Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ QuRT RTOS

User Guide for Hexagon SDK

80-VB419-178 A

December 7, 2017

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

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope

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, refer to ISBN 0470128720.

1.2 Conventions

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

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, submit a case to Qualcomm Technologies at https://support.cdmatech.com.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

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.

QuRT offers the following features:

- Low overhead (both in memory and processing)
- Simplicity of implementation
- Ease of porting standalone user programs to QuRT environment
- Ease of modification to accommodate specific target requirements

Note: This document describes information specific to QuRT version 02.4.14.

2.1 Basic Concepts

QuRT supports real-time priority-based preemptive multithreading:

- Multithreading means that 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, and the program can then create additional threads. The Hexagon processor can execute a fixed number of threads simultaneously – any additional threads must share the processor. QuRT handles all details of sharing.
- Priority-based means that each thread is assigned a priority level. The priority determines which thread has execution priority.
- Preemptive means that a thread can be preempted – i.e., have the processor taken away – when a higher-priority thread is ready to execute.
- Real-time means that the operating system is able to perform its operations within certain periods of time.

QuRT consists of the following items:

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

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

2.2 Features

QuRT offers the following features:

- Multithreading – A user program can create multiple threads which execute simultaneously.
- Processes – Enables programs and threads to execute in separate protected address spaces for improved system security and stability.
- Mutexes – Synchronize threads to ensure mutually exclusive access to shared resources.
- Signals – Synchronize threads on sets of mutex-like signals.
- Semaphores – Synchronize threads to ensure limited access to shared resources.
- Barriers – Synchronize threads to meet at a specific point in a user program.
- Condition variables – Synchronize threads based on the value of a data item.
- Pipes – Supports synchronized data exchange between threads.
- Timers – Threads can schedule actions to occur at specific times or intervals.
- Interrupt handling – Register threads to serve as interrupt handlers.
- Thread local storage – Allocates global storage which is private to specific threads.
- Exception handling – Supports exception handling for fatal and non-fatal exceptions.
- Memory management – User programs can dynamically manage their memory space.
- Profiling – Record cycle counts (both running and idle) for specific threads.
- Performance monitor – Supports code performance measurement during user program execution.
- Call tracing – Supports debug macros for tracing function calls and returns.

2.3 Processor Versions

QuRT supports Hexagon processor versions V5, V55, V56, and V60.

3 Using QuRT

3.1 User Programs

A QuRT system contains one or more user programs. Each user program is a complete program which uses the QuRT API (see Section 3.3) to access the QuRT services. When a user program is started it is assigned a single thread – to create additional threads, the program uses the QuRT thread services.

A user program typically 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 the following items:

- 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 the user program is built (Section 3.2).

Figure 3-1 shows a functional diagram of a user program image.

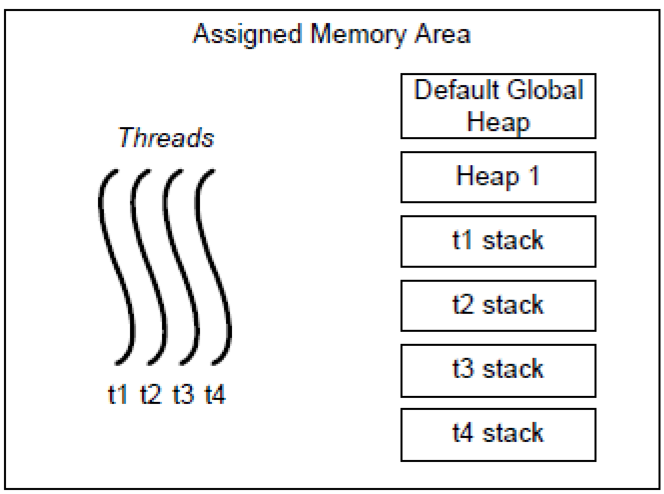

Figure 3-1 User program image

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.

3.2 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 stand-alone C/C++ program.

All 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. This file is then 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 be executed in two ways:

- 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).

3.3 API

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

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

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

For more information on QuRT API functions, see Section 3.7.

3.4 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 for managing objects:

- The init/destroy operations (shown in the preceding example) are used for objects that are stored wholly in memory allocated by the user program.
- The create/delete operations are used for objects that are stored partly in memory allocated automatically by the RTOS kernel.

Pipe objects support both operation pairs: init/destroy are used when the pipe buffer is user-allocated, while create/delete are used when the pipe buffer is automatically allocated by the kernel as part of initializing a pipe object.

Timer objects support only create/delete for object management. All other QuRT objects support only init/destroy for object management.

In addition to object management, most objects define additional operations which perform services associated with that object (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.

QuRT objects should be treated as having opaque types. They should be accessed only through QuRT functions.

3.5 Non-blocking and Cancellable Operations

QuRT defines several operations, which are non-blocking 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 non-blocking operations are identified by the prefix "try_" in their operation names, while the cancellable operations use the suffix "_cancellable".

Non-blocking 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 thread's user process is killed, or if the thread must finish its current QDI invocation and return to user space.

When an operation is canceled, the calling thread must assume that the operation will never be completed: the caller must stop waiting for the specified resource or event, and instead assume that the event will never occur or the resource will never become available.

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

If a driver detects a cancelled 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.

3.6 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 are able to access memory addresses above 4 GB.

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

3.7 QuRT Services

The following chapters describe the QuRT APIs.

- Threads
- Processes
- Mutexes
- Recursive Mutexes
- Priority Inheritance Mutexes
- Signals
- Any-signals
- All-signals
- Semaphores
- Barriers
- Condition Variables
- Pipes
- Timers
- System Clock
- Interrupts
- Thread Local Storage
- Exception Handling
- Memory Allocation
- Memory Management
- System Environment
- Profiling
- Performance Monitor
- Error Results
- Function Tracing
- QuRT Callbacks
- Predefined Symbols

4 Threads

Multitasking allows multiple instruction sequences in a user program to execute in parallel. Each sequence of instructions in a running user program is called a thread.

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

- Create thread – Create a thread and make it executable.
- Resume thread – Awaken the specified thread.
- Exit thread – Stop the current thread and destroy it.
- Join thread – Suspend the current thread until the specified thread stops.
- Get current thread – Return a reference to the current thread.

Once started, a thread exists in one of four states. Table 4-1 lists the states.

Table 4-1 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.

The kernel is responsible for switching threads between these states. It uses a scheduler to determine which threads to run – the scheduler always selects the highest-priority ready threads.

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. During system startup the scheduler selects the highest-priority threads for execution and changes their thread state to Running.

The action of suspending one thread and resuming another is called a context switch.

Figure 4-1 shows the events that can cause the kernel to perform context switches.

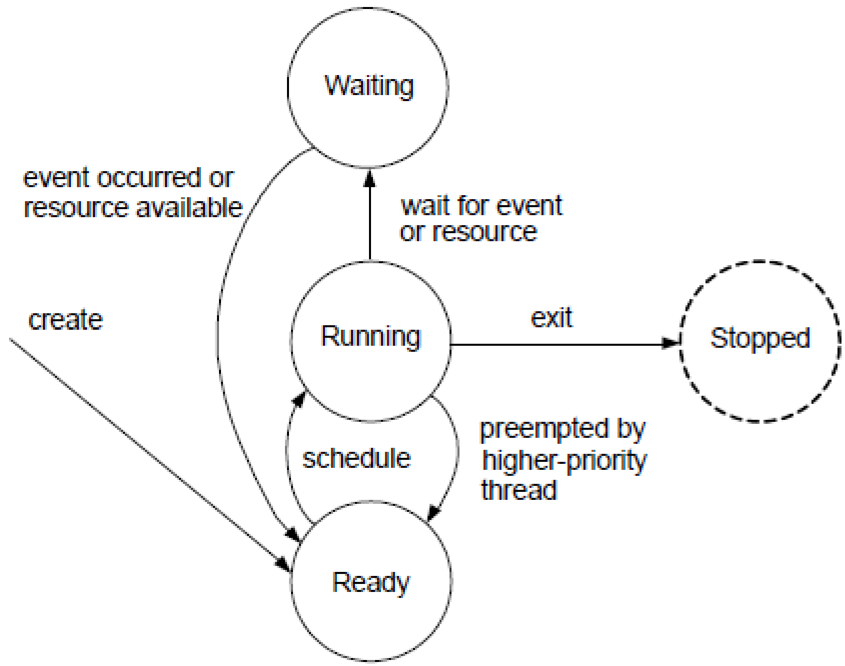

Figure 4-1 Thread state transitions

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

Thread priorities

The priority of a thread determines how often the thread executes relative to the other threads in the system: 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.

Threads are assigned priorities when they are first created; however, in some cases a user program system may need to adjust the priority of a thread after it has been created. For instance, to prevent priority inversion a thread may need to raise its own priority or the priority of another thread.

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.

Note: QuRT can be configured to have different priority ranges (Section 3.2).

Thread attributes

Threads have the following attributes:

- Name – Character string identifier used to identify the thread.
- TCB partition – Memory used for allocating thread control blocks (TCBs).
- Affinity – Hardware threads used by the thread.
- Priority – Thread execution priority.
- Bus priority – Internal bus priority state.
- Timetest ID – Numeric trace identifier used during hardware debugging.
- Stack size – Size (in bytes) of the memory area used for the thread call stack.
- Stack address – Base address of the memory area used for the thread call stack.
- Entry point – Function that the thread executes.
- Argument – Pointer passed to the thread function when it is executed.
- Signal – Signal object created by QuRT for each thread.
- Cache partition – Memory allocated for threads.

The thread name and timetest identifier are used to identify threads during debugging or profiling. These attributes differ from the kernel-generated thread identifiers used to specify threads in the API thread operations.

The thread entry point is the function executed by the thread when it is started. The function is defined in the user program, and must accept a single void pointer as a function parameter.

The thread argument is a pointer that is passed to the thread function when the thread is started. It allows a single function to be written so it can be executed by multiple threads.

The thread stack address and 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.

The thread priority determines the execution priority of the thread.

The thread affinity specifies which Hexagon processor hardware threads the thread can execute on.

The thread TCB partition specifies the maximum number of threads that have their thread control blocks allocated in TCM/LPM instead of regular memory.

The thread signal is a signal object which is created by QuRT for each thread.

The thread cache partition allocates memory for the current thread for the L1 I cache, l1 D cache, and L2 cache.

