaddr()

21.33.1 Function Documentation

21.33.1.1 static void qurt_mem_region_attr_get_virtaddr (qurt_mem_region_attr_t * *attr,* unsigned int * *addr*)

Gets the memory region virtual address from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*addr*	Pointer to the destination variable for the memory region virtual address.

Returns

None.

Dependencies

None.

21.34 qurt_mem_region_attr_get_physaddr_64()

21.34.1 Function Documentation

21.34.1.1 static void qurt_mem_region_attr_get_physaddr_64 (qurt_mem_region_attr_t * *attr*, qurt_paddr_64_t * *addr_64*)

Gets the memory region 64-bit physical address from the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_paddr_64_t

Parameters

in	*attr*	Pointer to the memory region attribute structure.
out	*addr_64*	Pointer to the destination variable for the memory region 64-bit physical address.

Returns

None.

Dependencies

None.

21.35 qurt_mem_region_attr_init()

21.35.1 Function Documentation

21.35.1.1 void qurt_mem_region_attr_init (qurt_mem_region_attr_t * *attr*)

Initializes the specified memory region attribute structure with default attribute values:

- Mapping – QURT_MEM_MAPPING_VIRTUAL
- Cache mode – QURT_MEM_CACHE_WRITEBACK
- Physical address – -1
- Virtual address – -1
- Memory type – QURT_MEM_REGION_LOCAL
- Size – -1

NOTE The memory physical address attribute must be explicitly set by calling the qurt_mem_region_attr_set_physaddr() function. The size and pool attributes are set directly as parameters in the memory region create operation.

Associated data types

qurt_mem_region_attr_t

Parameters

`in,out`	*attr*	Pointer to the destination structure for the memory region attributes.

Returns

None.

Dependencies

None.

21.36 qurt_mem_region_attr_set_bus_attr()

21.36.1 Function Documentation

21.36.1.1 static void qurt_mem_region_attr_set_bus_attr (qurt_mem_region_attr_t * *attr,* unsigned *abits*)

Sets the (A1, A0) bus attribute bits in the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*abits*	The (A1, A0) bits to use with the memory region, expressed as a 2-bit binary number.

Returns

None.

Dependencies

None.

21.37 qurt_mem_region_attr_set_cache_mode()

21.37.1 Function Documentation

21.37.1.1 static void qurt_mem_region_attr_set_cache_mode (qurt_mem_region_attr_t * *attr*, qurt_mem_cache_mode_t *mode*)

Sets the cache operation mode in the specified memory region attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_mem_cache_mode_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*mode*	Cache mode. Values: • QURT_MEM_CACHE_WRITEBACK • QURT_MEM_CACHE_WRITETHROUGH • QURT_MEM_CACHE_WRITEBACK_NONL2CACHEABLE • QURT_MEM_CACHE_WRITETHROUGH_NONL2CACHEABLE • QURT_MEM_CACHE_WRITEBACK_L2CACHEABLE • QURT_MEM_CACHE_WRITETHROUGH_L2CACHEABLE • QURT_MEM_CACHE_NONE

Returns

None.

Dependencies

None.

21.38 qurt_mem_region_attr_set_mapping()

21.38.1 Function Documentation

21.38.1.1 static void qurt_mem_region_attr_set_mapping (qurt_mem_region_attr_t * *attr,* qurt_mem_mapping_t *mapping*)

Sets the memory mapping in the specified memory region attribute structure.

The mapping value indicates how the memory region is mapped in virtual memory.

Associated data types

qurt_mem_region_attr_t
qurt_mem_mapping_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*mapping*	Mapping. Values: • QURT_MEM_MAPPING_VIRTUAL • QURT_MEM_MAPPING_PHYS_CONTIGUOUS • QURT_MEM_MAPPING_IDEMPOTENT • QURT_MEM_MAPPING_VIRTUAL_FIXED • QURT_MEM_MAPPING_NONE • QURT_MEM_MAPPING_VIRTUAL_RANDOM • QURT_MEM_MAPPING_INVALID

Returns

None.

Dependencies

None.

21.39 qurt_mem_region_attr_set_physaddr()

21.39.1 Function Documentation

21.39.1.1 static void qurt_mem_region_attr_set_physaddr (qurt_mem_region_attr_t * *attr,* qurt_paddr_t *addr*)

Sets the memory region 32-bit physical address in the specified memory attribute structure.

NOTE The physical address attribute is explicitly set only for memory regions with physical contiguous mapping. Otherwise QuRT automatically sets it when the memory region is created.

Associated data types

qurt_mem_region_attr_t
qurt_paddr_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*addr*	Memory region physical address.

Returns

None.

21.40 qurt_mem_region_attr_set_physaddr_64()

21.40.1 Function Documentation

21.40.1.1 static void qurt_mem_region_attr_set_physaddr_64 (qurt_mem_region_attr_t * *attr,* qurt_paddr_64_t *addr_64*)

Sets the memory region 64-bit physical address in the specified memory attribute structure.

NOTE The physical address attribute is explicitly set only for memory regions with physical contiguous mapping. Otherwise it is automatically set by QuRT when the memory region is created.

Associated data types

qurt_mem_region_attr_t
qurt_paddr_64_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*addr_64*	Memory region 64-bit physical address.

Returns

None.

21.41 qurt_mem_region_attr_set_type()

21.41.1 Function Documentation

21.41.1.1 static void qurt_mem_region_attr_set_type (qurt_mem_region_attr_t * *attr,* qurt_mem_region_type_t *type*)

Sets the memory type in the specified memory region attribute structure.

The type indicates whether the memory region is local to an application or shared between applications.

Associated data types

qurt_mem_region_attr_t
qurt_mem_region_type_t

Parameters

in,out	*attr*	Pointer to memory region attribute structure.
in	*type*	Memory type. Values: • QURT_MEM_REGION_LOCAL • QURT_MEM_REGION_SHARED

Returns

None.

Dependencies

None.

21.42 qurt_mem_region_attr_set_virtaddr()

21.42.1 Function Documentation

21.42.1.1 static void qurt_mem_region_attr_set_virtaddr (qurt_mem_region_attr_t * *attr,* qurt_addr_t *addr*)

Sets the memory region virtual address in the specified memory attribute structure.

Associated data types

qurt_mem_region_attr_t
qurt_addr_t

Parameters

in,out	*attr*	Pointer to the memory region attribute structure.
in	*addr*	Memory region virtual address.

Returns

None.

Dependencies

None.

21.43 qurt_mem_region_create()

21.43.1 Function Documentation

21.43.1.1 int qurt_mem_region_create (qurt_mem_region_t * *region*, qurt_size_t *size*, qurt_mem_pool_t *pool*, qurt_mem_region_attr_t * *attr*)

Creates a memory region with the specified attributes.

The application initializes the memory region attribute structure with qurt_mem_region_attr_init() and qurt_mem_region_attr_set_bus_attr().

If the virtual address attribute is set to its default value (Section 21.35.1.1), the virtual address of the memory region is automatically assigned any available virtual address value.

If the memory mapping attribute is set to virtual mapping, the physical address of the memory region is also automatically assigned.

NOTE The physical address attribute is explicitly set in the attribute structure only for memory regions with physical-contiguous-mapped mapping.

Memory regions are always assigned to memory pools. The pool value specifies the memory pool that the memory region is assigned to.

NOTE If attr is specified as NULL, the memory region is created with default attribute values (Section 21.35.1.1). QuRT predefines the memory pool object qurt_mem_default_pool (Section 21), which allocates memory regions in SMI memory.

Associated data types

qurt_mem_region_t
qurt_size_t
qurt_mem_pool_t
qurt_mem_region_attr_t

Parameters

out	*region*	Pointer to the memory region object.
in	*size*	Memory region size (in bytes). If size is not an integral multiple of 4K, it is rounded up to a 4K boundary.
in	*pool*	Memory pool of the region.
in	*attr*	Pointer to the memory region attribute structure.

Returns

QURT_EOK – Memory region successfully created.
QURT_EMEM – Not enough memory to create region.

Dependencies

None.

21.44 qurt_mem_region_delete()

21.44.1 Function Documentation

21.44.1.1 int qurt_mem_region_delete (qurt_mem_region_t *region*)

Deletes the specified memory region.

If the caller application creates the memory region, it is removed and the system reclaims its assigned memory.

If a different application creates the memory region (and is shared with the caller application), only the local memory mapping to the region is removed; the system does not reclaim the memory.

Associated data types

qurt_mem_region_t

Parameters

in	*region*	Memory region object.

Returns

QURT_EOK – Region successfully deleted.

Dependencies

None.

21.45 qurt_mem_region_query()

21.45.1 Function Documentation

21.45.1.1 int qurt_mem_region_query (qurt_mem_region_t * *region_handle,* qurt_addr_t *vaddr,* qurt_paddr_t *paddr*)

Queries a memory region.

This function determines whether a dynamically-created memory region (Section 21.43.1.1) exists for the specified virtual or physical address. When a memory region has been determined to exist, its attributes are accessible (Section 21.26.1.1).

NOTE This function returns QURT_EFATAL if QURT_EINVALID is passed to both vaddr and paddr (or to neither).

Associated data types

qurt_mem_region_t
qurt_paddr_t

Parameters

out	*region_handle*	Pointer to the memory region object (if it exists).
in	*vaddr*	Virtual address to query; if vaddr is specified, paddr must be set to the value QURT_EINVALID.
in	*paddr*	Physical address to query; if paddr is specified, vaddr must be set to the value QURT_EINVALID.

Returns

QURT_EOK – Query successfully performed.
QURT_EMEM – Region not found for the specified address.
QURT_EFATAL – Invalid input parameters.

Dependencies

None.

21.46 qurt_mem_region_query_64()

21.46.1 Function Documentation

21.46.1.1 int qurt_mem_region_query_64 (qurt_mem_region_t * *region_handle,* qurt_addr_t *vaddr,* qurt_paddr_64_t *paddr_64*)

Determines whether a dynamically created memory region (Section 21.43.1.1) exists for the specified virtual or physical address. When a memory region has been determined to exist, its attributes are accessible (Section 21.26.1.1).

NOTE This function returns QURT_EFATAL if QURT_EINVALID is passed to both vaddr and paddr (or to neither).

Associated data types

qurt_mem_region_t
qurt_addr_t
qurt_paddr_64_t

Parameters

out	*region_handle*	Pointer to the memory region object (if it exists).
in	*vaddr*	Virtual address to query; if vaddr is specified, paddr must be set to the value QURT_EINVALID.
in	*paddr_64*	64-bit physical address to query; if paddr is specified, vaddr must be set to the value QURT_EINVALID.

Returns

QURT_EOK – Success.
QURT_EMEM – Region not found for the specified address.
QURT_EFATAL – Invalid input parameters.

Dependencies

None.

21.47 qurt_mem_syncht()

21.47.1 Function Documentation

21.47.1.1 static void qurt_mem_syncht (void)

Performs heavy-weight synchronization of memory transactions.

This operation does not return until all previous memory transactions (cached and uncached load/store, mem_locked, and so on) that originated from the current thread are complete and globally observable.

NOTE This operation is implemented as a wrapper for the Hexagon syncht instruction.

Returns

None.

Dependencies

None.

21.48 Memory Management Data Types

- Memory pools are represented in QuRT as objects of type qurt_mem_pool_t.
- Memory regions are represented as objects of type qurt_mem_region_t.
- Memory region attributes are stored in structures of type qurt_mem_region_attr_t.
- Memory region types are stored as values of type qurt_mem_region_type_t.
- Memory region mappings are specified as values of type qurt_mem_mapping_t.
- Cache types are specified as values of type qurt_mem_cache_type_t.
- Cache modes are specified as values of type qurt_mem_cache_mode_t.
- Cache operation codes are specified as values of type qurt_mem_cache_op_t.
- QuRT pre-initializes the memory pool object qurt_mem_default_pool.

21.48.1 Define Documentation

21.48.1.1 #define QURT_POOL_REMOVE_ALL_OR_NONE 1

21.48.2 Data Structure Documentation

21.48.2.1 struct qurt_mem_region_attr_t

QuRT memory region attributes type.

21.48.2.2 struct qurt_mem_pool_attr_t

QuRT user physical memory pool type.

21.48.3 Typedef Documentation

21.48.3.1 typedef unsigned int qurt_addr_t

QuRT address type.

21.48.3.2 typedef unsigned int qurt_paddr_t

QuRT physical memory address type.

21.48.3.3 typedef unsigned long long qurt_paddr_64_t

QuRT 64-bit physical memory address type.

21.48.3.4 typedef unsigned int qurt_mem_region_t

QuRT memory regions type.

21.48.3.5 typedef unsigned int qurt_mem_fs_region_t

QuRT memory FS region type.

21.48.3.6 typedef unsigned int qurt_mem_pool_t

QuRT memory pool type.

21.48.3.7 typedef unsigned int qurt_size_t

QuRT size type.

21.48.4 Enumeration Type Documentation

21.48.4.1 enum qurt_mem_mapping_t

QuRT memory region mapping type.

Enumerator:

QURT_MEM_MAPPING_VIRTUAL Default mode. The region virtual address range maps to an available contiguous area of physical memory. For the most efficient use of virtual memory, the QuRT system chooses the base address in physical memory. This works for most memory use cases.

QURT_MEM_MAPPING_PHYS_CONTIGUOUS When external devices that bypass Hexagon virtual memory addressing access the memory region, the region virtual address space must map to a contiguous area of physical memory. The base address in physical memory must be explicitly specified.

QURT_MEM_MAPPING_IDEMPOTENT Region virtual address space maps to the identical area of physical memory.

QURT_MEM_MAPPING_VIRTUAL_FIXED Virtual address space of the region maps either to the specified area of physical memory or (if no area is specified) to available physical memory. Use this mapping to create regions from virtual space that was reserved by calling qurt_mem_region_create() with mapping.