Note: Threads are specified by the thread identifier returned by the thread create operation. This identifier is distinct from the thread name or timetest ID.

Setting thread attributes

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.

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: Two thread attributes – the thread identifier and thread signal – are read-only attributes set by the kernel.
The timetest ID attribute is stored in a Hexagon processor register.

Functions

Thread services are accessed with the following QuRT functions.

- qurt_thread_attr_get()
- qurt_thread_attr_init()
- qurt_thread_attr_set_affinity()
- qurt_thread_attr_set_bus_priority()
- 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_tcb_partition()
- qurt_thread_attr_set_timetest_id()
- qurt_thread_create()
- qurt_thread_exit()
- qurt_thread_get_anysignal()
- 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_set_timetest_id()
- qurt_thread_get_tls_base()
- Data Types
- Constants and Macros

4.1 qurt_thread_attr_get()

4.1.1 Function Documentation

4.1.1.1 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.

4.2 qurt_thread_attr_init()

4.2.1 Function Documentation

4.2.1.1 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, the individual attributes in the structure can be explicitly set using the thread attribute operations.

The default attribute values set by the initialize operation are the following:

- Name – Null string
- TCB partition – QURT_THREAD_ATTR_TCB_PARTITION_DEFAULT
- Affinity – QURT_THREAD_ATTR_AFFINITY_DEFAULT
- Priority – QURT_THREAD_ATTR_PRIORITY_DEFAULT
- ASID – QURT_THREAD_ATTR_ASID_DEFAULT
- Bus priority – QURT_THREAD_ATTR_BUS_PRIO_DEFAULT
- Timetest ID – QURT_THREAD_ATTR_TIMETEST_ID_DEFAULT
- stack_size – 0
- stack_addr – 0
- detach_state – QURT_THREAD_ATTR_CREATE_DETACHED

Associated data types

qurt_thread_attr_t

Parameters

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

Returns

None.

Dependencies

None.

4.3 qurt_thread_attr_set_affinity()

4.3.1 Function Documentation

4.3.1.1 static void qurt_thread_attr_set_affinity (qurt_thread_attr_t * *attr*, unsigned char *affinity*)

Specifies the Hexagon processor hardware threads that a QuRT thread can execute on.

This function sets the thread affinity attribute. The affinity value specifies a bitmask value identifying the hardware threads to be used.

Bits 0 through 5 in the 8-bit mask value specify hardware threads 0 through 5 respectively. If a bit is set to 1, the thread is eligible to run on the corresponding hardware thread.

Mask bit values are specified using the predefined bitmask symbols QURT_THREAD_CFG_* (Section 4.26.1.2). These symbols can be ORed together to specify more than one hardware thread, or the symbol QURT_THREAD_CFG_BITMASK_ALL can be used to specify all the threads.

Note: QURT_THREAD_CFG_BITMASK_ALL is defined to specify the proper set of hardware threads regardless of the Hexagon processor version (since the versions support different numbers of hardware threads).

Associated data types

qurt_thread_attr_t

Parameters

in,out	*attr*	Pointer to the thread attribute structure.
in	*affinity*	Bitmask indicating hardware threads used.

Returns

None.

Dependencies

None.

4.4 qurt_thread_attr_set_bus_priority()

4.4.1 Function Documentation

4.4.1.1 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.

4.5 qurt_thread_attr_set_name()

4.5.1 Function Documentation

4.5.1.1 static void qurt_thread_attr_set_name (qurt_thread_attr_t * *attr,* char * *name*)

Sets the thread name attribute.

This function specifies the name to be used by a thread. Thread names are used to identify a thread during debugging or profiling.

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.

4.6 qurt_thread_attr_set_priority()

4.6.1 Function Documentation

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

Sets the thread priority to be assigned to a thread. Thread priorities are specified as numeric values in the range 1 to 255, with 1 representing the highest priority.

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.

4.7 qurt_thread_attr_set_stack_addr()

4.7.1 Function Documentation

4.7.1.1 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 be used 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 4.8.1.1) 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 is created by the thread.

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.

4.8 qurt_thread_attr_set_stack_size()

4.8.1 Function Documentation

4.8.1.1 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 be used for a call stack of a thread.

The thread stack address (Section 4.7.1.1) 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.

4.9 qurt_thread_attr_set_tcb_partition()

4.9.1 Function Documentation

4.9.1.1 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 thread control block (TCB) of a thread is allocated. TCBs can be allocated 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.

4.10 qurt_thread_attr_set_timetest_id()

4.10.1 Function Documentation

4.10.1.1 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 be used 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.

4.11 qurt_thread_create()

4.11.1 Function Documentation

4.11.1.1 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.

Note: This function fails (with an error result) if the set of hardware threads specified in the thread attributes is invalid for the target processor version.

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.

4.12 qurt_thread_exit()

4.12.1 Function Documentation

4.12.1.1 void qurt_thread_exit (int *status*)

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

Any thread that has been suspended on the current thread (by performing a thread join – Section 4.19.1.1) is awakened and passed a user-defined status value indicating the status of the stopped thread.

Note: Exit must be called in the context of the thread to be stopped.

Parameters

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

Returns

None.

Dependencies

None.

4.13 qurt_thread_get_anysignal()

4.13.1 Function Documentation

4.13.1.1 unsigned int qurt_thread_get_anysignal (void)

Gets the signal of the current thread. Returns the RTOS-assigned signal of the current thread.

QuRT assigns every thread a signal to support communication between threads.

Returns

Signal object address – Any-signal object assigned to the current thread.

Dependencies

None.

4.14 qurt_thread_get_id()

4.14.1 Function Documentation

4.14.1.1 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.

4.15 qurt_thread_get_l2cache_partition()

4.15.1 Function Documentation

4.15.1.1 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 data type qurt_cache_partition_t.

Dependencies

None.

4.16 qurt_thread_get_name()

4.16.1 Function Documentation

4.16.1.1 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 (Section 4). They are used to 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.

4.17 qurt_thread_get_priority()

4.17.1 Function Documentation

4.17.1.1 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 0 through 255, with lower values representing higher priorities. 0 represents the highest possible thread priority.

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.
0 through 255 – Thread priority value.

Dependencies

None.

4.18 qurt_thread_get_timetest_id()

4.18.1 Function Documentation

4.18.1.1 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.

4.19 qurt_thread_join()

4.19.1 Function Documentation

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

Waits for a specified thread to finish. The specified thread should be another thread within the same process. The caller thread is suspended until the specified thread exits. When this happens, 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 happens, QuRT generates an exception (see Section 20).

Parameters

in	*tid*	Thread identifier.
out	*status*	Destination variable for thread exit status. Returns an application-defined value indicating 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.

4.20 qurt_thread_resume()

4.20.1 Function Documentation

4.20.1.1 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.

4.21 qurt_thread_set_cache_partition()

4.21.1 Function Documentation

4.21.1.1 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 type qurt_cache_partition_t to select the cache partition of the current thread for the L1 I cache, L1 D cache, and L2 cache.

Associated data types

qurt_cache_partition_t

Parameters

in	*l1_icache*	L1 I cache partition.
in	*l1_dcache*	L1 D cache partition.
in	*l2_cache*	L2 cache partition.

Returns

None.

Dependencies

None.

4.22 qurt_thread_set_priority()

4.22.1 Function Documentation

4.22.1.1 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 0 through 255, with lower values representing higher priorities. 0 represents the highest possible thread priority.

Note: QuRT can be configured to have different priority ranges. For more information see Section 3.2.

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.

4.23 qurt_thread_set_timetest_id()

4.23.1 Function Documentation

4.23.1.1 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.

4.24 qurt_thread_get_tls_base()

rest_dist

4.24.1 Function Documentation

4.24.1.1 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.

4.25 Data Types

This section describes data types for thread services.

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.

4.25.1 Data Structure Documentation

4.25.1.1 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 (in other words, TCM).
unsigned char	affinity	Hardware bitmask indicating the threads it can run on.
unsigned short	priority	Thread priority.
unsigned char	asid	Address space ID.
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.
unsigned int	is_autostack	Autostack.
unsigned int	frame_size	Frame size.

4.25.1.2 struct qurt_tls_info

Dynamic TLS attributes

Data fields

Type	Parameter	Description
unsigned int	module_id	Module ID for 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.

4.25.2 Typedef Documentation

4.25.2.1 typedef unsigned int qurt_thread_t

Thread ID type

4.25.3 Enumeration Type Documentation

4.25.3.1 enum qurt_cache_partition_t

Enumerator:

CCCC_PARTITION Use the CCCC page attribute bits to determine the main or auxiliary partition.
MAIN_PARTITION Use the main partition.
AUX_PARTITION Use the auxiliary partition.
MINIMUM_PARTITION Use the minimum. Allocates the least amount of cache (no-allocate policy possible) for this thread.

4.26 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.

4.26.1 Define Documentation

4.26.1.1 #define QURT_MAX_HTHREAD_LIMIT 6

The limit on the maximum number of hardware threads supported by QuRT for any Hexagon version. This definition can be used to define arrays, and so on, in target independent code.

4.26.1.2 #define QURT_THREAD_CFG_BITMASK_ALL 0x000000ff

Select all the hardware threads.

4.26.1.3 #define QURT_THREAD_BUS_PRIO_DISABLED 0

Thread internal bus priority disabled

4.26.1.4 #define QURT_THREAD_BUS_PRIO_ENABLED 1

Thread internal bus priority enabled

4.26.1.5 #define QURT_THREAD_ATTR_CREATE_DETACHED 0

4.26.1.6 #define QURT_THREAD_ATTR_TCB_PARTITION_DEFAULT QURT_THREAD-_ATTR_TCB_PARTITION_RAM

Backward compatibility.

4.26.1.7 #define QURT_THREAD_ATTR_PRIORITY_DEFAULT 255

Priority.

4.26.1.8 #define QURT_THREAD_ATTR_ASID_DEFAULT 0

ASID.

4.26.1.9 #define QURT_THREAD_ATTR_AFFINITY_DEFAULT (-1)

Affinity.

4.26.1.10 #define QURT_THREAD_ATTR_BUS_PRIO_DEFAULT 255

Bus priority.

4.26.1.11 #define QURT_THREAD_ATTR_TIMETEST_ID_DEFAULT (-2)

Timetest ID.

5 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. Process objects support the following operations:

- Create process – Creates a process with the specified attributes.
- Get current process – Returns a reference to the current process.
- Get process command line – Gets the command line string associated with the current process.