QURT_MEM_MAPPING_NONE Reserves a virtual memory area (VMA). Remapping a virtual range is not permitted without first deleting the memory region. When such a region is deleted, its corresponding virtual memory addressing remains intact.

QURT_MEM_MAPPING_VIRTUAL_RANDOM System chooses a random virtual address and maps it to available contiguous physical addresses.

QURT_MEM_MAPPING_PHYS_DISCONTIGUOUS While virtual memory is contiguous, allocates in discontiguous physical memory blocks. This helps when there are smaller contiguous blocks than the requested size. Physical address is not provided as part of the get_attr call

QURT_MEM_MAPPING_INVALID Reserved as an invalid mapping type.

21.48.4.2 enum qurt_mem_cache_mode_t

QuRT cache mode type.

Enumerator:

QURT_MEM_CACHE_WRITEBACK Write back.

QURT_MEM_CACHE_NONE_SHARED Normal uncached memory that can be shared with other subsystems.
QURT_MEM_CACHE_WRITETHROUGH Write through.
QURT_MEM_CACHE_WRITEBACK_NONL2CACHEABLE Write back non-L2-cacheable.
QURT_MEM_CACHE_WRITETHROUGH_NONL2CACHEABLE Write through non-L2-cacheable.
QURT_MEM_CACHE_WRITEBACK_L2CACHEABLE Write back L2 cacheable.
QURT_MEM_CACHE_WRITETHROUGH_L2CACHEABLE Write through L2 cacheable.
QURT_MEM_CACHE_DEVICE Volatile memory-mapped device. Access to device memory cannot be cancelled by interrupts, re-ordered, or replayed.
QURT_MEM_CACHE_NONE Deprecated – use QURT_MEM_CACHE_DEVICE instead.
QURT_MEM_CACHE_DEVICE_SFC Enables placing limitations on the number of outstanding transactions.
QURT_MEM_CACHE_INVALID Reserved as an invalid cache type.

21.48.4.3 enum qurt_perm_t

Memory access permission.

Enumerator:

QURT_PERM_NONE No permission.
QURT_PERM_READ Read permission.
QURT_PERM_WRITE Write permission.
QURT_PERM_EXECUTE Execution permission.
QURT_PERM_NODUMP Skip dumping the mapping. During process domain dump, must skip some mappings on host memory to avoid a race condition where the memory is removed from the host and DSP process crashed before the mapping is removed.
QURT_PERM_FULL Read, write, and execute permission.

21.48.4.4 enum qurt_mem_cache_type_t

QuRT cache type; specifies data cache or instruction cache.

Enumerator:

QURT_MEM_ICACHE Instruction cache.
QURT_MEM_DCACHE Data cache.

21.48.4.5 enum qurt_mem_cache_op_t

QuRT cache operation code type.

Enumerator:

QURT_MEM_CACHE_FLUSH Flush.
QURT_MEM_CACHE_INVALIDATE Invalidate
QURT_MEM_CACHE_FLUSH_INVALIDATE Flush invalidate.
QURT_MEM_CACHE_FLUSH_ALL Flush all.
QURT_MEM_CACHE_FLUSH_INVALIDATE_ALL Flush invalidate all.
QURT_MEM_CACHE_TABLE_FLUSH_INVALIDATE Table flush invalidate.
QURT_MEM_CACHE_FLUSH_INVALIDATE_L2 L2 flush invalidate.

21.48.4.6 enum qurt_mem_region_type_t

QuRT memory region type.

Enumerator:

QURT_MEM_REGION_LOCAL Local.
QURT_MEM_REGION_SHARED Shared.
QURT_MEM_REGION_USER_ACCESS User access.
QURT_MEM_REGION_FS FS.
QURT_MEM_REGION_INVALID Reserved as an invalid region type.

21.48.4.7 enum qurt_cache_type_t

Enumerator:

HEXAGON_L1_I_CACHE Hexagon L1 instruction cache.
HEXAGON_L1_D_CACHE Hexagon L1 data cache.
HEXAGON_L2_CACHE Hexagon L2 cache.

21.48.4.8 enum qurt_cache_partition_size_t

Enumerator:

FULL_SIZE Fully shared cache, without partitioning.
HALF_SIZE 1/2 for main, 1/2 for auxiliary.
THREE_QUARTER_SIZE 3/4 for main, 1/4 for auxiliary.
SEVEN_EIGHTHS_SIZE 7/8 for main, 1/8 for auxiliary; for L2 cache only.

21.48.5 Variable Documentation

21.48.5.1 qurt_mem_pool_t qurt_mem_default_pool

Memory pool object.

21.49 Memory Management Macros

21.49.1 Define Documentation

21.49.1.1 #define QURT_SYSTEM_ALLOC_VIRTUAL 1

Allocates available virtual memory in the address space of all processes.

22 Memory Mapping

Memory mapping services are accessed with the following QuRT functions.

- qurt_mem_mmap()
- qurt_mem_mmap2()
- qurt_mem_mmap_by_name()
- qurt_mem_mprotect2()
- qurt_mem_mprotect()
- qurt_mem_munmap()
- qurt_mem_munmap2()
- qurt_mem_munmap3()
- Memory Mapping Macros

22.1 qurt_mem_mmap()

22.1.1 Function Documentation

22.1.1.1 void* qurt_mem_mmap (int *client_handle,* qurt_mem_pool_t *pool,* qurt_mem_region_t * *pRegion,* void * *addr,* size_t *length,* int *prot,* int *flags,* int *fd,* unsigned long long *offset*)

Creates a memory mapping with the specified attributes. This API allows the root process caller to create mapping on behalf of a user process. If the client_handle belongs to a valid user process, the resulting mapping is created for the process. If -1 is passed in place of client_handle, the API creates mapping for the underlying process of the caller.

NOTE If the specified attributes are not valid, an error result is returned.

Parameters

out	*client_handle*	Client handle to use for this mapping (optional).
in	*pool*	Optional argument that specifies a pool handle if the user wants to allocate memory from a specific pool. The default value for this argument is NULL.
in	*pRegion*	Map region. This argument is unused, and the default value is NULL.
in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.
in	*prot*	Mapping access permissions (R/W/X).
in	*flags*	Mapping modes. • QURT_MAP_NAMED_MEMSECTION • QURT_MAP_FIXED • QURT_MAP_NONPROCESS_VPOOL • QURT_MAP_TRYFIXED • QURT_MAP_ANON • QURT_MAP_PHYSADDR • QURT_MAP_VA_ONLY
in	*fd*	File designator.
in	*offset*	Offset in file.

Returns

Valid virtual address – Success.
QURT_MAP_FAILED – Mapping creation failed.

22.2 qurt_mem_mmap2()

22.2.1 Function Documentation

22.2.1.1 void* qurt_mem_mmap2 (int *client_handle,* qurt_mem_pool_t *pool,* qurt_mem_region_t * *pRegion,* void * *addr,* size_t *length,* int *prot,* int *flags,* int *fd,* unsigned long long *offset*)

Creates a memory mapping with the specified attributes. Returns a more descriptive error code in case of failure. This API allows the root process caller to create mapping on behalf of a user process. If the client_handle belongs to a valid user process, the resulting mapping is created for the process. If -1 is passed in place of client_handle, the API creates mapping for the underlying process of the caller.

NOTE If the specified attributes are not valid, an error result is returned.

Parameters

out	*client_handle*	Client handle to use for this mapping (optional).
in	*pool*	Optional argument that allows the user to specify a pool handle when the user wants to allocate memory from a specific pool. Default value for this argument is NULL.
in	*pRegion*	Map region (unused argument); default value is NULL.
in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.
in	*prot*	Mapping access permissions (R/W/X). Cache attributes, bus attributes, User mode.
in	*flags*	Mapping modes; Shared, Private, or Anonymous.
in	*fd*	File designator.
in	*offset*	Offset in file.

Returns

Valid virtual address – Success.
QURT_EMEM – Physical address is not available.
QURT_EFAILED – VA is not available or mapping failed.
QURT_EINVALID – Invalid argument was passed (for example, an unaligned VA/PA).

22.3 qurt_mem_mmap_by_name()

22.3.1 Function Documentation

22.3.1.1 void* qurt_mem_mmap_by_name (const char * *name*, void * *addr*, size_t *length*, int *prot*, int *flags*, unsigned long long *offset*)

Creates a memory mapping for a named-memsection using the specified attributes. The named memsection should be specified in cust_config.xml.

NOTE If the specified attributes are not valid or the named memsection is not found, an error result is returned.

Parameters

in	*name*	Name of the memsection in cust_config.xml that specifies this mapping. Should be less than 25 characters.
in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.
in	*prot*	Mapping access permissions (R/W/X). Cache attributes, bus attributes, User mode
in	*flags*	Mapping modes, such as Shared, Private, or Anonymous.
in	*offset*	Offset relative to the physical address range specified in memsection. If offset + length exceeds size of memsection, failure is returned.

Returns

Valid virtual address – Success.
QURT_MAP_FAILED – Mapping creation failed.

22.4 qurt_mem_mprotect2()

22.4.1 Function Documentation

22.4.1.1 int qurt_mem_mprotect2 (int *client_handle,* const void * *addr,* size_t *length,* int *prot*)

Changes access permissions and attributes on an existing mapping based on the client_handle argument.

NOTE If the specified virtual address is not found or invalid attributes are passed, an error code is returned.

Parameters

in	*client_handle*	Obtained from the current invocation function (Section 3.4.1).
in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.
in	*prot*	Mapping access permissions (R/W/X). Cache attributes, Bus attributes, User mode.

Returns

QURT_EOK – Successfully changes permissions on the mapping.
QURT_EFATAL – Failed to change permissions on the mapping

22.5 qurt_mem_mprotect()

22.5.1 Function Documentation

22.5.1.1 int qurt_mem_mprotect (const void * *addr,* size_t *length,* int *prot*)

Changes access permissions and attributes on an existing mapping.

NOTE If the specified virtual address is not found or invalid attributes are passed, an error code is returned.

Parameters

in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.
in	*prot*	Mapping access permissions (R/W/X). Cache attributes, Bus attributes, User mode.

Returns

QURT_EOK – Successfully changes permissions on the mapping.
QURT_EFATAL – Failed to change permissions on the mapping.

22.6 qurt_mem_munmap()

22.6.1 Function Documentation

22.6.1.1 int qurt_mem_munmap (void * *addr*, size_t *length*)

Removes an existing mapping.

NOTE If the specified mapping is not found in the context of the caller process or invalid attributes are passed, an error code is returned.

Parameters

in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.

Returns

QURT_EOK – Successfully changes permissions on the mapping.
QURT_EFATAL – Failed to change permissions on the mapping.

22.7 qurt_mem_munmap2()

22.7.1 Function Documentation

22.7.1.1 int qurt_mem_munmap2 (int *client_handle,* void * *addr,* size_t *length*)

Removes an existing mapping for a specified process.

NOTE This API allows a root process entity, such as a driver, to remove mapping that was created for a user process. If the specified mapping is not found in the context of client handle or invalid attributes are passed, an error code is returned.

Parameters

out	*client_handle*	Client handle of the user process that owns this mapping.
in	*addr*	Virtual memory address.
in	*length*	Size of mapping in bytes.

Returns

QURT_EOK – Successfully changes permissions on the mapping.
QURT_EFATAL – Failed to change permissions on the mapping.

22.8 qurt_mem_munmap3()

22.8.1 Function Documentation

22.8.1.1 int qurt_mem_munmap3 (int *client_handle,* void * *addr,* size_t *length,* int *flags*)

Removes an existing mapping or reservation for a specified process.

Parameters

in	*client_handle*	Client handle of the user process that owns this mapping.
in	*addr*	Pointer to a virtual memory address.
in	*length*	Size of mapping in bytes.
in	*flags*	Specifies the flag.

Returns

QURT_EOK – Successfully changes permissions on the mapping.
QURT_EFATAL – Failed to change permissions on the mapping.

22.9 Memory Mapping Macros

22.9.1 Define Documentation

22.9.1.1 #define QURT_MAP_NAMED_MEMSECTION 0x0004U

Named memsection.

22.9.1.2 #define QURT_MAP_FIXED 0x0010U

Fixed virtual address.

22.9.1.3 #define QURT_MAP_RENAME 0x0020U

Rename.

22.9.1.4 #define QURT_MAP_NORESERVE 0x0040U

No reserve.

22.9.1.5 #define QURT_MAP_INHERIT 0x0080U

Inherit.

22.9.1.6 #define QURT_MAP_NONPROCESS_VPOOL 0x0100U

Use a virtual address outside of the default range of the processes. This option is only supported in the root process and only when virtual memory split is enabled in the XML. The root process can use this flag to create mapping for a user process, for example, if the virtual address is configured for a 3G/1G split, the root process can use this flag to create mapping in the top 1 GB area for the user process or the lower 3 GB area for the root process. This is useful for shared buffer use cases.

22.9.1.7 #define QURT_MAP_HASSEMAPHORE 0x0200U

Has semaphore.

22.9.1.8 #define QURT_MAP_TRYFIXED 0x0400U

Try to create a mapping for a virtual address that was passed. If the passed virtual address fails, use a random virtual address.

22.9.1.9 #define QURT_MAP_WIRED 0x0800U

Wired.

22.9.1.10 #define QURT_MAP_FILE 0x0000U

File.

22.9.1.11 #define QURT_MAP_ANON 0x1000U

Allocate physical memory from the pool that was passed. By default, memory is allocated from the default physpool.

22.9.1.12 #define QURT_MAP_VA_ONLY 0X2000U

Reserve a virtual address without mapping it.

22.9.1.13 #define QURT_MAP_FAILED ((void ∗) -1)

Mapping creation failed.

22.9.1.14 #define QURT_PROT_CACHE_MODE(*n*) QURT_MMAP_BUILD(QURT_PROT_CACHE_BOUNDS,(n))

22.9.1.15 #define QURT_PROT_BUS_ATTR(*n*) QURT_MMAP_BUILD(QURT_PROT_BUS_BOUNDS,(n))

22.9.1.16 #define QURT_PROT_USER_MODE(*n*) QURT_MMAP_BUILD(QURT_PROT_USER_BOUNDS,(n))

22.9.1.17 #define QURT_MAP_PHYSADDR QURT_MMAP_BUILD(QURT_MAP_PHYSADDR_BOUNDS,1U)

Use the physical address that was passed in offset field. This is allowed only for root process.

22.9.1.18 #define QURT_MAP_TYPE(*n*) QURT_MMAP_BUILD(QURT_MAP_TYPE_BOUNDS,(n))

22.9.1.19 #define QURT_MAP_REGION(*n*) QURT_MMAP_BUILD(QURT_MAP_REGION_BOUNDS,(n))

23 System Environment

Programs can access properties of the QuRT system environment.

The maximum pimutex priority specifies the highest priority that a thread can be set to while it has the lock on a priority inheritance mutex. This value enables other threads that are not using pimutexes to run with a thread priority higher than the pimutex maximum priority.

The system environment supports the following operations:

- qurt_sysenv_get_app_heap()
- qurt_sysenv_get_arch_version()
- qurt_sysenv_get_max_hw_threads()
- qurt_sysenv_get_max_pi_prio()
- qurt_sysenv_get_process_name2()
- qurt_sysenv_get_process_name()
- qurt_sysenv_get_stack_profile_count()
- qurt_sysenv_get_hw_threads()
- Data Types

23.1 qurt_sysenv_get_app_heap()

23.1.1 Function Documentation

23.1.1.1 int qurt_sysenv_get_app_heap (qurt_sysenv_app_heap_t * *aheap*)

Gets information on the program heap from the kernel.

Associated data types

qurt_sysenv_app_heap_t

Parameters

out	*aheap*	Pointer to information on the program heap.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.2 qurt_sysenv_get_arch_version()

23.2.1 Function Documentation

23.2.1.1 int qurt_sysenv_get_arch_version (qurt_arch_version_t * *vers*)

Gets the Hexagon processor architecture version from the kernel.

Associated data types

qurt_arch_version_t

Parameters

out	*vers*	Pointer to the Hexagon processor architecture version.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter

Dependencies

None.

23.3 qurt_sysenv_get_max_hw_threads()

23.3.1 Function Documentation

23.3.1.1 int qurt_sysenv_get_max_hw_threads (qurt_sysenv_max_hthreads_t * *mhwt*)

Gets the maximum number of hardware threads supported in the Hexagon processor. The API includes the disabled hardware threads to reflect the maximum hardware thread count. For example, if the image is configured for four hardware threads and hthread_mask is set to 0x5 in cust_config.xml, only HW0 and HW2 are initialized by QuRT. HW1 and HW3 are not used at all. Under such a scenario, qurt_sysenv_get_max_hw_threads() still returns four.

Associated data types

qurt_sysenv_max_hthreads_t

Parameters

out	*mhwt*	Pointer to the maximum number of hardware threads supported in the Hexagon processor.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.4 qurt_sysenv_get_max_pi_prio()

23.4.1 Function Documentation

23.4.1.1 int qurt_sysenv_get_max_pi_prio (qurt_sysenv_max_pi_prio_t * *mpip*)

Gets the maximum priority inheritance mutex priority from the kernel.

Associated data types

qurt_sysenv_max_pi_prio_t

Parameters

out	*mpip*	Pointer to the maximum priority inheritance mutex priority.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.5 qurt_sysenv_get_process_name2()

23.5.1 Function Documentation

23.5.1.1 int qurt_sysenv_get_process_name2 (int *client_handle*, qurt_sysenv_procname_t * *pname*)

Gets information on the system environment process names based on the client_handle argument.

Associated data types

qurt_sysenv_procname_t

Parameters

in	*client_handle*	Obtained from the current invocation function (Section 3.4.1).
out	*pname*	Pointer to information on the process names in the system.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.6 qurt_sysenv_get_process_name()

23.6.1 Function Documentation

23.6.1.1 int qurt_sysenv_get_process_name (qurt_sysenv_procname_t * *pname*)

Gets information on the system environment process names from the kernel.

Associated data types

qurt_sysenv_procname_t

Parameters

out	*pname*	Pointer to information on the process names in the system.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.7 qurt_sysenv_get_stack_profile_count()

23.7.1 Function Documentation

23.7.1.1 int qurt_sysenv_get_stack_profile_count (qurt_sysenv_stack_profile_count_t * *count*)

Gets information on the stack profile count from the kernel.

Associated data types

qurt_sysenv_stack_profile_count_t

Parameters

out	*count*	Pointer to information on the stack profile count.

Returns

QURT_EOK – Success.

Dependencies

None.

23.8 qurt_sysenv_get_hw_threads()

23.8.1 Function Documentation

23.8.1.1 int qurt_sysenv_get_hw_threads (qurt_sysenv_hthreads_t * *mhwt*)

Gets the number of hardware threads initialized by QuRT in Hexagon processor. For example, if the image is configured for four hardware threads and hthread_mask is set to 0x5 in cust_config.xml, QuRT only initializes HW0 and HW2. HW1 and HW3 are not used. In this scenario, qurt_sysenv_get_hw_threads() returns 2.

Associated data types

qurt_sysenv_hthreads_t

Parameters

out	*mhwt*	Pointer to the number of hardware threads active in the Hexagon processor.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.9 Data Types

23.9.1 Data Structure Documentation

23.9.1.1 struct qurt_sysenv_swap_pools_t

QuRT swap pool information type.

23.9.1.2 struct qurt_sysenv_app_heap_t

QuRT application heap information type.

23.9.1.3 struct qurt_arch_version_t

QuRT architecture version information type.

23.9.1.4 struct qurt_sysenv_max_hthreads_t

QuRT maximum hardware threads information type.

23.9.1.5 struct qurt_sysenv_hthreads_t

QuRT active hardware threads information type.

23.9.1.6 struct qurt_sysenv_max_pi_prio_t

QuRT maximum pi priority information type.

23.9.1.7 struct qurt_sysenv_procname_t

QuRT process name information type.

23.9.1.8 struct qurt_sysenv_stack_profile_count_t

QuRT stack profile count information type.

23.9.1.9 struct qurt_sysevent_error_t

QuRT system error event type.

Data fields

Type	Parameter	Description
unsigned int	thread_id	Thread ID.
unsigned int	fault_pc	Fault PC.
unsigned int	sp	Stack pointer.
unsigned int	badva	Virtual data address where the exception occurred.
unsigned int	cause	QuRT error result.
unsigned int	ssr	Supervisor status register.
unsigned int	fp	Frame pointer.
unsigned int	lr	Link register.

Type	Parameter	Description
unsigned int	pid	PID of the process to which this thread belongs.

23.9.1.10 struct qurt_sysevent_error_1_t

Data fields

Type	Parameter	Description
unsigned int	thread_id	Thread ID.
unsigned int	fault_pc	Fault PC.
unsigned int	sp	Stack pointer.
unsigned int	badva	Virtual data address where the exception occurred.
unsigned int	cause	QuRT error result.
unsigned int	ssr	Supervisor status register.
unsigned int	fp	Frame pointer.
unsigned int	lr	Link register.
unsigned int	pid	PID of the process to which this thread belongs.
unsigned int	fkey	Framekey.
unsigned int	reserved1	Reserved.
unsigned int	reserved2	Reserved.
unsigned int	reserved3	Reserved.

23.9.1.11 struct qurt_sysevent_pagefault_t

QuRT page fault error event information type.

Data fields

Type	Parameter	Description
qurt_thread_t	thread_id	Thread ID of the page fault thread.
unsigned int	fault_addr	Accessed address that caused the page fault.
unsigned int	ssr_cause	SSR cause code for the page fault.

24 Profiling

Threads use profiling to determine the cycle counts for selected parts of a user program. Use the collected data to determine the CPU utilization of a QuRT thread (or the entire QuRT user program system).

Profiling supports thread-specific cycle counting for both the running (executing) and idle (not executing) cycles. Resetting the counts enables cycle counting to be performed on specific parts of a user program.

All but one of the profile cycle counts are expressed in terms of processor cycles (the number of actual processor cycles executed by all hardware threads) as opposed to thread cycles (for example, the number of cycles executed by a specific hardware thread). Assuming six hardware threads, the following equation expresses the relation between these two cycle types:

```
thread_cycles = processor_cycles / 6
```

The qurt_profile_enable() operation selectively enables or disables profiling (which is disabled by default).

Note: Resetting the cycle counts must be done explicitly by calling the reset operations before starting cycle counting.

qurt_profile_get_threadid_pcycles() returns the current per-hardware thread running cycle counts for the specified QuRT thread (Section 3). This operation returns an array containing the current running cycle count for each hardware thread. Each count value represents the number of processor cycles that have elapsed on the corresponding hardware thread while that thread has been scheduled for the specified QuRT thread.

qurt_profile_get_thread_pcycles() returns the current running cycle count for the specified QuRT thread (Section 3). The count value represents the number of processor cycles that have elapsed on all hardware threads while that thread has been scheduled for the specified QuRT thread.

Note: This count value is equivalent to summing the per-hardware-thread cycle count values returned by the get profile thread ID processor cycles operation.

qurt_profile_get_thread_pcycles() returns the current running cycle counts for the current QuRT thread, expressed in terms of thread cycles.

The qurt_profile_get_idle_pcycles() operation returns the current idle cycle count (the number of cycles a hardware thread is in IDLE state and not executing any instructions). This operation returns an array containing the current idle cycle count for each hardware thread. Each count value represents the number of processor cycles that have elapsed on the corresponding hardware thread while that thread has been in Wait mode.

Note: Cycles executed in the kernel are classified as idle or running according to the state that the current thread was in (for example, idle or running) when transitioning to the kernel.

qurt_get_core_pcycles() returns the number of processor cycles executed since the Hexagon processor was last reset. This value is based on the hardware core clock, which varies in speed according to the processor clock frequency (and differs from the system clock described in Section 16).

In a specified time duration, the relationship between the number of cycles elapsed by this operation, and the values returned by the get profile thread/idle processor cycles operations (both described above), is expressed by the following equation:

```
total_PCYCLES = run_pcycles + idle_pcycles
```

In this equation, qurt_get_core_pcycles() returns the total_PCYCLES value.

run_pcycles and idle_pcycles are defined in terms of the cycle count values returned by the get profile thread/idle processor cycles operations:

```
for (<all QuRT threads>)      for (i = 0; i < MAX_HW_THREADS; i++)
run_pcycles += profile_thread_pcycles[i] / 6;
for (i = 0; i < MAX_HW_THREADS; i++)
idle_pcycles += profile_idle_pcycles[i] / 6;
```

The cycle counts are summed on a per-thread basis, therefore the above code must convert each processor cycle count to a thread cycle count (by dividing by 6).

Note: The hardware core clock stops running when the Hexagon processor shuts down (due to all its hardware threads being idle), treat the cycle values returned by this operation as relative rather than absolute.

Computing CPU utilization – The CPU utilization for a QuRT thread (or an entire QuRT application system) indicates how many of the cycles that the Hexagon processor executed in a specified period of time were used by a specific thread (or by the application system):

```
CPU_utilization = run_pcycles / total_PCYCLES
```

In this equation, run_pcycles is the cycle count value returned by qurt_profile_get_thread_pcycles().

total_PCYCLES is the value returned by qurt_get_core_pcycles().

The Hexagon processor might have spent part of the specified time period in Power-saving mode, where the hardware core clock is completely shut down (because all the hardware threads are idle). In this case, the value in total_PCYCLES does not represent the absolute time.

To accurately compute the CPU utilization in this case, adjust total_PCYCLES by the core clock shutdown time. Compute the shutdown time (also called the ALL_WAIT period) from the QuRT system clock using the following equation:

```
ALL_WAIT_pcycles = ((total_sclk_samples / QTIMER_clock_freq)*
core_clock_freq) - total_PCYCLES
```

In this equation total_sclk_samples is the number of cycles elapsed in the QuRT system clock (Section 16).

total_PCYCLES is the value returned by the get core processor cycles operation. QTIMER_clock_freq is 19.2 MHz on all target systems.

core_clock_freq is the Hexagon processor core clock frequency (which is specific to each target system).

Taking the ALL_WAIT period into consideration, the adjusted CPU utilization is:

```
CPU_utilization = run_pcycles / (total_PCYCLES + ALL_WAIT_pcycles)
```

Note: The ALL_WAIT_pcycles equation assumes that the Hexagon processor core clock frequency does not change during the time interval profiled. If the clock frequency does change in this interval, the input values must be corrected because the weight of each sample is different.

For more information on profiling QuRT threads, see Appendix A.

Profiling is performed with the following operations:

- qurt_get_core_pcycles()
- qurt_profile_enable()
- qurt_profile_enable2()
- qurt_profile_get()
- qurt_profile_get_idle_pcycles()
- qurt_profile_get_thread_pcycles()
- qurt_get_hthread_pcycles()
- qurt_get_hthread_commits()
- qurt_profile_get_threadid_pcycles()
- qurt_profile_reset_idle_pcycles()
- qurt_profile_reset_threadid_pcycles()
- Data Types
- Macros

24.1 qurt_get_core_pcycles()

24.1.1 Function Documentation

24.1.1.1 unsigned long long int qurt_get_core_pcycles (void)

Gets the count of core processor cycles executed.

Returns the current number of running processor cycles executed since the Hexagon processor was last reset.

This value is based on the hardware core clock, which varies in speed according to the processor clock frequency.