When a process is created, it automatically starts running the code in the specified executable file. A newly created process is assigned an identifier value which identifies the process.

The get current process operation returns the process identifier for the current process.

The get process command line operation gets the command line string associated with the current process.

Process attributes
Processes have the following attributes:

- Name - Character string identifier used to identify a process.
- Flags - Bit array used to specify properties of a process.

The process name specifies a process object which is already loaded in memory as part of the QuRT system.

The processor flags 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.

Functions

Process services are accessed with the following QuRT functions:

- qurt_process_attr_init()
- qurt_process_attr_set_executable()
- qurt_process_attr_set_flags()
- qurt_process_cmdline_get()
- qurt_process_create()
- qurt_process_get_id()
- Data Types

5.1 qurt_process_attr_init()

5.1.1 Function Documentation

5.1.1.1 static void qurt_process_attr_init (qurt_process_attr_t * *attr*)

Initializes the structure that is used to set 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 5-1 lists the default attribute values set by the initialize operation.

Table 5-1 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.

5.2 qurt_process_attr_set_executable()

5.2.1 Function Documentation

5.2.1.1 static void qurt_process_attr_set_executable (qurt_process_attr_t * *attr*, char * *name*)

Sets the process name in the specified process attribute structure.

Process names are used to 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.

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.

5.3 qurt_process_attr_set_flags()

5.3.1 Function Documentation

5.3.1.1 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_SUSPEND_ON_STARTUP suspends the process after creating it.

Returns

Dependencies

None.

5.4 qurt_process_cmdline_get()

5.4.1 Function Documentation

5.4.1.1 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 is running in a simulation environment.

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

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 buf.

Returns

None.

Dependencies

None.

5.5 qurt_process_create()

5.5.1 Function Documentation

5.5.1.1 int qurt_process_create (qurt_process_attr_t * *attr*)

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

The process executes the code in the specified executable ELF file.

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

None.

Dependencies

None.

5.6 qurt_process_get_id()

5.6.1 Function Documentation

5.6.1.1 int qurt_process_get_id (void)

Returns the process identifier for the current thread.

Returns

None.

Dependencies

Process identifier for the current thread..

5.7 Data Types

This section describes data types for processes.

5.7.1 Data Structure Documentation

5.7.1.1 struct qurt_process_attr_t

QuRT process type.

6 Mutexes

rest_dist

Threads use mutexes to synchronize their execution to ensure mutually exclusive access to shared resources. Mutexes are shared objects which support the following operations:

- Init mutex – Initialize mutex.
- Destroy mutex – Destroy the specified mutex.
- Lock mutex – Request access to a shared resource.
- Unlock mutex – Release access to a shared resource.
- Try lock mutex –Request access to a shared resource (does not suspend).

If a thread performs a lock operation on a mutex that is not being used, 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 being used 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 may occur.

Figure 6-1 shows an example of using mutexes.

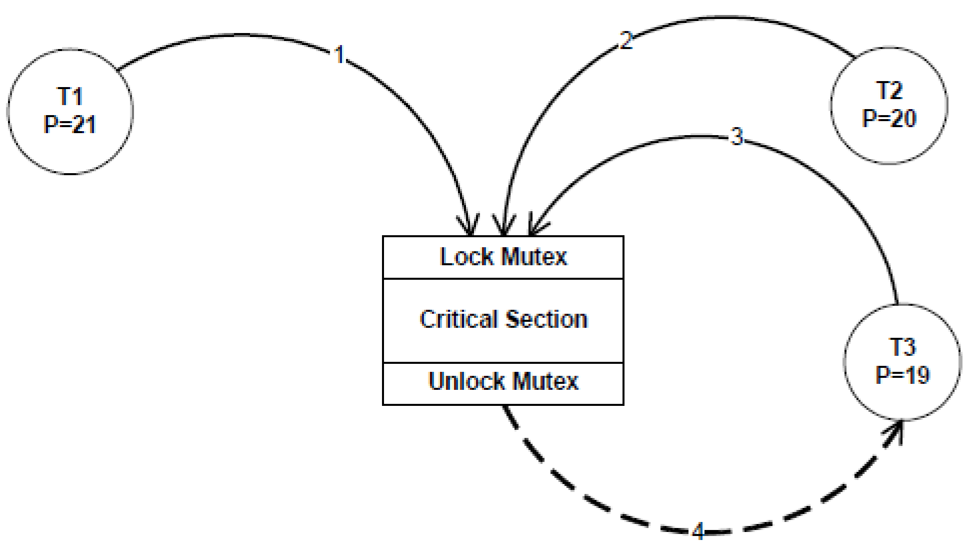

Figure 6-1 Mutex example

In figure 6-1 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.

Functions
Mutex services are accessed with the following QuRT functions.

- qurt_mutex_destroy()
- qurt_mutex_init()
- qurt_mutex_lock()
- qurt_mutex_try_lock()
- qurt_mutex_unlock()
- Data Types
- Constants and Macros

6.1 qurt_mutex_destroy()

6.1.1 Function Documentation

6.1.1.1 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 happens, 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.

6.2 qurt_mutex_init()

6.2.1 Function Documentation

6.2.1.1 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.

6.3 qurt_mutex_lock()

6.3.1 Function Documentation

6.3.1.1 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 being used, 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 being used 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. This can be avoided by using recursive mutexes (Section 7).

Associated data types

qurt_mutex_t

Parameters

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

Returns

None.

Dependencies

None.

6.4 qurt_mutex_try_lock()

6.4.1 Function Documentation

6.4.1.1 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 being used, 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 – Fail.

Dependencies

None.

6.5 qurt_mutex_unlock()

6.5.1 Function Documentation

6.5.1.1 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.

6.6 Data Types

This section describes data types for mutex services.

Mutexes are represented in QuRT as objects of type qurt_mutex_t.

6.6.1 Data Structure Documentation

6.6.1.1 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.

6.7 Constants and Macros

This section describes constants and macros for mutex services.

6.7.1 Define Documentation

6.7.1.1 #define MUTEX_MAGIC 0xfe

6.7.1.2 #define QURT_MUTEX_INIT {{MUTEX_MAGIC, 0, QURTK_FUTEX_FREE_MAGIC,0}}

Suitable as an initializer for a variable of type qurt_mutex_t.

7 Recursive Mutexes

QuRT supports a variant of mutexes known as recursive mutexes. Recursive mutexes are shared objects which support the following operations:

- Init recursive mutex – Initialize the recursive mutex.
- Destroy recursive mutex – Destroy the specified recursive mutex.
- Lock recursive mutex – Request access to a shared resource.
- Unlock recursive mutex – Release access to a shared resource.
- Try lock recursive mutex – Request access to a shared resource (without suspend).

Recursive mutexes are functionally equivalent to regular mutexes (Section 6), 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 being used by another thread, the thread is suspended.
- If a thread performs a lock on a recursive mutex that is already being used by itself, the operation is treated as a nested lock and the thread continues executing as if the mutex were 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 were unlocked.

Functions
Recursive mutex services are accessed with the following QuRT functions.

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

7.1 qurt_rmutex_destroy()

7.1.1 Function Documentation

7.1.1.1 void qurt_rmutex_destroy (qurt_mutex_t * *lock*)

Destroys the specified recursive mutex.

Note: Recursive mutexes must be destroyed when they are no longer in use. Failure to do this causes resource leaks in the QuRT kernel.

Recursive mutexes must not be destroyed while they are still in use. If this happens 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.

7.2 qurt_rmutex_init()

7.2.1 Function Documentation

7.2.1.1 void qurt_rmutex_init (qurt_mutex_t * *lock*)

Initializes a recursive mutex object. The recursive mutex is initially unlocked.

Note: Each rmutex-based object has one or more kernel resources associated with it; to prevent resource leaks, be sure to call qurt_rmutex_destroy() when this object is not used anymore

Associated data types

qurt_mutex_t

Parameters

out	*lock*	Pointer to the recursive mutex object.

Returns

None.

Dependencies

None.

7.3 qurt_rmutex_lock()

7.3.1 Function Documentation

7.3.1.1 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 being used, 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 being used 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.

Associated data types

qurt_mutex_t

Parameters

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

Returns

None.

Dependencies

None.

7.4 qurt_rmutex_try_lock()

7.4.1 Function Documentation

7.4.1.1 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 being used, 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 is already being used by another thread, qurt_rmutex_try_lock immediately returns with a nonzero result value.

Note: If a thread performs a try_lock operation on a mutex that it has already locked, qurt_mutex_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.

7.5 qurt_rmutex_unlock()

7.5.1 Function Documentation

7.5.1.1 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 is awakened. 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.

8 Priority Inheritance Mutexes

QuRT supports a variant of recursive mutexes known as priority inheritance mutexes. Priority inheritance mutexes are shared objects which support the following operations:

- Init priority inheritance mutex – Initialize the priority inheritance mutex.
- Destroy priority inheritance mutex – Destroy the specified priority inheritance mutex.
- Lock priority inheritance mutex – Request access to a shared resource.
- Unlock priority inheritance mutex – Release access to a shared resource.
- Try lock priority inheritance mutex – Request access to a shared resource (without suspend).

Priority inheritance mutexes are functionally equivalent to recursive mutexes (Section 7), 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 4) becomes suspended on the mutex, then the thread with the lock acquires the higher priority of the suspended thread.
- If multiple threads are suspended on a priority inheritance mutex, then 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 only 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 were unlocked.

Functions
Priority inheritance mutex services are accessed with the following QuRT functions.

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

8.1 qurt_pimutex_init()

8.1.1 Function Documentation

8.1.1.1 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.

8.2 qurt_pimutex_destroy()

8.2.1 Function Documentation

8.2.1.1 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 happens, 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.

8.3 qurt_pimutex_lock

8.3.1 Function Documentation

8.3.1.1 void qurt_pimutex_lock (qurt_mutex_t * *lock*)

If a thread performs a lock operation on a mutex that is not being used, 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 being used 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 by itself. 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 are competing 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.

8.4 qurt_pimutex_try_lock()

8.4.1 Function Documentation

8.4.1.1 int qurt_pimutex_try_lock (qurt_mutex_t * *lock*)

Attempts to lock the specified priority inheritance mutex.

If a thread performs a try_lock operation on a priority inheritance mutex that is not being used, 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 being used 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.