NOTE Because the hardware core clock stops running when the processor shuts down (due to all of the hardware threads being idle), treat the cycle values returned by this operation as relative rather than absolute.

Thread cycle counts are valid only in the V4 Hexagon processor version.

Returns

Integer – Count of core processor cycles.

Dependencies

None.

24.2 qurt_profile_enable()

24.2.1 Function Documentation

24.2.1.1 void qurt_profile_enable (int *enable*)

Enables profiling.

Enables or disables cycle counting of the running and idle processor cycles. Profiling is disabled by default.

NOTE Enabling profiling does not automatically reset the cycle counts – reset must be done explicitly by calling the reset operations before starting cycle counting.

Parameters

in	*enable*	Profiling. Values: • 0 – Disable profiling • 1 – Enable profiling

Returns

None.

Dependencies

None.

24.3 qurt_profile_enable2()

24.3.1 Function Documentation

24.3.1.1 int qurt_profile_enable2 (qurt_profile_param_t *param*, qurt_thread_t *thread_id*, int *enable*)

Starts profiling of a specific parameter on a specific thread (as applicable).

Parameters

in	*param*	Profiling parameter.
in	*thread_id*	ID of the thread (if applicable) for which the specified paramter must be profiled.
in	*enable*	QURT_PROFILE_DISABLE – disable QURT_PROFILE_ENABLE – enable

Returns

QURT_EOK – Success
QURT_EALREADY – Measurement already in progress or already stopped
QURT_ENOTHREAD – Thread does not exist
QURT_EINVALID – Invalid profiling parameter

Dependencies

None.

24.4 qurt_profile_get()

24.4.1 Function Documentation

24.4.1.1 int qurt_profile_get (qurt_profile_param_t *param*, qurt_thread_t *thread_id*, qurt_profile_result_t * *result*)

Gets the value of the profiling parameter that was previously enabled.

Parameters

in	*param*	Profiling parameter.
in	*thread_id*	ID of thread (if applicable) for which the specified profiling paramter must be retrieved.
out	*result*	Profiling result associated with the parameter for the specified thread (if applicable).

Returns

QURT_EOK – Success
QURT_EFAILED – Operation failed; profiling was not enabled
QURT_ENOTHREAD – Thread does not exist
QURT_EINVALID – Invalid profiling parameter

Dependencies

None.

24.5 qurt_profile_get_idle_pcycles()

24.5.1 Function Documentation

24.5.1.1 void qurt_profile_get_idle_pcycles (unsigned long long * *pcycles*)

Gets the counts of idle processor cycles.

Returns the current idle processor cycle counts for all hardware threads.

This operation accepts a pointer to a user-defined array, and writes to the array the current idle cycle count for each hardware thread.

Each count value represents the number of processor cycles that have elapsed on the corresponding hardware thread while that thread has been in Wait mode.

NOTE This operation does not return the idle cycles that occur when the Hexagon processor shuts down (due to all of the hardware threads being idle).

Parameters

out	*pcycles*	User array [0..MAX_HW_THREADS-1] where the function stores the current idle cycle count values.

Returns

None.

Dependencies

None.

24.6 qurt_profile_get_thread_pcycles()

24.6.1 Function Documentation

24.6.1.1 unsigned long long int qurt_profile_get_thread_pcycles (void)

Gets the count of the running processor cycles for the current thread.

Returns the current running processor cycle count for the current QuRT thread.

Returns

Integer – Running processor cycle count for current thread.

Dependencies

None.

24.7 qurt_get_hthread_pcycles()

24.7.1 Function Documentation

24.7.1.1 unsigned int qurt_get_hthread_pcycles (int *n*)

Reads the GCYCLE_nT register to allow performance measurement when N threads are in Run mode.

NOTE Returns 0 when architecture is earlier than v67 or for invalid hardware thread ID.

Parameters

in	*n*	Threads in run mode. Valid values are 1 through maximum hardware threads.

Returns

Value read from GCYCLE_nT register. This value indicates the total number of pcycles that executed from reset to the current point of execution when n threads are in Run mode

Dependencies

PMU must be enabled.

24.8 qurt_get_hthread_commits()

24.8.1 Function Documentation

24.8.1.1 unsigned int qurt_get_hthread_commits (int *n*)

Reads the GCOMMIT_nT register to allow performance measurement when N threads are in Run mode.

NOTE Returns 0 when architecture is earlier than v67 or for invalid hardware thread ID.

Parameters

in	*n*	Threads in run mode. Valid values: 1 through maximum hardware threads.

Returns

Value read from the GCOMMIT_nT register. This value indicates the total number of packets committed from reset to the current point of execution when n threads are in Run mode.

Dependencies

PMU must be enabled.

24.9 qurt_profile_get_threadid_pcycles()

24.9.1 Function Documentation

24.9.1.1 void qurt_profile_get_threadid_pcycles (int *thread_id,* unsigned long long * *pcycles*)

Gets the counts of the running processor cycles for the specified QuRT thread.

Returns the current per-hardware-thread running cycle counts for the specified QuRT thread.

Each count value represents the number of processor cycles that have elapsed on the corresponding hardware thread while that thread has been scheduled for the specified QuRT thread.

Parameters

in	*thread_id*	Thread identifier.
out	*pcycles*	Pointer to a user array [0..MAX_HW_THREADS-1] where the function stores the current running cycle count values.

Returns

None.

Dependencies

None.

24.10 qurt_profile_reset_idle_pcycles()

24.10.1 Function Documentation

24.10.1.1 void qurt_profile_reset_idle_pcycles (void)

Sets the per-hardware-thread idle cycle counts to zero.

Returns

None.

Dependencies

None.

24.11 qurt_profile_reset_threadid_pcycles()

24.11.1 Function Documentation

24.11.1.1 void qurt_profile_reset_threadid_pcycles (int *thread_id*)

Sets the per-hardware-thread running cycle counts to zero for the specified QuRT thread.

Parameters

in	*thread_id*	Thread identifier.

Returns

None.

Dependencies

None.

24.12 Data Types

24.12.1 Data Structure Documentation

24.12.1.1 union qurt_profile_result_t

Profiling results.

Data fields

Type	Parameter	Description
struct qurt_-profile_result_t	thread_ready_-time	Result associated with QURT_PROFILE_PARAM_THREAD_READY_TIME.

24.12.1.2 struct qurt_profile_result_t.thread_ready_time

Result associated with QURT_PROFILE_PARAM_THREAD_READY_TIME.

Data fields

Type	Parameter	Description
unsigned int	ticks	Cumulative ticks the thread was ready.

24.13 Macros

24.13.1 Define Documentation

24.13.1.1 #define QURT_PROFILE_DISABLE 0

Disable profiling.

24.13.1.2 #define QURT_PROFILE_ENABLE 1

Enable profiling.

24.13.1.3 #define QURT_PROFILE_PARAM_THREAD_READY_TIME 0U

Profile thread ready time.

25 Performance Monitor

Threads use the performance monitor to measure code performance in real time during user program execution.

The performance monitor unit (PMU) is a hardware feature in the Hexagon processor. It is controlled by accessing a set of dedicated processor registers.

The performance monitor is controlled in QuRT with the following operations:

- qurt_pmu_enable()
- qurt_pmu_get()
- qurt_pmu_set()
- qurt_pmu_get_pmucnt()
- Macros

25.1 qurt_pmu_enable()

25.1.1 Function Documentation

25.1.1.1 void qurt_pmu_enable (int *enable*)

Enables or disables the Hexagon processor PMU. Profiling is disabled by default.

NOTE Enabling profiling does not automatically reset the count registers – this must be done explicitly before starting event counting.

Parameters

in	*enable*	Performance monitor. Values: • 0 – Disable performance monitor • 1 – Enable performance monitor

Returns

None.

Dependencies

None.

25.2 qurt_pmu_get()

25.2.1 Function Documentation

25.2.1.1 unsigned int qurt_pmu_get (int *reg_id*)

Gets the PMU register.

Returns the current value of the specified PMU register.

Parameters

in	*reg_id*	PMU register. Values: • QURT_PMUCNT0 • QURT_PMUCNT1 • QURT_PMUCNT2 • QURT_PMUCNT3 • QURT_PMUCFG • QURT_PMUEVTCFG • QURT_PMUCNT4 • QURT_PMUCNT5 • QURT_PMUCNT6 • QURT_PMUCNT7 • QURT_PMUEVTCFG1

Returns

Integer – Current value of the specified PMU register.

Dependencies

None.

25.3 qurt_pmu_set()

25.3.1 Function Documentation

25.3.1.1 void qurt_pmu_set (int *reg_id,* unsigned int *reg_value*)

Sets the value of the specified PMU register.

NOTE Setting PMUEVTCFG automatically clears the PMU registers PMUCNT0 through PMUCNT3.

Parameters

in	*reg_id*	PMU register. Values: • QURT_PMUCNT0 • QURT_PMUCNT1 • QURT_PMUCNT2 • QURT_PMUCNT3 • QURT_PMUCFG • QURT_PMUEVTCFG • QURT_PMUCNT4 • QURT_PMUCNT5 • QURT_PMUCNT6 • QURT_PMUCNT7 • QURT_PMUEVTCFG1
in	*reg_value*	Register value.

Returns

None.

Dependencies

None.

25.4 qurt_pmu_get_pmucnt()

25.4.1 Function Documentation

25.4.1.1 int qurt_pmu_get_pmucnt (void * *buf*)

Reads PMU counters in a single trap.

Parameters

out	*buf*	Pointer to a buffer to save values read from PMU counters. buffer size should be at least 32 bytes to read all eight PMU counters.

Returns

QURT_EOK – Successful read.
QURT_EFATAL – Failure.

Dependencies

None.

25.5 Macros

25.5.1 Define Documentation

25.5.1.1 #define QURT_PMUCNT0 0

25.5.1.2 #define QURT_PMUCNT1 1

25.5.1.3 #define QURT_PMUCNT2 2

25.5.1.4 #define QURT_PMUCNT3 3

25.5.1.5 #define QURT_PMUCFG 4

25.5.1.6 #define QURT_PMUEVTCFG 5

25.5.1.7 #define QURT_PMUCNT4 6

25.5.1.8 #define QURT_PMUCNT5 7

25.5.1.9 #define QURT_PMUCNT6 8

25.5.1.10 #define QURT_PMUCNT7 9

25.5.1.11 #define QURT_PMUEVTCFG1 10

25.5.1.12 #define QURT_PMUSTID0 11

25.5.1.13 #define QURT_PMUSTID1 12

25.5.1.14 #define QURT_PMUCNTSTID0 13

25.5.1.15 #define QURT_PMUCNTSTID1 14

25.5.1.16 #define QURT_PMUCNTSTID2 15

25.5.1.17 #define QURT_PMUCNTSTID3 16

25.5.1.18 #define QURT_PMUCNTSTID4 17

25.5.1.19 #define QURT_PMUCNTSTID5 18

25.5.1.20 #define QURT_PMUCNTSTID6 19

25.5.1.21 #define QURT_PMUCNTSTID7 20

26 Error Results

QuRT functions return error results in one of two ways:

- As function result values
- As values passed to the user-defined exception handler

QuRT defines a set of standard symbols for the error result values. This section lists the symbols and their corresponding values.

26.0.1 Define Documentation

26.0.1.1 #define QURT_EOK 0

Operation successfully performed.

26.0.1.2 #define QURT_EVAL 1

Wrong values for the parameters. The specified page does not exist.

26.0.1.3 #define QURT_EMEM 2

Not enough memory to perform the operation.

26.0.1.4 #define QURT_EINVALID 4

Invalid argument value; invalid key.

26.0.1.5 #define QURT_EFAILED 12

Operation failed.

26.0.1.6 #define QURT_ENOTALLOWED 13

Operation not allowed.

26.0.1.7 #define QURT_ENOREGISTERED 20

No registered interrupts.

26.0.1.8 #define QURT_ETLSAVAIL 23

No free TLS key is available.

26.0.1.9 #define QURT_ETLSENTRY 24

TLS key is not already free.

26.0.1.10 #define QURT_EINT 26

Invalid interrupt number (not registered).

26.0.1.11 #define QURT_ESIG 27

Invalid signal bitmask (cannot set more than one signal at a time).

26.0.1.12 #define QURT_ENOTHREAD 30

Thread no longer exists.

26.0.1.13 #define QURT_EALIGN 32

Not aligned.

26.0.1.14 #define QURT_EDEREGISTERED 33

Interrupt is already deregistered.

26.0.1.15 #define QURT_EEXISTS 35

File or message queue already exists.

26.0.1.16 #define QURT_ENAMETOOLONG 36

Name too long for message queue creation.

26.0.1.17 #define QURT_EPRIVILEGE 36

Caller does not have privilege for this operation.

26.0.1.18 #define QURT_ECANCEL 37

A cancellable request was canceled because the associated process was asked to exit.

26.0.1.19 #define QURT_EISLANDUSEREXIT 44

User call has resulted in island exit.

26.0.1.20 #define QURT_ENOISLANDENTRY 45

Island mode had not yet been entered.

26.0.1.21 #define QURT_EISLANDINVALIDINT 46

Exited Island mode due to an invalid island interrupt.

26.0.1.22 #define QURT_ETIMEDOUT 47

Operation timed-out.

26.0.1.23 #define QURT_EALREADY 48

Operation already in progress.

26.0.1.24 #define QURT_ERETRY 49 /∗< Retry the operation. ∗/

26.0.1.25 #define QURT_ENORESOURCE 56

No resource.

26.0.1.26 #define QURT_EDTINIT 57

Problem with device tree intialization.

26.0.1.27 #define QURT_EDESTROY 94

A destroy request was made to waiting threads.

26.0.1.28 #define QURT_EFATAL -1

Fatal error.