8.5 qurt_pimutex_unlock()

8.5.1 Function Documentation

8.5.1.1 void qurt_pimutex_unlock (qurt_mutex_t * *lock*)

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.

9 Signals

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

- Init signal - Initialize a signal object.
- Destroy signal - Destroy the specified signal object.
- Wait on any signal - Suspend the current thread until one of the specified signals is set.
- Wait on all signals - Suspend the current thread until all of the specified signals are set.
- Wait on signals cancellable - Suspends the current thread until either the specified signals are set or the wait operation is cancelled.
- Set signal - Set signals in the specified signal object.
- Get signal - Return the current value of the specified signal object.
- Clear signal - Clear signals in the specified signal object.

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, then 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, then the thread is awakened.

A 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 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.

Functions

Signal services are accessed with the following QuRT functions:

- 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_64_init()
- qurt_signal_64_destroy()
- qurt_signal_64_wait()
- qurt_signal_64_set()
- qurt_signal_64_get()
- qurt_signal_64_clear()
- Data Types

9.1 qurt_signal_clear()

9.1.1 Function Documentation

9.1.1.1 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 is to be cleared, and 0 indicates that it is not to be cleared.

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, which identifies the individual signals to be cleared in the signal object.

Returns

None.

Dependencies

None.

9.2 qurt_signal_destroy()

9.2.1 Function Documentation

9.2.1.1 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 happens 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.

9.3 qurt_signal_get()

9.3.1 Function Documentation

9.3.1.1 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.

9.4 qurt_signal_init()

9.4.1 Function Documentation

9.4.1.1 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.

9.5 qurt_signal_set()

9.5.1 Function Documentation

9.5.1.1 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 that a signal is to be set, and 0 indicates that it is not to be set.

Associated data types

qurt_signal_t

Parameters

in	*signal*	Pointer to the signal object to be modified.
in	*mask*	Mask value, which identifies the individual signals to be set in the signal object.

Returns

None.

Dependencies

None.

9.6 qurt_signal_wait()

9.6.1 Function Documentation

9.6.1.1 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 that a signal is to be waited on, and 0 indicates that it is not to be waited on.

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 once 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, which identifies the individual signals in the signal object to be waited on.
in	*attribute*	Indicates whether the thread waits for any of the signals to be set, or for all of them to be 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.

9.7 qurt_signal_wait_all()

9.7.1 Function Documentation

9.7.1.1 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 that a signal is to be waited on, and 0 indicates that it is not to be waited on.

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, which identifies the individual signals in the signal object to be waited on.

Returns

A 32-bit word with current signals.

Dependencies

None.

9.8 qurt_signal_wait_any()

9.8.1 Function Documentation

9.8.1.1 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 that a signal is to be waited on, and 0 indicates that it is not to be waited on.

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, which identifies the individual signals in the signal object to be waited on.

Returns

A 32-bit word with current signals.

Dependencies

None.

9.9 qurt_signal_wait_cancellable()

9.9.1 Function Documentation

9.9.1.1 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 is to be waited on, and 0 indicates that it is not to be waited on.

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 should assume that the signal is never going to be set.

Associated data types

qurt_signal_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 be waited on.
in	*attribute*	Indicates whether the thread waits for any of the signals to be set, or for all of them to be 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.

9.10 qurt_signal_64_init()

9.10.1 Function Documentation

9.10.1.1 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.

9.11 qurt_signal_64_destroy()

9.11.1 Function Documentation

9.11.1.1 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 happens 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.

9.12 qurt_signal_64_wait()

9.12.1 Function Documentation

9.12.1.1 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 is to be waited on, and 0 indicates that it is not to be waited on.

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 be waited on.
in	*attribute*	Indicates whether the thread waits for any of the signals to be set, or for all of them to be 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.

9.13 qurt_signal_64_set()

9.13.1 Function Documentation

9.13.1.1 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 is to be set, and 0 indicates that it is not to be set.

Associated data types

qurt_signal_64_t

Parameters

in	*signal*	Pointer to the signal object to be modified.
in	*mask*	Mask value, which identifies the individual signals to be set in the signal object.

Returns

None.

Dependencies

None.

9.14 qurt_signal_64_get()

9.14.1 Function Documentation

9.14.1.1 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.

9.15 qurt_signal_64_clear()

9.15.1 Function Documentation

9.15.1.1 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 is to be cleared, and 0 indicates that it is not to be cleared.

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.

9.16 Data Types

This section describes data types for signal services.

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

9.16.1 Define Documentation

9.16.1.1 #define QURT_SIGNAL_ATTR_WAIT_ANY 0x00000000

Wait any.

9.16.1.2 #define QURT_SIGNAL_ATTR_WAIT_ALL 0x00000001

Wait all.

9.16.2 Data Structure Documentation

9.16.2.1 union qurt_signal_t

QuRT signal type.

9.16.2.2 struct qurt_signal_64_t

QuRT 64-bit signal type.

10 Any-signals

Threads use any-signals to synchronize their execution based on the occurrence of any one of a number of internal events. Any-signals are stored in shared objects which support the following operations:

- Init any-signal – Initialize the any-signal object.
- Destroy any-signal – Destroy the specified any-signal object.
- Wait any-signal – Suspend the current thread until one of the specified signals is set.
- Set any-signal – Set signals in the specified any-signal object.
- Get any-signal – Return the current value of the specified any-signal object.
- Clear any-signal – Clear signals in the specified any-signal object.

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

Functions
Any-signal services are accessed with the following QuRT functions.

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

10.1 qurt_anysignal_clear()

10.1.1 Function Documentation

10.1.1.1 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 is to be cleared, and 0 that it is not to be cleared.

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, which identifies the individual signals to be cleared in the any-signal object.

Returns

Bitmask – Old signal values (before clear).

Dependencies

None.

10.2 qurt_anysignal_destroy()

10.2.1 Function Documentation

10.2.1.1 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 happens 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.

10.3 qurt_anysignal_get()

10.3.1 Function Documentation

10.3.1.1 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.

10.4 qurt_anysignal_init()

10.4.1 Function Documentation

10.4.1.1 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.

10.5 qurt_anysignal_set()

10.5.1 Function Documentation

10.5.1.1 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 is to be set, and 0 that it is not to be set.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Pointer to the any-signal object to be modified.
in	*mask*	Signal mask value, which identifies the individual signals to be set in the any-signal object.

Returns

Bitmask of old signal values (before set).

Dependencies

None.

10.6 qurt_anysignal_wait()

10.6.1 Function Documentation

10.6.1.1 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 is to be waited on, and 0 that it is not to be waited on. 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 may 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 be waited on.

Returns

Bitmask of current signal values

Dependencies

None.

10.7 Data Types

This section describes data types for any-signal services.

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

10.7.1 Typedef Documentation

10.7.1.1 typedef qurt_signal_t qurt_anysignal_t

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

11 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 which support the following operations:

- Init all-signal – Initialize the all-signal object.
- Destroy all-signal – Destroy the specified all-signal object.
- Wait all-signal – Suspend the current thread until all of the specified signals are set.
- Set all-signal – Set signals in the specified all-signal object.
- Get all-signal – Return the current value of the specified all-signal object.

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, then the thread is awakened. If the awakened thread has higher priority than the current thread, a context switch may 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.

Functions
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

11.1 qurt_allsignal_destroy()

11.1.1 Function Documentation

11.1.1.1 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 happens, 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.

11.2 qurt_allsignal_get()

11.2.1 Function Documentation

11.2.1.1 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.

11.3 qurt_allsignal_init()

11.3.1 Function Documentation

11.3.1.1 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.

11.4 qurt_allsignal_set()

11.4.1 Function Documentation

11.4.1.1 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 is to be set, and 0 that it is not to be set.

Associated data types

qurt_allsignal_t

Parameters

in	*signal*	Pointer to the all-signal object to be modified.
in	*mask*	Signal mask value, which identifies the individual signals to be set in the all-signal object.

Returns

None.

Dependencies

None.

11.5 qurt_allsignal_wait()

11.5.1 Function Documentation

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

Waits on the all-signal object.n 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 is to 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 may 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 be waited on.

Returns

None.

Dependencies

None.

11.6 Data Types

This section describes data types for all-signal services.

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

11.6.1 Data Structure Documentation

11.6.1.1 union qurt_allsignal_t

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

12 Semaphores

Threads use semaphores to synchronize their access to shared resources. Semaphores are shared objects which support the following operations:

- Init semaphore – Initialize the semaphore.
- Init semaphore with value – Initialize the semaphore with the specified value.
- Destroy semaphore – Destroy the specified semaphore.
- Down semaphore – Request access to a shared resource.
- Up semaphore – Release access to a shared resource.
- Add semaphore – Release access (by specified value).
- Try down semaphore – Request access to a shared resource (without suspend).
- Get semaphore value – Return the count value of the specified semaphore.

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.

Down
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.

Up
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, then 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 may occur.

The add operation is similar to up, but can increment the semaphore count value by an amount greater than 1. 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.

Functions

Semaphore services are accessed with the following QuRT functions.

- 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()
- Data Types

12.1 qurt_sem_add()

12.1.1 Function Documentation

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

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

When a thread performs an add operation on a semaphore, the semaphore count value is incremented by the specified value. 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, then 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 may 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.

12.2 qurt_sem_destroy()

12.2.1 Function Documentation

12.2.1.1 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 happens, 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.

12.3 qurt_sem_down()

12.3.1 Function Documentation

12.3.1.1 int qurt_sem_down (qurt_sem_t * *sem*)

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.

Associated data types

qurt_sem_t

Parameters

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

Returns

Unused integer value.

Dependencies

None.

12.4 qurt_sem_get_val()

12.4.1 Function Documentation

12.4.1.1 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.

12.5 qurt_sem_init()

12.5.1 Function Documentation

12.5.1.1 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.

12.6 qurt_sem_init_val()

12.6.1 Function Documentation

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

Initializes a semaphore object.

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.

12.7 qurt_sem_try_down()

12.7.1 Function Documentation

12.7.1.1 int qurt_sem_try_down (qurt_sem_t * *sem*)

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

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

12.8 qurt_sem_up()

12.8.1 Function Documentation

12.8.1.1 static int qurt_sem_up (qurt_sem_t * *sem*)

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, then 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 may occur.

Associated data types

qurt_sem_t

Parameters

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

Returns

Unused integer value.

Dependencies

None.

12.9 Data Types

This section describes data types for semaphore services.

- Semaphores are represented in QuRT as objects of type qurt_sem_t.

12.9.1 Data Structure Documentation

12.9.1.1 union qurt_sem_t

QuRT semaphore type.

13 Barriers

Threads use barriers to synchronize their execution at a specific point in a program. Barriers are shared objects which support the following operations:

- Init barrier – Initialize a barrier.
- Destroy barrier – Destroy the specified barrier.
- Wait barrier – Suspend the current thread on the specified barrier.

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.

Functions
Barrier services are accessed with the following QuRT functions.

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

13.1 qurt_barrier_destroy()

13.1.1 Function Documentation

13.1.1.1 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 happens 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.

13.2 qurt_barrier_init()

13.2.1 Function Documentation

13.2.1.1 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.

13.3 qurt_barrier_wait()

13.3.1 Function Documentation

13.3.1.1 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.

13.4 Data Types

This section describes data types for barrier services.

- Barriers are represented in QuRT as objects of type qurt_barrier_t.

13.4.1 Define Documentation

13.4.1.1 #define QURT_BARRIER_SERIAL_THREAD 1

Serial thread.

13.4.1.2 #define QURT_BARRIER_OTHER 0

Other.

13.4.2 Data Structure Documentation

13.4.2.1 union qurt_barrier_t

QuRT barrier type.

14 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 efficiently accomplish the same task without the need for polling.

Condition variables are shared objects which support the following operations:

- Init condition – Initialize the condition variable.
- Destroy condition – Destroy the specified condition variable.
- Wait condition – Suspend current thread until the specified condition is true.
- Signal condition – Signal a waiting thread that the specified condition is true.
- Broadcast condition – Signal multiple waiting threads that the specified condition is true.

A condition variable is always used with an associated mutex (Section 6) to ensure that the shared data item is checked and updated without thread contention.

When a thread wishes 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).

When a thread wishes 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 may 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.

Functions
Condition variable services are accessed with the following QuRT functions.

- qurt_cond_broadcast()
- qurt_cond_destroy()
- qurt_cond_init()
- qurt_cond_signal()
- qurt_cond_wait()

- qurt_cond_wait2()
- Data Types

14.1 qurt_cond_broadcast()

14.1.1 Function Documentation

14.1.1.1 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 a condition variable's mutex may cause the threads to never be suspended (or suspended but never awakened).

Condition variables can be used 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.

14.2 qurt_cond_destroy()

14.2.1 Function Documentation

14.2.1.1 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 happens, 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.

14.3 qurt_cond_init()

14.3.1 Function Documentation

14.3.1.1 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.

14.4 qurt_cond_signal()

14.4.1 Function Documentation

14.4.1.1 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 may cause the threads to never be suspended (or suspended but never awakened).

Condition variables can be used 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.

14.5 qurt_cond_wait()

14.5.1 Function Documentation

14.5.1.1 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 (which suspends the thread and unlocks the mutex).

Note: Failure to properly lock and unlock a condition variable's mutex can cause the threads to never be suspended (or suspended but never awakened).

Condition variables can be used 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.

14.6 qurt_cond_wait2()

14.6.1 Function Documentation

14.6.1.1 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 a condition variable's mutex can cause the threads to never be suspended (or suspended but never awakened).

Condition variables can be used 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(), but this version is used if the mutex being used is 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.

14.7 Data Types

This section describes data types for condition variable services.

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

14.7.1 Data Structure Documentation

14.7.1.1 union qurt_cond_t

QuRT condition variable type.

15 Pipes

Threads use pipes to perform synchronized exchange of data streams. Pipes are shared objects which support the following operations:

- Init pipe – Initialize the pipe.
- Create pipe – Initialize the pipe and allocate the pipe buffer.
- Destroy pipe – Destroy the pipe.
- Delete pipe – Destroy the pipe and deallocate the pipe buffer.
- Send pipe – Write data to the pipe.
- Receive pipe – Read data from the pipe.
- Try send pipe – Write data to the pipe (without suspend).
- Try receive pipe – Read data from the pipe (without suspend).
- Send pipe cancellable – Write data to the pipe (with suspend), cancellable.
- Receive pipe cancellable – Read data from the pipe (with suspend), cancellable.
- Pipe is empty – Return value indicating whether the pipe contains any data.

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 a pipe object is created 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 thread's user process is killed, or if the thread must finish its current QDI invocation and return to 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.

Pipe attributes

Pipes have the following attributes:

- Buffer – Pipe buffer address.
- Elements – Pipe buffer length.
- Buffer partition – Pipe buffer allocated in either RAM or TCM/LPM.

The pipe buffer address specifies the byte address of the start of the pipe data buffer.

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

Setting pipe attributes

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

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.

Functions

Pipe services are accessed with the following QuRT functions.

- 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

15.1 qurt_pipe_attr_init()

15.1.1 Function Documentation

15.1.1.1 static void qurt_pipe_attr_init (qurt_pipe_attr_t * *attr*)

Initializes the structure that is used to set 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.

15.2 qurt_pipe_attr_set_buffer()

15.2.1 Function Documentation

15.2.1.1 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 be used for a pipe's data buffer.

The base address and size (Section 15.4.1.1) 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.

15.3 qurt_pipe_attr_set_buffer_partition()

15.3.1 Function Documentation

15.3.1.1 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. Pipes can be allocated in RAM or TCM/LPM.

Note: If a pipe is specified as being 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.

15.4 qurt_pipe_attr_set_elements()

15.4.1 Function Documentation

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

Specifies the length of the memory area to be used for a pipe's data buffer.

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 15.2.1.1) 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.

15.5 qurt_pipe_create()

15.5.1 Function Documentation

15.5.1.1 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 being 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.

15.6 qurt_pipe_delete()

15.6.1 Function Documentation

15.6.1.1 void qurt_pipe_delete (qurt_pipe_t * *pipe*)

Deletes the pipe.

Destroys the specified pipe (Section 15.7.1.1) and deallocates the pipe object and its associated data buffer.

Note: Pipes should be deleted 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 happens, 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.

15.7 qurt_pipe_destroy()

15.7.1 Function Documentation

15.7.1.1 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 happens 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.

15.8 qurt_pipe_init()

15.8.1 Function Documentation

15.8.1.1 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 has already been allocated by the user.

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.

15.9 qurt_pipe_is_empty()

15.9.1 Function Documentation

15.9.1.1 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.

15.10 qurt_pipe_receive()

15.10.1 Function Documentation

15.10.1.1 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: Data items larger than 64 bits can be transferred 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.

15.11 qurt_pipe_receive_cancellable()

15.11.1 Function Documentation

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

Reads a data item from the specified pipe.

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: Data items larger than 64 bits can be transferred 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.

15.12 qurt_pipe_send()

15.12.1 Function Documentation

15.12.1.1 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: Data items larger than 64 bits can be transferred 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 be written.

Returns

None.

Dependencies

None.

15.13 qurt_pipe_send_cancellable()

15.13.1 Function Documentation

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

Writes a data item to the specified pipe.

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 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 writes are limited to transferring a single 64-bit data item per operation.

Note: Data items larger than 64 bits can be transferred 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 be written.

Returns

QURT_EOK – Send completed.
QURT_ECANCEL – Send cancelled.

Dependencies

None.

15.14 qurt_pipe_try_receive()

15.14.1 Function Documentation

15.14.1.1 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: Data items larger than 64 bits can be transferred 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.

15.15 qurt_pipe_try_send()

15.15.1 Function Documentation

15.15.1.1 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: Data items larger than 64 bits can be transferred 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 be written.

Returns

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

Dependencies

None.

15.16 Data Types

This section describes data types for pipe services.

- 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.

15.16.1 Define Documentation

15.16.1.1 #define QURT_PIPE_MAGIC 0xF1FEF1FE

Magic.

15.16.1.2 #define QURT_PIPE_ATTR_MEM_PARTITION_RAM 0

RAM.

15.16.1.3 #define QURT_PIPE_ATTR_MEM_PARTITION_TCM 1

TCM.

15.16.2 Data Structure Documentation

15.16.2.1 struct qurt_pipe_t

QuRT pipe type.

15.16.2.2 struct qurt_pipe_attr_t

QuRT pipe attributes type.

15.16.3 Typedef Documentation

15.16.3.1 typedef unsigned long long int qurt_pipe_data_t

QuRT pipe data values type.

16 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.

Timer objects are assigned to specific threads. They support the following operations:

- Create timer – Create the timer with specified attributes and start it.
- Delete timer – Delete the specified timer.
- Stop timer – Stop running the timer.
- Restart timer – Restart the stopped timer.
- Sleep timer – Suspend the current thread for the specified duration.
- Timer group enable – Enable all timers assigned to the specified timer group.
- Timer group disable – Disable all timers assigned to the specified timer group.
- Get timer attributes – Get attributes assigned to a timer when it was created.

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.

A running timer can be stopped by calling the timer stop operation. A stopped (or expired) timer can be restarted with a specified duration by calling the timer restart operation.

A thread can suspend itself (Section 4) 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 which make it possible to enable or disable one or more timers with a single operation. A timer state is saved across disabling and subsequent reenabling.

The static attributes of a running timer can be accessed 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.

Attributes
Timers have the following attributes:

- Duration – Interval between timer events.
- Type – Timer functional behavior (one-shot or periodic).
- Group – Timer group that timer is assigned to.
- Remaining – Time remaining before next timer event (read-only).
- Expiry – Absolute time when timer expires (one-shot only).

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 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.

The timer group specifies the group that the timer is assigned to. Timer groups are used to enable or disable one or more timers with a single operation.

The timer remaining returns the time remaining (in microseconds) before the generation of the next timer event on the timer. This attribute is read-only.

The timer expiry specifies 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.

Setting timer attributes
The timer attributes are set before a timer is created using the qurt_timer_attr_init and qurt_timer_attr_set functions.

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.

Getting timer attributes
Timer attributes can be retrieved from a created timer using the qurt_timer_get_attr() and qurt_timer_attr_get functions.

Of the various 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. All the other returned attributes are static, and remain unchanged from when they were set.

Functions

Timer services are accessed with the following QuRT functions.

- 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()
- Data Types

16.1 qurt_timer_attr_get_duration()

16.1.1 Function Documentation

16.1.1.1 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.

16.2 qurt_timer_attr_get_group()

16.2.1 Function Documentation

16.2.1.1 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.