26.0.1.29 #define QURT_EXCEPT_PRECISE 0x01U

Precise exception occurred. For this cause code, Cause2 is SSR[7:0].

26.0.1.30 #define QURT_EXCEPT_NMI 0x02U

NMI occurred; Cause2 is not defined.

26.0.1.31 #define QURT_EXCEPT_TLBMISS 0x03U

TLBMISS RW occurred; for this cause code, Cause2 is SSR[7:0].

26.0.1.32 #define QURT_EXCEPT_RSVD_VECTOR 0x04U

Interrupt raised on a reserved vector, which must never occur. Cause2 is not defined.

26.0.1.33 #define QURT_EXCEPT_ASSERT 0x05U

Kernel assert. Cause2 QURT_ABORT_∗ are listed below.

26.0.1.34 #define QURT_EXCEPT_BADTRAP 0x06U

trap0(num) called with unsupported num. Cause2 is 0.

26.0.1.35 #define QURT_EXCEPT_UNDEF_TRAP1 0x07U

Trap1 is not supported. Using Trap1 causes this error. Cause2 is not defined.

26.0.1.36 #define QURT_EXCEPT_EXIT 0x08U

Application called qurt_exit() or qurt_exception_raise_nonfatal(). Can be called from C library. Cause2 is "[Argument passed to qurt_exception_raise_nonfatal() & 0xFF]".

26.0.1.37 #define QURT_EXCEPT_TLBMISS_X 0x0AU

TLBMISS X (execution) occurred. Cause2 is not defined.

26.0.1.38 #define QURT_EXCEPT_STOPPED 0x0BU

Running thread stopped due to fatal error on other hardware thread. Cause2 is not defined.

26.0.1.39 #define QURT_EXCEPT_FATAL_EXIT 0x0CU

Application called qurt_fatal_exit(). Cause2 is not defined.

26.0.1.40 #define QURT_EXCEPT_INVALID_INT 0x0DU

Kernel received an invalid L1 interrupt. Cause2 is not defined.

26.0.1.41 #define QURT_EXCEPT_FLOATING_POINT 0x0EU

Kernel received an floating point error. Cause2 is not defined.

26.0.1.42 #define QURT_EXCEPT_DBG_SINGLE_STEP 0x0FU

Cause2 is not defined.

26.0.1.43 #define QURT_EXCEPT_TLBMISS_RW_ISLAND 0x10U

Read write miss in Island mode. Cause2 QURT_TLB_MISS_RW_MEM∗ are listed below.

26.0.1.44 #define QURT_EXCEPT_TLBMISS_X_ISLAND 0x11U

Execute miss in Island mode. For this cause code, Cause2 is SSR[7:0].

26.0.1.45 #define QURT_EXCEPT_SYNTHETIC_FAULT 0x12U

Synthetic fault with user request that kernel detected. Cause2 QURT_SYNTH_* are listed below.

26.0.1.46 #define QURT_EXCEPT_INVALID_ISLAND_TRAP 0x13U

Invalid trap in Island mode. Cause2 is trap number.

26.0.1.47 #define QURT_EXCEPT_UNDEF_TRAP0 0x14U

trap0(num) was called with unsupported num. Cause2 is trap number.

26.0.1.48 #define QURT_EXCEPT_PRECISE_DMA_ERROR 0x28U

Precise DMA error. Cause2 is DM4[15:8]. Badva is DM5 register.

26.0.1.49 #define QURT_ECODE_UPPER_LIBC (0U << 16)

Upper 16 bits is 0 for libc.

26.0.1.50 #define QURT_ECODE_UPPER_QURT (0U << 16)

Upper 16 bits is 0 for QuRT.

26.0.1.51 #define QURT_ECODE_UPPER_ERR_SERVICES (2U << 16)

Upper 16 bits is 2 for error service.

26.0.1.52 #define QURT_SYNTH_ERR 0x01U

26.0.1.53 #define QURT_SYNTH_INVALID_OP 0x02U

26.0.1.54 #define QURT_SYNTH_DATA_ALIGNMENT_FAULT 0x03U

26.0.1.55 #define QURT_SYNTH_FUTEX_INUSE 0x04U

26.0.1.56 #define QURT_SYNTH_FUTEX_BOGUS 0x05U

26.0.1.57 #define QURT_SYNTH_FUTEX_ISLAND 0x06U

26.0.1.58 #define QURT_SYNTH_FUTEX_DESTROYED 0x07U

26.0.1.59 #define QURT_SYNTH_PRIVILEGE_ERR 0x08U

26.0.1.60 #define QURT_ABORT_FUTEX_WAKE_MULTIPLE 0x01U

futex_asm.s: Abort cause - futex wake multiple.

26.0.1.61 #define QURT_ABORT_WAIT_WAKEUP_SINGLE_MODE 0x02U

power.c: Abort cause - thread waits to wake up in Single Threaded mode.

26.0.1.62 #define QURT_ABORT_TCXO_SHUTDOWN_NOEXIT 0x03U

power.c : Abort cause - call TCXO shutdown without exit.

26.0.1.63 #define QURT_ABORT_FUTEX_ALLOC_QUEUE_FAIL 0x04U

futex.c: Abort cause - futex allocation queue failure - QURTK_futexhash_lifo empty.

26.0.1.64 #define QURT_ABORT_INVALID_CALL_QURTK_WARM_INIT 0x05U

init_asm.S: Abort cause - invalid call QURTK_warm_init() in NONE CONFIG_POWER_MGMT mode.

26.0.1.65 #define QURT_ABORT_THREAD_SCHEDULE_SANITY 0x06U

switch.S: Abort cause - sanity schedule thread is not supposed to run on the current hardware thread.

26.0.1.66 #define QURT_ABORT_REMAP 0x07U

Remap in the page table; the correct behavior must remove mapping if necessary.

26.0.1.67 #define QURT_ABORT_NOMAP 0x08U

No mapping in page table when removing a user mapping.

26.0.1.68 #define QURT_ABORT_INVALID_MEM_MAPPING_TYPE 0x0AU

Invalid memory mapping type when creating qmemory.

26.0.1.69 #define QURT_ABORT_NOPOOL 0x0BU

No pool available to attach.

26.0.1.70 #define QURT_ABORT_LIFO_REMOVE_NON_EXIST_ITEM 0x0CU

Cannot allocate more futex waiting queue.

26.0.1.71 #define QURT_ABORT_ASSERT 0x0EU

Assert abort.

26.0.1.72 #define QURT_ABORT_FATAL 0x0FU

Fatal error; must never occur.

26.0.1.73 #define QURT_ABORT_FUTEX_RESUME_INVALID_QUEUE 0x10U

futex_asm.s: Abort cause - invalid queue ID in futex resume.

26.0.1.74 #define QURT_ABORT_FUTEX_WAIT_INVALID_QUEUE 0x11U

futex_asm.s: Abort cause - invalid queue ID in futex wait.

26.0.1.75 #define QURT_ABORT_FUTEX_RESUME_INVALID_FUTEX 0x12U

futex.c: Abort cause - invalid futex object in hashtable.

26.0.1.76 #define QURT_ABORT_NO_ERHNDLR 0x13U

No registered error handler.

26.0.1.77 #define QURT_ABORT_ERR_REAPER 0x14U

Exception in the reaper thread.

26.0.1.78 #define QURT_ABORT_FREEZE_UNKNOWN_CAUSE 0x15U

Abort in thread freeze operation.

26.0.1.79 #define QURT_ABORT_FUTEX_WAIT_WRITE_FAILURE 0x16U

During futex wait processing, could not perform a necessary write operation to userland data; most likely due to a DL Pager eviction.

26.0.1.80 #define QURT_ABORT_ERR_ISLAND_EXP_HANDLER 0x17U

Exception in Island exception handler task.

26.0.1.81 #define QURT_ABORT_L2_TAG_DATA_CHECK_FAIL 0x18U

Detected error in L2 tag/data during warm boot. The L2 tag/data check is done when CONFIG_DEBUG_L2_POWER_COLLAPSE is enabled.

26.0.1.82 #define QURT_ABORT_ERR_SECURE_PROCESS 0x19U

Abort error in secure process.

26.0.1.83 #define QURT_ABORT_ERR_EXP_HANDLER 0x20U

No exception handler, or the handler caused an exception.

26.0.1.84 #define QURT_ABORT_ERR_NO_PCB 0x21U

PCB of the thread context failed initialization, PCB was NULL.

26.0.1.85 #define QURT_ABORT_NO_PHYS_ADDR 0x22U

Unable to find the physical address for the virtual address.

26.0.1.86 #define QURT_ABORT_OUT_OF_FASTINT_CONTEXTS 0x23U

Fast interrupt contexts exhausted.

26.0.1.87 #define QURT_ABORT_CLADE_ERR 0x24U

Fatal error seen with CLADE interrupt.

26.0.1.88 #define QURT_ABORT_ETM_ERR 0x25U

Fatal error seen with ETM interrupt.

26.0.1.89 #define QURT_ABORT_ECC_DED_ASSERT 0x26U

ECC two-bit DED error.

26.0.1.90 #define QURT_ABORT_VTLB_ERR 0x27U

Fatal error in the VTLB layer.

26.0.1.91 #define QURT_ABORT_TLB_ENCODE_DECODE_FAILURE 0x28U

Failure during the TLB encode or decode operation.

26.0.1.92 #define QURT_ABORT_VTLB_WALKOBJS_BOUND_FAILURE 0x29U

Failure to lookup entry in the page table.

26.0.1.93 #define QURT_TLB_MISS_X_FETCH_PC_PAGE 0x60U

26.0.1.94 #define QURT_TLB_MISS_X_2ND_PAGE 0x61U

26.0.1.95 #define QURT_TLB_MISS_X_ICINVA 0x62U

26.0.1.96 #define QURT_TLB_MISS_RW_MEM_READ 0x70U

26.0.1.97 #define QURT_TLB_MISS_RW_MEM_WRITE 0x71U

26.0.1.98 #define QURT_FLOATING_POINT_EXEC_ERR 0xBFU

Execute floating-point.

26.0.1.99 #define QURT_AUTOSTACKV2_CANARY_NOT_MATCH 0xC1U

Cause2 - autostackv2 - 8 bits

26.0.1.100 #define QURT_CFI_VIOLATION 0xC3U

Cause2 - CFI violation - 8 bits

26.0.1.101 #define QURT_FP_EXCEPTION_ALL 0x1FU << 25

26.0.1.102 #define QURT_FP_EXCEPTION_INEXACT 0x1U << 29

26.0.1.103 #define QURT_FP_EXCEPTION_UNDERFLOW 0x1U << 28

26.0.1.104 #define QURT_FP_EXCEPTION_OVERFLOW 0x1U << 27

26.0.1.105 #define QURT_FP_EXCEPTION_DIVIDE0 0x1U << 26

26.0.1.106 #define QURT_FP_EXCEPTION_INVALID 0x1U << 25

27 Function Tracing

QuRT supports function tracing to assist in debugging programs.

- qurt_trace_changed()
- qurt_trace_get_marker()
- qurt_etm_set_pc_range()
- qurt_etm_set_range()
- qurt_etm_set_atb()
- qurt_etm_set_sync_period()
- qurt_stm_trace_set_config()
- Data Types
- Macros

27.1 qurt_trace_changed()

27.1.1 Function Documentation

27.1.1.1 int qurt_trace_changed (unsigned int *prev_trace_marker,* unsigned int *trace_mask*)

Determines whether specific kernel events have occurred.

Returns a value that indicates whether the specified kernel events are recorded in the kernel trace buffer since the specified kernel trace marker was obtained.

The prev_trace_marker parameter specifies a kernel trace marker that was obtained by calling qurt_trace_get_marker().

NOTE Used with qurt_trace_get_marker(), this function determines whether certain kernel events occurred in a block of code.

This function cannot determine whether a specific kernel event type has occurred unless that event type has been enabled in the trace_mask element of the system configuration file.

QuRT supports the recording of interrupt and context switch events only (such as a trace_mask value of 0x3).

Parameters

in	*prev_trace_marker*	Previous kernel trace marker.
in	*trace_mask*	Mask value that indicates which kernel events to check for.

Returns

1 – Kernel events of the specified type have occurred since the specified trace marker was obtained.
0 – No kernel events of the specified type have occurred since the specified trace marker was obtained.

Dependencies

None.

27.2 qurt_trace_get_marker()

27.2.1 Function Documentation

27.2.1.1 unsigned int qurt_trace_get_marker (void)

Gets the kernel trace marker.

Returns the current value of the kernel trace marker. The marker consists of a hardware thread identifier and an index into the kernel trace buffer. The trace buffer records kernel events.

NOTE Using this function with qurt_trace_changed() determines whether certain kernel events occurred in a block of code.

Returns

Integer – Kernel trace marker.

Dependencies

None.

27.3 qurt_etm_set_pc_range()

27.3.1 Function Documentation

27.3.1.1 unsigned int qurt_etm_set_pc_range (unsigned int *range_num,* unsigned int *low_addr,* unsigned int *high_addr*)

Sets the PC address range for ETM filtering. Depending on the Hexagon core design, a maximum of four PC ranges are supported.

Parameters

in	*range_num*	0 to 3.
in	*low_addr*	Lower boundary of PC address range.
in	*high_addr*	Higher boundary of PC address range.

Returns

QURT_ETM_SETUP_OK – Success.
QURT_ETM_SETUP_ERR – Failure.

Dependencies

None.

27.4 qurt_etm_set_range()

27.4.1 Function Documentation

27.4.1.1 unsigned int qurt_etm_set_range (unsigned int *addr_source_type*, unsigned int *trig_block_num*, unsigned int *pid*, unsigned int *low_addr*, unsigned int *high_addr*)

Sets the address range for ETM filtering. It allows the user to select the source type of addresses - QURT_ETM_SOURCE_PC and QURT_ETM_SOURCE_DATA.