16.3 qurt_timer_attr_get_remaining()

16.3.1 Function Documentation

16.3.1.1 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.

16.4 qurt_timer_attr_get_type()

16.4.1 Function Documentation

16.4.1.1 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.

16.5 qurt_timer_attr_init()

16.5.1 Function Documentation

16.5.1.1 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 16)
- 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.

16.6 qurt_timer_attr_set_duration()

16.6.1 Function Documentation

16.6.1.1 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 16). Otherwise, the set operation is ignored.

Note: The maximum timer duration is 36 hours.

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.

16.7 qurt_timer_attr_set_expiry()

16.7.1 Function Documentation

16.7.1.1 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.

16.8 qurt_timer_attr_set_group()

16.8.1 Function Documentation

16.8.1.1 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) (Section 16).

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.

16.9 qurt_timer_attr_set_type()

rest_dist

16.9.1 Function Documentation

16.9.1.1 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.

16.10 qurt_timer_create()

16.10.1 Function Documentation

16.10.1.1 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.

Dependencies

None.

16.11 qurt_timer_delete()

16.11.1 Function Documentation

16.11.1.1 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.

16.12 qurt_timer_get_attr()

16.12.1 Function Documentation

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

Gets the timer attributes of the specified timer.

Note: After a timer is created, the attributes assigned to the thread cannot be changed by the user.

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.

16.13 qurt_timer_group_disable()

16.13.1 Function Documentation

16.13.1.1 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.

16.14 qurt_timer_group_enable()

16.14.1 Function Documentation

16.14.1.1 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.

16.15 qurt_timer_restart()

16.15.1 Function Documentation

16.15.1.1 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, with the starting time being 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 16.17.1.1.

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.

16.16 qurt_timer_sleep()

16.16.1 Function Documentation

16.16.1.1 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 16).

Note: The maximum sleep duration is 36 hours. The error margin of the sleep timer is approximately 90 microseconds (due to a setup time of two ticks and resolution of one tick).

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.

16.17 qurt_timer_stop()

16.17.1 Function Documentation

16.17.1.1 int qurt_timer_stop (qurt_timer_t *timer*)

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

Note: Stopped timers can be restarted with the timer restart operation, see Section 16.15.1.1.

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.

16.18 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.

16.18.1 Define Documentation

16.18.1.1 #define QURT_TIMER_DEFAULT_TYPE QURT_TIMER_ONESHOT

One shot.

16.18.1.2 #define QURT_TIMER_DEFAULT_DURATION 1000uL

Default value.

16.18.1.3 #define QURT_TIMER_MAX_GROUPS 5

Maximum groups.

16.18.1.4 #define QURT_TIMER_DEFAULT_GROUP 0

Default groups.

16.18.2 Data Structure Documentation

16.18.2.1 struct qurt_timer_attr_t

QuRT timer attribute type.

16.18.3 Typedef Documentation

16.18.3.1 typedef unsigned int qurt_timer_t

QuRT timer type.

16.18.3.2 typedef unsigned long long qurt_timer_duration_t

QuRT timer duration type.

16.18.3.3 typedef unsigned long long qurt_timer_time_t

QuRT timer time type.

16.18.4 Enumeration Type Documentation

16.18.4.1 enum qurt_timer_type_t

QuRT timer types.

Enumerator:

QURT_TIMER_ONESHOT One shot.
QURT_TIMER_PERIODIC Periodic.

16.19 Constants and Macros

This section describes constants and macros for timer services.

16.19.1 Define Documentation

16.19.1.1 #define QURT_TIMER_MIN_DURATION 100uL

The minimum microseconds value is 100 microseconds (sleep timer).

16.19.1.2 #define QURT_TIMER_MAX_DURATION QURT_SYSCLOCK_MAX_DURATION

The maximum microseconds value for Qtimer is 1042499 hours.

17 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 will occur 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.

The system clock supports the following operations:

- Register client – Register the client for the system clock event.
- Create alarm – Create the system clock alarm with the specified time and start it.
- Create timer – Create the system clock timer with the specified duration and start it.
- Get hardware ticks – Get the hardware tick count.
- Get timer expiry – Return the time remaining before the next event on any timer.

Unlike regular timers (Section 16), 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 get hardware ticks operation 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 must be used with care because of the wrap-around behavior.

The get timer expiry operation returns the number of system clock ticks remaining before the next timer event occurs on any user-defined timer or alarm in the QuRT application system (including those described in Section 16).

Functions

System clock services are accessed with the following QuRT functions.

- qurt_sysclock_alarm_create()
- qurt_sysclock_get_expiry()
- qurt_sysclock_get_hw_ticks()
- qurt_sysclock_get_hw_ticks_32()
- qurt_sysclock_get_hw_ticks_16()
- qurt_sysclock_register()
- qurt_sysclock_timer_create()

17.1 qurt_sysclock_alarm_create()

17.1.1 Function Documentation

17.1.1.1 unsigned long long qurt_sysclock_alarm_create (int *id*, unsigned long long *ref_count*, unsigned long long *match_value*)

Creates a system clock alarm with the specified time value, and starts the alarm.

Parameters

in	*id*	System clock client ID; indirectly indicating the signal that the alarm-expired event sets. The signal must already be registered to receive a system clock event (Section 17.6.1.1) – it is specified here by the client identifier that is returned by the register operation.
in	*ref_count*	System clock count when the match value was calculated, which specifies the system clock time when the match_value parameter is calculated. This value is obtained using the get hardware ticks operation (Section 17.3.1.1).
in	*match_value*	Match value to be programmed in system clock hardware; indicates the system clock time (in system clock ticks) when the alarm should expire.

Returns

Integer – Match value programmed.

Dependencies

None.

17.2 qurt_sysclock_get_expiry()

17.2.1 Function Documentation

17.2.1.1 unsigned long long qurt_sysclock_get_expiry (void)

Gets the duration until next timer event.

Returns the number of system ticks that elapse before the next timer event occurs on any active timer in the QuRT application system.

A system tick is defined as one cycle of the Hexagon processor's 19.2 MHz Qtimer clock.

Returns

Integer – Number of system ticks until next timer event.

Dependencies

None.

17.3 qurt_sysclock_get_hw_ticks()

17.3.1 Function Documentation

17.3.1.1 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.

17.4 qurt_sysclock_get_hw_ticks_32()

17.4.1 Variable Documentation

17.4.1.1 int qurt_timer_base

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.

17.5 qurt_sysclock_get_hw_ticks_16()

17.5.1 Function Documentation

17.5.1.1 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.

17.6 qurt_sysclock_register()

17.6.1 Function Documentation

17.6.1.1 int qurt_sysclock_register (qurt_anysignal_t ∗ *signal,* unsigned int *signal_mask*)

Register a signal object to receive an event on a system clock alarm or timer.

The return value is a client identifier value, which is used to associate a registered signal with a system clock alarm or timer object.

Associated data types

qurt_anysignal_t

Parameters

in	*signal*	Signal object set when the system clock event occurs.
in	*signal_mask*	Signal mask, which specifies the signal to set in the signal object when the clock event occurs.

Returns

Integer – System clock client identifier.
QURT_EFATAL – Not enough memory to create timer.

Dependencies

None.

17.7 qurt_sysclock_timer_create()

17.7.1 Function Documentation

17.7.1.1 int qurt_sysclock_timer_create (int *id*, unsigned long long *duration*)

Creates a system clock timer with the specified duration, and starts the timer.

Parameters

in	*id*	System clock client ID. Indirectly specifies the signal that the timer-expired event sets. The signal must already be registered to receive a system clock event (Section 17.6.1.1) – the client identifier returned by the register operation specifies the signal.
in	*duration*	Timer duration (in system clock ticks). Specifies the interval between the creation of the system clock timer object and the generation of the corresponding timer event.

Returns

QURT_EOK – Timer successfully created.

Dependencies

None.

18 Interrupts

Threads use interrupts to respond to external events. Interrupts are processor resources, which support the following operations:

- Register interrupt – Enable the signal object to receive a specific interrupt.
- Deregister interrupt – Disable the signal object from receiving a specific interrupt.
- Enable interrupt - Enable the specified interrupt.
- Disable interrupt - Disable the specified interrupt.
- Acknowledge interrupt – Re-enable the interrupt after it has been processed.
- Get registered interrupts – Return the bitmask indicating registered interrupts.
- Interrupt status – Return the pending status of the specified interrupt.
- Clear interrupt – Clear the pending status of the specified interrupt.
- Raise interrupt –Trigger an interrupt from software.
- Get interrupt configuration – Return the L2VIC interrupt type and polarity.
- Set interrupt configuration – Set the L2VIC interrupt type and polarity.

When an interrupt is registered, 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 (i.e., 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 20).

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

Interrupts do not support init and destroy operations because no objects (Section 3.4) are created for them.

Pending interrupts
A pending interrupt can be explicitly cleared with the clear interrupt operation.

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

L2VIC interrupts
In the V5 Hexagon processor, all interrupts are based on the L2VIC interrupt controller. All interrupts must be specified 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.

Functions
Interrupt services are accessed with the following QuRT functions.

- qurt_interrupt_acknowledge()
- qurt_interrupt_clear()
- qurt_interrupt_deregister()
- qurt_interrupt_disable()
- qurt_interrupt_enable()
- qurt_interrupt_get_config()
- qurt_interrupt_get_registered()
- qurt_interrupt_raise()
- qurt_interrupt_register()
- qurt_interrupt_set_config()
- qurt_interrupt_status()
- Constants

18.1 qurt_interrupt_acknowledge()

18.1.1 Function Documentation

18.1.1.1 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 has been processed by an interrupt service thread (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 subsequent occurrences of the interrupt from being lost or reprocessed, an IST must clear the interrupt signal (Section 10.1.1.1) before acknowledging the interrupt.

Parameters

in	*int_num*	Interrupt that is being re-enabled (0 through 31).

Returns

QURT_EOK – Interrupt acknowledge was successful.
QURT_EDEREGISTERED – Interrupt has already been deregistered.

Dependencies

None.

18.2 qurt_interrupt_clear()

18.2.1 Function Documentation

18.2.1.1 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; range is 0 to 31.

Returns

QURT_EOK – Success.
QURT_EINT – Invalid interrupt number.

Dependencies

None.

18.3 qurt_interrupt_deregister()

18.3.1 Function Documentation

18.3.1.1 unsigned int qurt_interrupt_deregister (int *int_num*)

Disables the specified interrupt and disassociate it from any QuRT signal object. If the specified interrupt was never registered (Section 18.9.1.1), the deregister operation returns the status value QURT_EINT.

Note: If an interrupt is deregistered while an interrupt service thread (IST) is waiting to receive it, the IST may 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.

18.4 qurt_interrupt_disable()

18.4.1 Function Documentation

18.4.1.1 unsigned int qurt_interrupt_disable (int *int_num*)

Disables the interrupt.

The interrupt for the int_num must be registered prior to calling this function. After calling qurt_interrupt_disable(), the corresponding interrupt can no longer be sent to the Hexagon processor until qurt_interrupt_enable() is called with the same int_num.

Parameters

in	*int_num*	Interrupt number.

Returns

QURT_EOK – Interrupt successfully disabled.
QURT_EINT – Invalid interrupt number.
QURT_EVAL – Interrupt has not been registered.

Dependencies

None.

18.5 qurt_interrupt_enable()

18.5.1 Function Documentation

18.5.1.1 unsigned int qurt_interrupt_enable (int *int_num*)

Enables the interrupt.

The interrupt for the int_num must be registered prior to calling this function. After calling qurt_interrupt_disable(), the corresponding interrupt can no longer be sent to the Hexagon processor until qurt_interrupt_enable() is called with the same int_num.

Parameters

in	*int_num*	Interrupt number.

Returns

QURT_EOK – Interrupt successfully enabled.
QURT_EINT – Invalid interrupt number.
QURT_EVAL – Interrupt has not been registered.

Dependencies

None.

18.6 qurt_interrupt_get_config()

18.6.1 Function Documentation

18.6.1.1 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 indicates a level-triggered interrupt, 1 indicates an edge-triggered interrupt.
out	*int_polarity*	Pointer to interrupt polarity. 0 indicates an active-high interrupt, and 1 indicates an active-low interrupt.

Returns

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

Dependencies

None.

18.7 qurt_interrupt_get_registered()

18.7.1 Function Documentation

18.7.1.1 unsigned int qurt_interrupt_get_registered (void)

Gets the registered L1 interrupts.

This function returns a bitmask indicating which L1 interrupts have been registered.

Interrupts are represented as bits 0 through 31 in the 32-bit mask value. A mask bit value of 1 indicates that an interrupt is registered. 0 indicates that an interrupt is not registered.

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

Returns

Bitmask – Registered L1 interrupts.

Dependencies

None.

18.8 qurt_interrupt_raise()

18.8.1 Function Documentation

18.8.1.1 int qurt_interrupt_raise (unsigned int *interrupt_num*)

Raises the interrupt.

On the V5 Hexagon processor, 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

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

Dependencies

None.

18.9 qurt_interrupt_register()

18.9.1 Function Documentation

18.9.1.1 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 is to be waited on, and 0 that it is not to be waited on.

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

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.

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.

Note: The valid range for an interrupt number may 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 10).
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.

18.10 qurt_interrupt_set_config()

18.10.1 Function Documentation

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

Parameters

in	*int_num*	L2VIC interrupt that is being re-enabled.
in	*int_type*	Interrupt type, with 0 indicating a level-triggered interrupt, and 1 an edge-triggered interrupt.
in	*int_polarity*	Interrupt polarity, with 0 indicating an active-high interrupt, and 1 an active-low interrupt.

Returns

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

Dependencies

None.

18.11 qurt_interrupt_status()

18.11.1 Function Documentation

18.11.1.1 unsigned int qurt_interrupt_status (int *int_num*, int * *status*)

Returns a value indicating the pending status of the specified interrupt.

Parameters

in	*int_num*	Interrupt that is being checked; the range is 0 to 31.
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.

18.12 Constants

This section describes constants for interrupt services.

18.12.1 Define Documentation

18.12.1.1 #define SIG_INT_ABORT 0x80000000

19 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.

Thread local storage supports the following operations:

- Init TLS – Initialize thread local storage.
- Create key – Create a unique key for accessing a data item in thread local storage.
- Set specific – Store a data item to thread local storage along with the specified key.
- Get specific – Load a data item that is stored in thread local storage with the specified key.
- Delete key – Delete the specified key.

The create key operation returns a value (known as a key) which specifies a data item within the thread local storage. The key is used to access the data item in subsequent get and set operations. A destructor function for the item can be optionally specified as part of the create key operation.

The set and get specific operations both use key values to specify data items for loading or storing in thread local storage. The data items themselves are always pointers to user data.

The delete key operation deletes the specified key from thread local storage.

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 the kernel. QuRT's thread local storage service is POSIX-compatible.

Thread local storage keys in QuRT are identified by values of type int.

Functions
Thread local storage services are accessed with the following QuRT functions.

- qurt_tls_create_key()
- qurt_tls_delete_key()
- qurt_tls_get_specific()
- qurt_tls_set_specific()

19.1 qurt_tls_create_key()

19.1.1 Function Documentation

19.1.1.1 int qurt_tls_create_key (int * *key*, void(*)(void *) *destructor*)

Creates a key for accessing a thread local storage data item.

The key value is used in subsequent get and set operations.

Note: The destructor function performs any clean-up operations needed by a thread local storage item when its containing thread is deleted (Section 4.12.1.1).

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.

19.2 qurt_tls_delete_key()

19.2.1 Function Documentation

19.2.1.1 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 19.1.1.1).

Parameters

in	*key*	Thread local storage key value to delete.

Returns

QURT_EOK – Key successfully deleted.
QURT_ETLSENTRY – Key already free.

Dependencies

None.

19.3 qurt_tls_get_specific()

19.3.1 Function Documentation

19.3.1.1 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.

19.4 qurt_tls_set_specific()

19.4.1 Function Documentation

19.4.1.1 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.

20 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 the following types of exceptions:

- Program exceptions (fatal or nonfatal)
- Kernel exceptions
- Imprecise exceptions

Program 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.

Program exceptions are handled by a thread (Section 4) which is registered as the program exception handler.

Program exception handling supports the following operations:

- Wait for exception - Register the current thread as the program exception handler and wait for a program exception.
- Raise nonfatal exception - Raise a nonfatal program exception.
- Raise fatal exception - Raise a fatal system exception.
- Shutdown fatal - Shut down the system due to a fatal exception.
- Register fatal notification - Register a fatal notification handler.
- Enable FP exceptions - Enable floating point exceptions.
- Wait on page fault - Register a page fault handler.

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.

Depending on how it is implemented, a program exception handler may handle a nonfatal exception either by reloading the QuRT program (if it has the ability to do so) or by terminating the execution of the QuRT program system.

Note: If no program exception handler is registered, or if the registered handler itself calls raise nonfatal exception, then QuRT raises a kernel exception.

If a thread runs in Supervisor mode, any errors are treated as kernel exceptions.

If multiple program exceptions occur, all exceptions are forwarded to the program exception handler in the order that the exceptions occur. To handle multiple exceptions, the exception handler must make repeated calls to qurt_exception_wait to process the error information from the queued exceptions.

Fatal program exceptions
Fatal program exceptions terminate the execution of the QuRT program system without invoking the program exception handler. They are intended for use where the program will handle all the system shutdown operations.

Fatal exceptions are raised by calling the raise fatal exception operation, which masks the Hexagon processor interrupts and stops all 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, etc.).

Once the program is ready to shut down the system, it calls the fatal shutdown operation, which performs the following actions:

1. If the raise fatal exception operation was not already called, mask the processor interrupts and stop all 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.

Kernel exceptions
Kernel exceptions are raised by the QuRT kernel – 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 in post-mortem 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 all 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
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 also 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.

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.

Functions
Exception handling services are accessed with the following QuRT functions.

- qurt_exception_enable_fp_exceptions()
- qurt_exception_raise_fatal()
- qurt_exception_raise_nonfatal()
- qurt_exception_register_fatal_notification()
- qurt_exception_shutdown_fatal()
- qurt_exception_shutdown_fatal2()
- qurt_exception_wait()
- qurt_exception_wait2()
- qurt_exception_wait_pagefault()

20.1 qurt_exception_enable_fp_exceptions()

20.1.1 Function Documentation

20.1.1.1 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 register USR.

The mask argument specifies a mask value identifying the individual floating point exceptions to be 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 any floating point operations are performed.

Parameters

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 register.

Dependencies

None.

20.2 qurt_exception_raise_fatal()

20.2.1 Function Documentation

20.2.1.1 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 20.

This operation always returns, so the calling program can perform the necessary shutdown operations (data logging, etc.).

Note: Context switches do not work after this operation has been called.

Returns

None.

Dependencies

None.

20.3 qurt_exception_raise_nonfatal()

20.3.1 Function Documentation

20.3.1.1 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 20.

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.

20.4 qurt_exception_register_fatal_notification()

20.4.1 Function Documentation

20.4.1.1 unsigned int qurt_exception_register_fatal_notification (void(*)(void *) *entryfuncpoint*, void * *argp*)

Registers a fatal exception notification handler with the RTOS kernel.

The handler function is intended to perform the final steps of system shutdown after all the other shutdown actions have been performed (e.g., notifying the host processor of the shutdown). It should perform only a minimal amount of execution.

Note: The fatal notification handler executes on the Hexagon processor in user mode.

Parameters

in	*entryfuncpoint*	Pointer to the handler function.
in	*argp*	Pointer to the argument list passed to handler function when it is invoked.

Returns

Registry status:
QURT_EOK – Success
QURT_EVAL – Failure; invalid parameter

Dependencies

None.

20.5 qurt_exception_shutdown_fatal()

20.5.1 Function Documentation

20.5.1.1 void qurt_exception_shutdown_fatal (void)

Performs the fatal shutdown procedure for handling a fatal program exception.

For more information on the fatal shutdown procedure, see Section 20.

Note: This operation does not return, as it shuts down the system.

Returns

None.

Dependencies

None.

20.6 qurt_exception_shutdown_fatal2()

20.6.1 Function Documentation

20.6.1.1 void qurt_exception_shutdown_fatal2 (void)

Performs the fatal shutdown procedure for handling a fatal program exception. This operation always returns, so the calling program can complete the fatal shutdown procedure. For more information on the fatal shutdown procedure, see Section 20.

Note: This function differs from qurt_exception_shutdown_fatal() by always returning to the caller.