Parameters

in	*addr_source_type*	Type of the address source: • QURT_ETM_SOURCE_PC • QURT_ETM_SOURCE_DATA
in	*trig_block_num*	0 to 3.
in	*pid*	ID of the process.
in	*low_addr*	Lower boundary of PC address range.
in	*high_addr*	Higher boundary of PC address range.

Returns

QURT_ETM_SETUP_OK – Success.
QURT_ETM_SETUP_ERR – Failure.

Dependencies

None.

27.5 qurt_etm_set_atb()

27.5.1 Function Documentation

27.5.1.1 unsigned int qurt_etm_set_atb (unsigned int *flag*)

Sets the advanced trace bus (ATB) state to notify QuRT that the ATB is actively enabled or disabled. QuRT performs the corresponding actions at low power management.

Parameters

in	*flag*	Values: QURT_ATB_ON QURT_ATB_OFF

Returns

QURT_ETM_SETUP_OK – Success.
QURT_ETM_SETUP_ERR – Failure

Dependencies

None.

27.6 qurt_etm_set_sync_period()

27.6.1 Function Documentation

27.6.1.1 unsigned int qurt_etm_set_sync_period (unsigned int *sync_type,* unsigned int *period*)

Sets the period for types of synchronization trace packets.

ASYNC defines the period between alignment synchronization packets. Period is in terms of bytes in the packet stream.

ISYNC defines the period between instruction synchronization packets. Period is per thread and is defined as the bytes sent out for that thread.

GSYNC is the defined period in thread cycles between GSYNC packets.

Parameters

in	*sync_type*	Type of synchronization packets: QURT_ETM_ASYNC_PERIOD QURT_ETM_ISYNC_PERIOD QURT_ETM_GSYNC_PERIOD
in	*period*	Period value.

Returns

QURT_ETM_SETUP_OK – Success.
QURT_ETM_SETUP_ERR – Failure.

Dependencies

None.

27.7 qurt_stm_trace_set_config()

27.7.1 Function Documentation

27.7.1.1 unsigned int qurt_stm_trace_set_config (qurt_stm_trace_info_t * *stm_config_info*)

Sets up a STM port for tracing events.

Associated data types

qurt_stm_trace_info_t

Parameters

in	*stm_config_info*	Pointer to the STM trace information used to set up the trace in the kernel. The strucure must have the following: • One port address per hardware thread • Event ID for context switches • Event ID for interrupt tracing n • Header or marker to identify the beginning of the trace.

Returns

QURT_EOK – Success.
QURT_EINVALID – Failure; possibly because the passed port address is not in the page table.

Dependencies

None.

27.8 Data Types

27.8.1 Data Structure Documentation

27.8.1.1 struct qurt_stm_trace_info_t

STM trace information.

27.9 Macros

27.9.1 Define Documentation

27.9.1.1 #define QURT_ETM_SOURCE_PC 0U

ETM memory source of SAC∗ is PC.

27.9.1.2 #define QURT_ETM_SOURCE_DATA 1U

ETM memory source of SAC∗ is data.

27.9.1.3 #define QURT_ETM_ASYNC_PERIOD 0

Async.

27.9.1.4 #define QURT_ETM_ISYNC_PERIOD 1

Isync.

27.9.1.5 #define QURT_ETM_GSYNC_PERIOD 2

Gsync.

27.9.1.6 #define QURT_ETM_SETUP_OK 0

ETM setup OK.

27.9.1.7 #define QURT_ETM_SETUP_ERR 1

ETM setup error.

27.9.1.8 #define QURT_ATB_OFF 0

ATB off.

27.9.1.9 #define QURT_ATB_ON 1

ATB on.

27.9.1.10 #define QURT_TRACE(*str, ...*) __VA_ARGS__

Function tracing is implemented with the QURT_TRACE debug macro, which optionally generates printf statements both before and after every function call that is passed as a macro argument.

For example, in the following macro calls in the source code:

```
QURT_TRACE(myfunc, my_func(33))
```

generates the following debug output:

```
myfile:nnn: my_func >>> calling my_func(33)
myfile:nnn: my_func >>> returned my_func(33)
```

The debug output includes the source file and line number of the function call, along with the text of the call. Compile the client source file with -D __FILENAME__ defined for its file name.

The library function qurt_printf() generates the debug output. The QURT_DEBUG symbol controls generation of the debug output. If this symbol is not defined, function tracing is not generated.

NOTE The debug macro is accessed through the QuRT API header file.

28 Atomic Operations

QuRT kernel atomic operations are accessed with the following QuRT functions.

- qurt_atomic_set()
- qurt_atomic_and()
- qurt_atomic_and_return()
- qurt_atomic_or()
- qurt_atomic_or_return()
- qurt_atomic_xor()
- qurt_atomic_xor_return()
- qurt_atomic_set_bit()
- qurt_atomic_clear_bit()
- qurt_atomic_change_bit()
- qurt_atomic_add()
- qurt_atomic_add_return()
- qurt_atomic_add_unless()
- qurt_atomic_sub()
- qurt_atomic_sub_return()
- qurt_atomic_inc()
- qurt_atomic_inc_return()
- qurt_atomic_dec()
- qurt_atomic_dec_return()
- qurt_atomic_compare_and_set()
- qurt_atomic_barrier()
- qurt_atomic64_set()
- qurt_atomic64_and_return()
- qurt_atomic64_or()
- qurt_atomic64_or_return()
- qurt_atomic64_xor_return()

- qurt_atomic64_set_bit()
- qurt_atomic64_clear_bit()
- qurt_atomic64_change_bit()
- qurt_atomic64_add()
- qurt_atomic64_add_return()
- qurt_atomic64_sub_return()
- qurt_atomic64_inc()
- qurt_atomic64_inc_return()
- qurt_atomic64_dec_return()
- qurt_atomic64_compare_and_set()
- qurt_atomic64_barrier()

28.1 qurt_atomic_set()

28.1.1 Function Documentation

28.1.1.1 static QURT_INLINE unsigned int qurt_atomic_set (unsigned int * *target,* unsigned int *value*)

Sets the atomic variable with the specified value.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*value*	Value to set.

Returns

Value successfuly set.

Dependencies

None.

28.2 qurt_atomic_and()

28.2.1 Function Documentation

28.2.1.1 static QURT_INLINE void qurt_atomic_and (unsigned int * *target,* unsigned int *mask*)

Bitwise AND operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise AND.

Returns

None

Dependencies

None.

28.3 qurt_atomic_and_return()

28.3.1 Function Documentation

28.3.1.1 static QURT_INLINE unsigned int qurt_atomic_and_return (unsigned int * *target,* unsigned int *mask*)

Bitwise AND operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise AND.

Returns

AND result of atomic variable with mask.

Dependencies

None.

28.4 qurt_atomic_or()

28.4.1 Function Documentation

28.4.1.1 static QURT_INLINE void qurt_atomic_or (unsigned int * *target,* unsigned int *mask*)

Bitwise OR operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise OR.

Returns

None.

Dependencies

None.

28.5 qurt_atomic_or_return()

28.5.1 Function Documentation

28.5.1.1 static QURT_INLINE unsigned int qurt_atomic_or_return (unsigned int * *target,* unsigned int *mask*)

Bitwise OR operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise OR.

Returns

Returns the OR result of the atomic variable with mask.

Dependencies

None.

28.6 qurt_atomic_xor()

28.6.1 Function Documentation

28.6.1.1 static QURT_INLINE void qurt_atomic_xor (unsigned int * *target,* unsigned int *mask*)

Bitwise XOR operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise XOR.

Returns

None.

Dependencies

None.

28.7 qurt_atomic_xor_return()

28.7.1 Function Documentation

28.7.1.1 static QURT_INLINE unsigned int qurt_atomic_xor_return (unsigned int * *target,* unsigned int *mask*)

Bitwise XOR operation of the atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	Mask for bitwise XOR.

Returns

XOR result of atomic variable with mask.

Dependencies

None.

28.8 qurt_atomic_set_bit()

28.8.1 Function Documentation

28.8.1.1 static QURT_INLINE void qurt_atomic_set_bit (unsigned int * *target,* unsigned int *bit*)

Sets a bit in the atomic variable at a specified position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to set.

Returns

None.

Dependencies

None.

28.9 qurt_atomic_clear_bit()

28.9.1 Function Documentation

28.9.1.1 static QURT_INLINE void qurt_atomic_clear_bit (unsigned int * *target*, unsigned int *bit*)

Clears a bit in the atomic variable at a specified position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to clear.

Returns

None.

Dependencies

None.

28.10 qurt_atomic_change_bit()

28.10.1 Function Documentation

28.10.1.1 static QURT_INLINE void qurt_atomic_change_bit (unsigned int * *target,* unsigned int *bit*)

Toggles a bit in a atomic variable at a bit position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to toggle.

Returns

None.

Dependencies

None.

28.11 qurt_atomic_add()

28.11.1 Function Documentation

28.11.1.1 static QURT_INLINE void qurt_atomic_add (unsigned int * *target,* unsigned int *v*)

Adds an integer to atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	Integer value to add.

Returns

None.

Dependencies

None.

28.12 qurt_atomic_add_return()

28.12.1 Function Documentation

28.12.1.1 static QURT_INLINE unsigned int qurt_atomic_add_return (unsigned int * *target*, unsigned int *v*)

Adds an integer to atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	Integer value to add.

Returns

Result of arithmetic sum.

Dependencies

None.

28.13 qurt_atomic_add_unless()

28.13.1 Function Documentation

28.13.1.1 static QURT_INLINE unsigned int qurt_atomic_add_unless (unsigned int * *target*, unsigned int *delta*, unsigned int *unless*)

Adds the delta value to an atomic variable unless the current value in the target matches the unless variable.

NOTE The function retries until load lock and store conditional are successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*delta*	Value to add to the current value.
in	*unless*	Perform the addition only when the current value is not equal to this unless value.

Returns

TRUE – 1 - Addition was performed.
FALSE – 0 - Addition was not done.

Dependencies

None.

28.14 qurt_atomic_sub()

28.14.1 Function Documentation

28.14.1.1 static QURT_INLINE void qurt_atomic_sub (unsigned int * *target,* unsigned int *v*)

Subtracts an integer from an atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	Integer value to subtract.

Returns

None.

Dependencies

None.

28.15 qurt_atomic_sub_return()

28.15.1 Function Documentation

28.15.1.1 static QURT_INLINE unsigned int qurt_atomic_sub_return (unsigned int * *target*, unsigned int *v*)

Subtracts an integer from an atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	Integer value to subtract.

Returns

Result of arithmetic subtraction.

Dependencies

None.

28.16 qurt_atomic_inc()

28.16.1 Function Documentation

28.16.1.1 static QURT_INLINE void qurt_atomic_inc (unsigned int * *target*)

Increments an atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

None.

Dependencies

None.

28.17 qurt_atomic_inc_return()

28.17.1 Function Documentation

28.17.1.1 static QURT_INLINE unsigned int qurt_atomic_inc_return (unsigned int * *target*)

Increments an atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

Incremented value.

Dependencies

None.

28.18 qurt_atomic_dec()

28.18.1 Function Documentation

28.18.1.1 static QURT_INLINE void qurt_atomic_dec (unsigned int * *target*)

Decrements an atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

None.

Dependencies

None.

28.19 qurt_atomic_dec_return()

28.19.1 Function Documentation

28.19.1.1 static QURT_INLINE unsigned int qurt_atomic_dec_return (unsigned int * *target*)

Decrements an atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

Decremented value.

Dependencies

None.

28.20 qurt_atomic_compare_and_set()

28.20.1 Function Documentation

28.20.1.1 static QURT_INLINE unsigned int qurt_atomic_compare_and_set (unsigned int * *target*, unsigned int *old_val*, unsigned int *new_val*)

Compares the current value of the atomic variable with the specified value and set to a new value when compare is successful.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*old_val*	Old value to compare.
in	*new_val*	New value to set.

Returns

FALSE – Specified value is not equal to the current value.
TRUE --Specified value is equal to the current value.

Dependencies

None.

28.21 qurt_atomic_barrier()

28.21.1 Function Documentation

28.21.1.1 static QURT_INLINE void qurt_atomic_barrier (void)

Allows the compiler to enforce an ordering constraint on memory operation issued before and after the function.

Returns

None.

Dependencies

None.

28.22 qurt_atomic64_set()

28.22.1 Function Documentation

28.22.1.1 static QURT_INLINE unsigned long long qurt_atomic64_set (unsigned long long * *target,* unsigned long long *value*)

Sets the 64-bit atomic variable with the specified value.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*value*	64-bit value to set.

Returns

Successfuly set value.

Dependencies

None.

28.23 qurt_atomic64_and_return()

28.23.1 Function Documentation

28.23.1.1 static QURT_INLINE unsigned long long qurt_atomic64_and_return (unsigned long long * *target*, unsigned long long *mask*)

Bitwise AND operation of a 64-bit atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	64-bit mask for bitwise AND.

Returns

AND result of 64-bit atomic variable with mask.

Dependencies

None.

28.24 qurt_atomic64_or()

28.24.1 Function Documentation

28.24.1.1 static QURT_INLINE void qurt_atomic64_or (unsigned long long * *target,* unsigned long long *mask*)

Bitwise OR operation of a 64-bit atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	64-bit mask for bitwise OR.

Returns

None.

Dependencies

None.

28.25 qurt_atomic64_or_return()

28.25.1 Function Documentation

28.25.1.1 static QURT_INLINE unsigned long long qurt_atomic64_or_return (unsigned long long * *target*, unsigned long long *mask*)

Bitwise OR operation of a 64-bit atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	64-bit mask for bitwise OR.

Returns

OR result of the atomic variable with mask.

Dependencies

None.

28.26 qurt_atomic64_xor_return()

28.26.1 Function Documentation

28.26.1.1 static QURT_INLINE unsigned long long qurt_atomic64_xor_return (unsigned long long * *target*, unsigned long long *mask*)

Bitwise XOR operation of 64-bit atomic variable with mask.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*mask*	64-bit mask for bitwise XOR.

Returns

XOR result of atomic variable with mask.

Dependencies

None.

28.27 qurt_atomic64_set_bit()

28.27.1 Function Documentation

28.27.1.1 static QURT_INLINE void qurt_atomic64_set_bit (unsigned long long * *target,* unsigned int *bit*)

Sets a bit in a 64-bit atomic variable at a specified position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to set.

Returns

None.

Dependencies

None.

28.28 qurt_atomic64_clear_bit()

28.28.1 Function Documentation

28.28.1.1 static QURT_INLINE void qurt_atomic64_clear_bit (unsigned long long * *target*, unsigned int *bit*)

Clears a bit in a 64-bit atomic variable at a specified position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to clear.

Returns

None.

Dependencies

None.

28.29 qurt_atomic64_change_bit()

28.29.1 Function Documentation

28.29.1.1 static QURT_INLINE void qurt_atomic64_change_bit (unsigned long long * *target*, unsigned int *bit*)

Toggles a bit in a 64-bit atomic variable at a bit position.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*bit*	Bit position to toggle.

Returns

None.

Dependencies

None.

28.30 qurt_atomic64_add()

28.30.1 Function Documentation

28.30.1.1 static QURT_INLINE void qurt_atomic64_add (unsigned long long * *target,* unsigned long long *v*)

Adds a 64-bit integer to 64-bit atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	64-bit integer value to add.

Returns

None.

Dependencies

None.

28.31 qurt_atomic64_add_return()

28.31.1 Function Documentation

28.31.1.1 static QURT_INLINE unsigned long long qurt_atomic64_add_return (unsigned long long * *target*, unsigned long long *v*)

Adds a 64-bit integer to 64-bit atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	64-bit integer value to add.

Returns

Result of arithmetic sum.

Dependencies

None.

28.32 qurt_atomic64_sub_return()

28.32.1 Function Documentation

28.32.1.1 static QURT_INLINE unsigned long long qurt_atomic64_sub_return (unsigned long long * *target*, unsigned long long *v*)

Subtracts a 64-bit integer from an atomic variable.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*v*	64-bit integer value to subtract.

Returns

Result of arithmetic subtraction.

Dependencies

None.

28.33 qurt_atomic64_inc()

28.33.1 Function Documentation

28.33.1.1 static QURT_INLINE void qurt_atomic64_inc (unsigned long long * *target*)

Increments a 64-bit atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

None.

Dependencies

None.

28.34 qurt_atomic64_inc_return()

28.34.1 Function Documentation

28.34.1.1 static QURT_INLINE unsigned long long qurt_atomic64_inc_return (unsigned long long * *target*)

Increments a 64-bit atomic variable by one

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

Incremented value.

Dependencies

None.

28.35 qurt_atomic64_dec_return()

28.35.1 Function Documentation

28.35.1.1 static QURT_INLINE unsigned long long qurt_atomic64_dec_return (unsigned long long * *target*)

Decrements a 64-bit atomic variable by one.

NOTE The function retries until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.

Returns

Decremented value.

Dependencies

None.

28.36 qurt_atomic64_compare_and_set()

28.36.1 Function Documentation

28.36.1.1 static QURT_INLINE int qurt_atomic64_compare_and_set (unsigned long long * *target,* unsigned long long *old_val,* unsigned long long *new_val*)

Compares the current value of an 64-bit atomic variable with the specified value and sets to a new value when compare is successful.

NOTE The function keep retrying until load lock and store conditional is successful.

Parameters

in,out	*target*	Pointer to the atomic variable.
in	*old_val*	64-bit old value to compare.
in	*new_val*	64-bit new value to set.

Returns

FALSE – Specified value is not equal to the current value.
TRUE – Specified value is equal to the current value.

Dependencies

None.

28.37 qurt_atomic64_barrier()

28.37.1 Function Documentation

28.37.1.1 static QURT_INLINE void qurt_atomic64_barrier (void)

Allows compiler to enforce an ordering constraint on memory operation issued before and after the function.

Returns

None.

Dependencies

None.

29 QuRT Callbacks

The QuRT RTOS defines a callback function that enables users to perform program-specific operations during certain QuRT system events.

The QuRT callback framework provides a safe mechanism for root process drivers to execute callback functions in a user process. The framework hosts dedicated worker threads in corresponding processes that handle the execution of the callback fuction. This ensures that the callbacks occur in context of the appropriate process thread, in result maintaining privilege boundaries.

Prerequisites for use of this framework are:

1. Driver is a QDI driver and client communicates with drivers using QDI invocations.

2. Callback configuration is specified in cust_config.xml for the user process that intends to use this framework.

qurt_cb_data_t is the public data structure that allows a client to store all the required information about the callback, including the callback function and the arguments to pass to this function when it executes. The client uses the QDI interface to register this structure with the root driver.

The callback framework provides two APIs that a root driver can use to invoke callbacks. These functions are described in the qurt_qdi.h header file.

qurt_qdi_cb_invoke_async() invokes an asynchronous callback, wherein the invoking driver does not wait for the callback to finish executing.

qurt_qdi_cb_invoke_sync() invokes a synchronous callback. Upon invocation, the invoking thread is suspended until the callback function finishes execution.

- qurt_cb_data_set_cbarg()
- qurt_cb_data_set_cbfunc()
- qurt_cb_data_init()
- Data Types

29.1 qurt_cb_data_set_cbarg()

29.1.1 Function Documentation

29.1.1.1 static void qurt_cb_data_set_cbarg (qurt_cb_data_t * *cb_data,* unsigned *cb_arg*)

Sets up the callback argument. This function sets up the argument passed to the callback function when it executes.

Associated data types

qurt_cb_data_t

Parameters

in	*cb_data*	Pointer to the callback data structure.
in	*cb_arg*	Argument for the callback function.

Returns

None.

Dependencies

None.

29.2 qurt_cb_data_set_cbfunc()

29.2.1 Function Documentation

29.2.1.1 static void qurt_cb_data_set_cbfunc (qurt_cb_data_t * *cb_data,* void * *cb_func*)

Sets up the callback function in the callback registration data structure.

Associated data types

qurt_cb_data_t

Parameters

in	*cb_data*	Pointer to the callback data structure.
in	*cb_func*	Pointer to the callback function.

Returns

None.

Dependencies

None.

29.3 qurt_cb_data_init()

29.3.1 Function Documentation

29.3.1.1 static void qurt_cb_data_init (qurt_cb_data_t * *cb_data*)

Initializes the callback data structure. Entity registering a callback with the root process driver must call this function to initialize callback registration data structure to the default value.

Associated data types

qurt_cb_data_t

Parameters

in	*cb_data*	Pointer to the callback data structure.

Returns

None.

Dependencies

None.

29.4 Data Types

This section describes data types for callbacks.

29.4.1 Data Structure Documentation

29.4.1.1 struct qurt_cb_data_t

Callback registration data structure.

30 SRM Drivers

SRM driver services are accessed with the following QuRT functions.

- qurt_srm_get_pid()

30.1 qurt_srm_get_pid()

30.1.1 Function Documentation

30.1.1.1 unsigned qurt_srm_get_pid (int *client_handle*)

Gets the PID for the client_handle that is passed.

Parameters

in	*client_handle*	Client handle for which PID is required.

Returns

PID of the client

Dependencies

None.

31 HVX

The Hexagon Vector Extension (HVX) is an optional component of the Hexagon DSP for vector operations.

To get the HVX hardware configuration that the chipset supports, use this API.

- qurt_hvx_get_units()

31.1 qurt_hvx_get_units()

31.1.1 Function Documentation

31.1.1.1 int qurt_hvx_get_units (void)

Gets the HVX hardware configuration that the chipset supports.

NOTE The function returns the HVX hardware configuration supported by the chipset.

Returns

Bitmask of the units: 1X64, 2X64, 4X64, 1X128, 2X128, and so on.

- QURT_HVX_HW_UNITS_2X126B_4X64B – V60, V62, or V65 HVX
- QURT_HVX_HW_UNITS_4X128B_0X64B – V66 CDSP or newer
- 0 – not available

Dependencies

None.

32 Predefined Symbols

QuRT predefines the symbol QURT_API_VERSION to support backwards compatibility of the QuRT API. This symbol returns a numeric value which represents a specific compatible version of the QuRT API.

QURT_API_VERSION is redefined with a new value only when a new version of the QuRT API is released that adds new API functions, or introduces changes to the existing API functions that make them incompatible with the previous API version.

Use the symbol in conditional compilation directives to write QuRT program code that works with more than one version of the QuRT API.

For example, consider the case of a QuRT API function which is redefined in a new version of the QuRT API (for example, version N) to have a second argument. The program code can then be written to conditionally use either version of this function:

```
#if QURT_API_VERSION < N
result = qurt_func (arg1);
#else /* QURT_API_VERSION < N */
result = qurt_func (arg1, arg2);
#endif /* QURT_API_VERSION < N */
```

NOTE The value of QURT_API_VERSION remains unchanged across multiple QuRT releases as long as the API compatibility is not affected by the new releases.

32.0.1 Define Documentation

32.0.1.1 #define QURT_API_VERSION 12

QURT API version.

A Thread-level Profiling

The profiling support in QuRT (Section 24) can be used to profile the execution of one or more QuRT threads individually, or the entire QuRT user program system as a whole.

The following sections describe the procedure for profiling QuRT threads. The description is presented in terms of a client/server model:

- The client resides outside the system, and is connected by some means to the server that is using the QuRT system.
- The client sends the profiling information in units of packets.
- The server processes the packets and plots a graph displaying the CPU utilization.

A.1 Server Behavior

The server receives and processes the following events:

- Start command from client
- Timer expiry
- Stop command from client

A.1.1 Start command

The start command specifies the sampling period for profiling. The use of a sampling period limits the overhead imposed on the overall system by the profiling task.

Upon receiving the start command, the server initializes its state by performing the following steps:

- Record the system clock using qurt_sysclock_get_hw_ticks(). This value is referred to as tick_base.
- Record the PCYCLE count from the core using qurt_get_core_pcycles(). This value is referred to as pcycle_base.
- Clear the pcycles of all threads of the system (or alternatively a specific subset of threads) using qurt_profile_reset_threadid_pcycles().
- Clear the idle thread pcycles using qurt_profile_reset_idle_pcycles().
- Start a periodic timer (Section 15) with the period specified by the sampling period received from the start command.
- Enable QuRT profiling using qurt_profile_enable(1).

A.1.2 Timer expiry

The timer expiry triggers the start of the collection of the profiling information. The server performs the following steps when the timer expires.

- Record the system clock using qurt_sysclock_attr_get_hw_ticks(). This value is referred to as tick_base. Compute the value ticks using the following equation: ticks = new_tick_base – tick_base
- Record the PCYCLE count from the core using qurt_get_core_pcycles().This value is referred to as pcycle_base. Compute the value total_pcycles using the following equation: total_pcycles = new_pcycle_base – pcycle_base
- Obtain the run time information of a thread using qurt_profile_get_thread_pcycles(). This value is referred to as pcycles.
- For each thread being profiled, construct a packet with the following information:
  - ticks
  - total_pcycles
  - pcycles
  - core_clock_freq
  - thread_ID
- Send the constructed packets to the client.

A.1.3 Stop command

Upon receiving the stop command, the server performs the following steps:

- Stop the periodic timer started by the start command.
- Disable QuRT profiling using qurt_profile_enable(0).

A.2 Client Behavior

The client accepts user input to start and stop profiling. It receives the packets sent by the server, and converts the information to absolute time.

When the client issues a start command, it resets to zero both the run time of each thread and the total run time. Assume that the client maintains the following values:

- prev_thread_pcycles
- prev_ticks
- thread_run_time
- system_run_time

All of these values are set to 0 when the client issues the start command.

Given the above values, the following logic can be used to determine the run time and CPU utilization of a QuRT thread.

```
net_run_pcycles = pcycles - prev_pcycles;
```

```
net_ticks = ticks - prev_ticks;
thread_run_time = thread_run_time +
(net_run_pcycles / (6 * core_clock_freq));
system_run_time = system_run_time +
(net_ticks / QTIMER_clk_freq);
prev_pcycles = pcycles;
prev_ticks = ticks;
```

This logic works even if the core clock frequency changes during the course of the profiling. Any change in the core clock frequency is limited to only a single iteration; therefore, the error accumulated is insignificant.

Note: The Qtimer clock frequency used above is fixed at 19.2 MHz on all target systems.

A.3 Profiling the System

System profiling can be performed efficiently without having to profile all the QuRT threads in the system.

The client can request the server to send idle information. For example, the server sends the idle thread information using the same parameters used for thread profiling; the only difference is that pcycles represents the idle thread run time.

The idle thread run time is equivalent to the idle time of the hardware thread (i.e., the duration the hardware thread spent in the wait state).

With minor modifications, the same logic used for thread profiling can be used to determine the run time and CPU utilization of the system:

```
net_run_pcycles = pcycles - prev_pcycles;
net_total_pcycles = total_pcycles - prev_total_pcycles;
net_ticks = ticks - prev_ticks;
run_time = net_run_time +
((net_total_pcycles - net_run_pcycles) /
(6 * core_clock_freq));
system_run_time = system_run_time +
(net_ticks / sleep_clk_freq);
prev_pcycles = pcycles;
prev_total_pcycles = total_pcycles;
prev_ticks = ticks;
```

This makes it possible to plot the CPU utilization of each hardware thread of the system without going through all the threads in the system.

B Debugging Errors and Cause Codes

B.1 Debugging Errors and Exceptions

The QuRT error handling routine populates the QURT_error_info data structure with error information.

This can be used to identify:

- Nonfatal (recoverable) errors, seen when:
  - User program raises an ASSERT using a QuRT API
  - User program results in an exception
- Fatal (non-recoverable) errors, seen:
  - When the program exception handler promotes a nonfatal error as a fatal error
  - When a QuRT program code (kernel) raises a fatal error (ASSERT)
  - When an exception is seen in Supervisor mode.
  - On imprecise exceptions

To triage crash dumps:

- Identify the reason behind the fatal errors by looking for values in QURT_error_info.status.cause and QURT_error_info.status.cause2 from the cause-cause2 table.
- For fatal errors, local and global registers are saved into QURT_error_info.locregs and QURT_error_info.globregs.
- At times, the application triggers the fatal error and in most cases is the program exception handler handling nonfatal errors. Use QURT_error_info.user_errors to triage nonfatal errors.
- For nonfatal exceptions, the local registers for the faulting thread are saved into TCB. Can be accessed by QURT_error_info.user_errors.entry[counter].error_tcb.
- For nonfatal errors raised through qurt_exception_raise_nonfatal(), only callee-saved registers are saved into TCB.

B.2 Cause Codes

Cause and cause2 are error codes to distinguish multiple errors. All cause and cause2 error codes can range from 1 to 255, and every cause can have 1 to 255 error codes. Hence the system can have up to 255 * 255 unique error codes.

The combination of cause and cause2 is represented as ((cause2 « 8) 'Logical OR' (cause)).

Some Cause2 codes are statically defined, whereas some are obtained from SSR[7:0] cause codes. SSR cause codes are defined in 80-V9418-27 and 80-V9418-25. All possible combinations are listed in Tables B.5 and B.6.

B.3 Debugging a Fatal Error

Error	Description
QURT_error_info.status.status	Indicates whether an error occurred.
QURT_error_info.status.cause	Cause code for fatal error, see Table B.5.
QURT_error_info.status.cause2	Cause2 code for fatal error, see Table B.6.
QURT_error_info.status.fatal	Indicates whether a fatal error occurred. A user error can result in a fatal error if the exception handler is not registered
QURT_error_info.status.hw_tnum	Indicates the index of QURT_error_info.locregs[], where the context is saved if the error is a fatal error.
QURT_error_info.global_regs	Contains the values of the global registers of Q6.
QURT_error_info.local_regs [QURT_error_info.status.hw_tnum]	Provides the CPU context if the error is a supervisor error.

B.4 Debugging a Nonfatal Error

Error	Description
QURT_error_info.user_errors	All user errors are logged here
QURT_error_info.user_errors.counter	Index to last logged error
QURT_error_info.user_errors.entry[0...counter]	Structure for logged error
QURT_error_info.user_ errors.entry[0...counter].error_tcb	TCB for the user error
QURT_error_info.user_ errors.entry[0...counter].error_tcb.error	Information about error; Cause, Cause2, Badva, and HWTID
QURT_error_info.user_ errors.entry[0...counter].error_code	((cause2 « 8) 'Logical OR' (cause)). See Table B.6.
QURT_error_info.user_ errors.entry[0...counter].hw_thread	Hardware thread ID for error.
QURT_error_info.user_ errors.entry[0...counter].pcycle	Pcycle for error.

B.5 Cause

Cause	Value	Description
QURT_EXCEPT_PRECISE	0x01	Precise exception occurred.
QURT_EXCEPT_NMI	0x02	NMI occurred.
QURT_EXCEPT_TLBMISS	0x03	TLBMISS RW occurred.
QURT_EXCEPT_RSD_VECTOR	0x04	Interrupt was raised on reserved vector, should never happen.
QURT_EXCEPT_ASSERT	0x05	Kernel assert.
QURT_EXCEPT_BADTRAP	0x06	Trap0 was called with unsupported num.
QURT_EXCEPT_UNDEF_TRAP1	0x07	Trap1 is not supported; using Trap1 causes this error.
QURT_EXCEPT_EXIT	0x08	Application called qurt_exit(), or called qurt_exception_raise_nonfatal(). Can call from C library.
QURT_EXCEPT_TLBMISS_X	0x0A	TLBMISS X (execution) occurred.
QURT_EXCEPT_STOPPED	0x0B	Running thread stopped due to fatal error on another hardware thread.
QURT_EXCEPT_FATAL_EXIT	0x0C	Call ended because of an internal error
QURT_EXCEPT_INVALID_INT	0x0D	Kernel received an invalid L1 interrupt.
QURT_EXCEPT_FLOATING_POINT	0x0E	Kernel received a floating point error.
QURT_EXCEPT_DBG_SINGLE_STEP	0x0F	Cause2 is not defined.
QURT_EXCEPT_TLBMISS_RW_ ISLAND	0x10	Read write miss in Island mode.
QURT_EXCEPT_TLBMISS_X_ISLAND	0x11	Execute miss in Island mode.
QURT_EXCEPT_SYNTHETIC_FAULT	0x12	Synthetic fault with user request that the kernel detected. See Table B.6.
QURT_EXCEPT_INVALID_ ISLAND_TRAP	0x13	Invalid trap in Island mode.
QURT_EXCEPT_UNDEF_TRAP0	0x14	Trap0 called with an unsupported number.
QURT_EXCEPT_PRECISE_DMA_ ERROR	0x28	Precise DMA error. Cause2 is DM4[15:8]. Badva is DM5 register.

B.6 Cause 2

Cause	Possible cause 2	Value	Cause2 description
QURT_EXCEPT_PRECISE	SSR[7:0]	–	–
QURT_EXCEPT_NMI	Not defined.	–	–
QURT_EXCEPT_TLBMISS	SSR[7:0]	–	–
QURT_EXCEPT_RSD_ VECTOR	Not defined.	–	–
QURT_EXCEPT_ASSERT	QURT_ABORT_FUTEX_ WAKE_MULTIPLE	0x01	Abort cause; futex wake multiple
	QURT_ABORT_WAIT_ WAKEUP_SINGLE_MODE	0X02	Abort cause; thread waits to wake up in Single-threaded mode
	QURT_ABORT_TCXO_ SHUTDOWN_NOEXIT	0x03	Abort cause; TCXO shutdown is called without exit
	QURT_ABORT_FUTEX_ ALLOC_QUEUE_FAIL	0x04	Abort cause; futex allocation queue fail
	QURT_ABORT_INVALID_ CALL_QURTK_WARM_INIT	0x05	Abort cause; invalid QURTK_warm_init() call in NONE CON-FIG_POWER_MGMT mode
	QURT_ABORT_THREAD_ SCHEDULE_SANITY	0x06	Abort cause; sanity schedule thread is not supposed to run on the current hardware thread
	QURT_ABORT_REMAP	0x07	Remap in the page table; the correct behavior is to always remove mapping
	QURT_ABORT_NOMAP	0x08	No mapping in the page table when removing a user mapping
	QURT_ABORT_OUT_ OF_SPACES	0x09	–
	QURT_ABORT_INVALID_ MEM_MAPPING_TYPE	0x0A	Invalid memory mapping type when creating qmemory
	QURT_ABORT_NOPOOL	0x0B	No pool available to attach
	QURT_ABORT_LIFO_ REMOVE_NON_EXIST_ ITEM	0x0C	Cannot allocate more futex waiting queue
	QURT_ABORT_ARG_ERROR	0x0D	–
	QURT_ABORT_ASSERT	0x0E	Assert abort
	QURT_ABORT_FATAL	0x0F	Fatal error that shall never happen
	QURT_ABORT_FUTEX_ RESUME_INVALID_QUEUE	0x10	Abort cause; invalid queue ID in futex resume

Cause	Possible cause 2	Value	Cause2 description
	QURT_ABORT_FUTEX_ WAIT_INVALID_QUEUE	0x11	Abort cause; invalid queue ID in futex wait
	QURT_ABORT_FUTEX_ RESUME_INVALID_FUTEX	0x12	Abort cause; invalid futex object in hashtable
	QURT_ABORT_NO_ ERHNDLR	0x13	No registered error handler
	QURT_ABORT_ERR_ REAPER	0x14	Exception in reaper thread
	QURT_ABORT_FREEZE_ UNKNOWN_CAUSE	0x15	Abort in thread freeze operation
	QURT_ABORT_FUTEX_ WAIT_WRITE_FAILURE	0x16	Unable to perform a necessary write operation to userland data during futex wait processing; most likely due to a DL Pager eviction.
	QURT_ABORT_ERR_ ISLAND_EXP_HANDLER	0x17	Exception in Island exception handler task.
	QURT_ABORT_L2_ TAG_DATA_CHECK_FAIL	0x18	Detected error in L2 Tag/Data during warm boot, The L2 Tag/Data check is done when CONFIG_DEBUG_L2 is enabled.
	QURT_ABORT_ERR_ SECURE_PROCESS	0x19	Abort error secure process.
	QURT_ABORT_ERR_EXP_ HANDLER	0x20	Either no exception handler or handler itself caused an exception.
	QURT_ABORT_ERR_NO_ PCB	0x21	Thread context PCB failed initialization, PCB was NULL.
	QURT_ABORT_NO_PHYS_ ADDR	0x22	Unable to find the physical address for the virtual address.
	QURT_ABORT_OUT_OF_ FASTINT_CONTEXTS	0x23	Fast interrupt contexts exhausted.
	QURT_ABORT_CLADE_ERR	0x24	Fatal error seen with CLADE interrupt.
	QURT_ABORT_ETM_ERR	0x25	Fatal error seen with ETM interrupt.
	QURT_ABORT_ECC_DED_ ASSERT	0x26	ECC two-bit DED error.
	QURT_ABORT_VTLB_ERR	0x27	Fatal error in the VTLB layer.
	QURT_ABORT_TLB_ENCODE_ DECODE_FAILURE	0x28	Failure during the TLB encode or decode operation.

Cause	Possible cause 2	Value	Cause2 description
	QURT_ABORT_VTLB_WALKOBJS_BOUND_FAILURE	0x29	Failure to look up entry in the page table.
QURT_EXCEPT_BADTRAP	0	–	–
QURT_EXCEPT_UNDEF_TRAP1	Not defined	–	–
QURT_EXCEPT_EXIT	Argument passed to qurt_exception_raise_nonfatal() and 0XFF	–	–
QURT_EXCEPT_TLBMISS_X	SSR[7:0]	–	–
QURT_EXCEPT_STOPPED	Not defined	–	–
QURT_EXCEPT_FATAL_EXIT	Not defined	–	–
QURT_EXCEPT_INVALID_INT	Not defined	–	–
QURT_EXCEPT_FLOATING_POINT	Not defined	–	–
QURT_EXCEPT_DBG_SINGLE_STEP	Not defined	–	–
QURT_EXCEPT_TLBMISS_RW_ISLAND	QURT_TLB_MISS_RW_MEM_READ	0x70	
	QURT_TLB_MISS_RW_MEM_WRITE	0x71	–
QURT_EXCEPT_TLBMISS_X_ISLAND	SSR[7:0]	–	–
QURT_EXCEPT_SYNTHETIC_FAULT	QURT_SYNTH_ERR	0X01	–
	QURT_SYNTH_INVALID_OP	0X02	–
	QURT_SYNTH_DATA_ALIGNMENT_FAULT *	0X03	–
	QURT_SYNTH_FUTEX_INUSE *	0X04	–
	QURT_SYNTH_FUTEX_BOGUS *	0X05	–
	QURT_SYNTH_FUTEX_ISLAND *	0X06	–
	QURT_SYNTH_FUTEX_DESTROYED *	0X07	–
	QURT_SYNTH_PRIVILEGE_ERR	0X08	–
QURT_EXCEPT_INVALID_ISLAND_TRAP	Trap number	–	–
QURT_EXCEPT_UNDEF_TRAP0	Trap number	–	

*** Badva is the object address for these cause2 codes.**

C References

C.1 Related Documents

Title	Number
Resources	
Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts. John Wiley and Sons, 2008.	ISBN No. 0470128720

C.2 Acronyms and Terms

Acronym or term	Definition
API	Application programming interface
Application	Category of user program (multimedia, modem firmware, modem software)
ATB	Advanced trace bus
BSP	Board support package
CDSP	Compute DSP
DED	Double error detection
DSP	Digital signal processor
DTB	Device tree blob
EBI	External bus interface (memory type)
ELF	Executable and linking format
ETM	Embedded trace macro
Edge-triggered	Interrupt triggered by a rising or falling transition on the interrupt request line
HVX	Hexagon Vector Extensions
Interrupt	Externally generated processor event that interrupts the normal flow of program control
IST	Interrupt service thread
L2VIC	Second-level vector interrupt controller
Level-triggered	Interrupt triggered by a high or low level on the interrupt request line
LPM	Low-power memory
LR	Link register
MMU	Memory management unit
NMI	Nonmaskable interrupt
PCB	Process control block
PDC	Peripheral DMA controller
PID	Process ID.
PMU	Performance monitor unit – Hexagon processor feature that measures code performance
Priority	User-defined thread attribute that prioritizes thread execution

Acronym or term	Definition
QDI	QuRT driver invocation – Set of facilities that support the implementation of device drivers in the QuRT system
QuRT	Real time operating system for the Hexagon processor
RTOS	Real-time operating system
SAC	Secure access control
SMI	Stack memory interface
SRM	System resource manager
SSR	Supervisor status register
STID	Software thread ID
TCB	Task control block
TCM	Tightly coupled memory
TCXO	Temperature compensated crystal oscillator
TLB	Translation lookaside buffer
TLS	Thread local storage
User program	Complete program that makes calls to the QuRT API to perform RTOS operations
UGP	User general pointer
USR	User status register
VA	Virtual address
VMA	Virtual memory area