Returns

None.

Dependencies

None.

20.7 qurt_exception_wait()

20.7.1 Function Documentation

20.7.1.1 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, then 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.

20.8 qurt_exception_wait2()

20.8.1 Function Documentation

20.8.1.1 static unsigned int qurt_exception_wait2 (qurt_sysevent_error_t * *sys_err*)

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).

Associated data types

qurt_sysevent_error_t

Parameters

out	*sys_err*	Pointer to the error event record.

Returns

Registry status:

- QURT_EFATAL – Failure.
- Thread ID – Success.

Dependencies

None.

20.9 qurt_exception_wait_pagefault()

20.9.1 Function Documentation

20.9.1.1 unsigned int qurt_exception_wait_pagefault (qurt_sysevent_pagefault_t * *sys_pagefault*)

Registers the page fault handler. This function assigns the current thread as the QuRT page fault handler and suspends the thread until a page fault occurs.

When a page fault occurs, the thread is awakened with page fault information assigned to the parameters of this operation.

Parameters

out	*sys_pagefault*	Pointer to the page fault event record, the instruction memory address where the exception occurred.

Returns

Registry status:
QURT_EOK – Success.
QURT_EFAILED – Failure due to existing pager registration.

Dependencies

None.

21 Memory Allocation

QuRT user programs are assigned a default global heap, which is accessed by the standard C functions malloc and free (Section 3.1).

Threads use memory allocation to create additional heap-based storage allocators within user programs.

Memory allocation supports the following operations:

- Allocate memory – Allocate memory area in heap.
- Allocate array – Allocate an array in heap.
- Reallocate memory – Reallocate existing memory area in heap.
- Deallocate memory – Free allocated memory from heap.

The allocate memory operations are functionally similar to the corresponding operations malloc and calloc that are defined in the C library.

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.

The deallocate memory operation is functionally similar to free.

Note: Memory allocation cannot allocate memory outside the thread assigned memory area (Section 3.1). This can be done using the QuRT memory management services (Section 22).

Functions

Memory allocation services are accessed with the following QuRT functions.

- qurt_calloc()
- qurt_free()
- qurt_malloc()
- qurt_realloc()

21.1 qurt_calloc()

21.1.1 Function Documentation

21.1.1.1 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.

21.2 qurt_free()

21.2.1 Function Documentation

21.2.1.1 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 be deallocated.

Returns

None.

Dependencies

The memory item specified by the ptr value must have been previously allocated using one of the memory allocation functions (qurt_calloc, qurt_malloc, qurt_realloc). Otherwise the behavior of QuRT is undefined.

21.3 qurt_malloc()

21.3.1 Function Documentation

21.3.1.1 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.

21.4 qurt_realloc()

21.4.1 Function Documentation

21.4.1.1 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.

Note: This function may 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 re-allocated memory area.

Returns

Nonzero – Pointer to re-allocated memory area.
0 – Not enough memory in heap to re-allocate the memory area.

Dependencies

None.

22 Memory Management

Threads use memory management to dynamically allocate user program memory, share memory with other user programs, and manage virtual memory. Memory management is performed with memory pools and regions.

Threads use memory management to perform the following tasks:

- Dynamically allocate memory outside the assigned user program memory
- Manage virtual memory

Memory management is performed with memory pools, regions, and mappings, objects that support the following operations:

- Create pool – Create a memory pool from a physical address range.
- Attach pool – Attach a memory pool to a predefined pool.
- Create region – Create a memory region with specified attributes.
- Delete region – Delete the specified region.
- Create mapping – Create a memory map entry in the page table.
- Remove mapping – Delete the specified memory map entry from the page table.
- Add pages – Add physical address range to the specified pool.
- Remove pages – Remove physical address range from the specified pool.
- Query region – Determine if a region exists for a specified address.
- Query map – Determine if a memory page is statically mapped.
- Lookup physical address – Translate a virtual address to a physical address.
- Cache clean – Perform a cache control operation.

To dynamically allocate memory outside its assigned memory area (Section 3.1), a thread first 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 then 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

Memory pools are used to assign memory regions to different types of physical (not virtual) memory. For example, the Hexagon processor can access SMI, TCM, and EBI memory; therefore, to allocate regions in each of these memories a separate memory pool must be defined 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.

Memory pools are predefined in the system configuration file (Section 3.2), and are specified by their assigned pool name in memory pool attach operations. All memory pools are accessible by all user programs in the QuRT user program system.

Memory pools can also be created at run time using qurt_mem_pool_create().

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.

The add pages and remove pages operations are used to directly manipulate the memory pools.

Memory regions

Memory regions are used to define memory areas with a fixed set of attributes that specify an area's virtual memory mapping and cache type. A core set of regions is predefined in the system configuration file (Section 3.2), 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.
- Mapping – Memory mapping.
- Physical address – Memory region physical address.
- Virtual address – Memory region virtual address.
- Cache mode – Cache type.
- Bus – Bus attributes.
- Type – Memory region type (local/shared).

The pool specifies the memory pool that the region belongs to. Each region must have a corresponding pool.

The mapping indicates how the memory region is mapped in virtual memory:

- Virtual mapped regions have their virtual address 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.

The physical address indicates the physical base address of the memory region. It is set only when using physical-contiguous-mapped memory regions.

The virtual address is a read-only attribute, which returns the base address of the memory region.

The bus attributes indicate the (A1, A0) bus attribute bits.

The Cache mode indicates whether the memory region uses the instruction or data cache.

The type 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.

Setting region attributes
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 – all the other memory region attributes are set before the create operation.

Getting region attributes
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
Memory maps specify the mapping between virtual memory and physical memory in the Hexagon processor.

The create mapping and remove mapping operations are used to 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 page's virtual address. 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 cache
The memory caches are managed with the following operations:

- Cache clean – Perform operation on memory cache.
- Lock L2 cache line – Lock the specified lines in the L2 cache.
- Unlock L2 cache line – Unlock the specified lines in the L2 cache.
- Disable L2 fetch – Disable L2 fetch on all hardware threads.
- Configure cache partition – Configure cache partition at the system level.

The cache clean operation performs the specified operation on the Hexagon memory cache. The operation can specify the data or instruction cache, and supports the following flush options: flush, invalidate, or flush and invalidate.

Memory ordering

Some devices require synchronization of stores and loads when they are accessed. This synchronization can be done with the following operations:

- Barrier – Create a barrier for memory transactions.
- Syncht – Performs heavy-weight synchronization of memory transactions.

The barrier operation ensures that all previous memory transactions are globally observable before any future memory transactions are globally observable.

The syncht operation does not return until all previous memory transactions (such as cached and uncached load, and store) that originated from the current thread are completed and globally observable.

Functions

Memory management services are accessed with the following QuRT functions.

- qurt_l2fetch_disable()
- qurt_lookup_physaddr()
- 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_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_attr_get()
- qurt_mem_pool_attr_get_addr()
- 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()
- Data Types

22.1 qurt_l2fetch_disable()

22.1.1 Function Documentation

22.1.1.1 void qurt_l2fetch_disable (void)

Disables L2FETCH activities on all hardware threads.

Returns

None.

Dependencies

None.

22.2 qurt_lookup_physaddr()

22.2.1 Function Documentation

22.2.1.1 qurt_paddr_t qurt_lookup_physaddr (qurt_addr_t *vaddr*)

Translates a virtual memory address to the physical memory address it is mapped to.

Associated data types

qurt_addr_t
qurt_paddr_t

Parameters

in	*vaddr*	Virtual address.

Returns

Nonzero – Physical address the virtual address is mapped to.
0 – Virtual address not mapped.

Dependencies

None.

22.3 qurt_lookup_physaddr_64()

22.3.1 Function Documentation

22.3.1.1 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.

Associated data types

qurt_paddr_64_t
qurt_addr_t

Parameters

in	*vaddr*	Virtual address.

Returns

Nonzero – 64-bit physical address the virtual address is mapped to.
0 – Virtual address has not been mapped.

Dependencies

None.

22.4 qurt_mapping_create()

22.4.1 Function Documentation

22.4.1.1 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.

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, etc.).
in	*perm*	Access permissions.

Returns

QURT_EOK – Mapping created.
QURT_EMEM – Failed to create mapping.

Dependencies

None.

22.5 qurt_mapping_create_64()

22.5.1 Function Documentation

22.5.1.1 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.

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, etc.).
in	*perm*	Access permissions.

Returns

QURT_EOK – Success.
QURT_EMEM – Failure.

Dependencies

None.

22.6 qurt_mapping_remove()

22.6.1 Function Documentation

22.6.1.1 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.

22.7 qurt_mapping_remove_64()

22.7.1 Function Documentation

22.7.1.1 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.

22.8 qurt_mem_barrier()

22.8.1 Function Documentation

22.8.1.1 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.

22.9 qurt_mem_cache_clean()

22.9.1 Function Documentation

22.9.1.1 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.

Note: The flush all operation can be performed only on the data cache.

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 be flushed.
in	*size*	Size (in bytes) of data to be flushed.
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
	type	Cache type. Values: • QURT_MEM_ICACHE • QURT_MEM_DCACHE

Returns

QURT_EOK – Cache operation performed successfully.
QURT_EVAL – Invalid cache type.
QURT_EALIGN – Aligning data or address failed.

Dependencies

None.

22.10 qurt_mem_configure_cache_partition()

rest_dist

22.10.1 Function Documentation

22.10.1.1 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: This operation can be called 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.

22.11 qurt_mem_l2cache_line_lock()

22.11.1 Function Documentation

22.11.1.1 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: The line lock operation can be performed 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 be locked; the address must be 32-byte aligned.
in	*size*	Size (in bytes) of L2 cache memory to be line locked; size must be a multiple of 32 bytes.

Returns

QURT_EOK – Success.
QURT_EALIGN – Data alignment or address failure.

Dependencies

None.

22.12 qurt_mem_l2cache_line_unlock()

22.12.1 Function Documentation

22.12.1.1 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: The line unlock operation can be performed 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 be unlocked; the address must be 32-byte aligned.
in	*size*	Size (in bytes) of the L2 cache memory line to be unlocked; 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.

22.13 qurt_mem_map_static_query()

22.13.1 Function Documentation

22.13.1.1 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 if 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.

  QuRT memory maps are defined in the system configuration file.

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, etc.).
in	*perm*	Access permissions.

Returns

QURT_EOK – Specified page is statically mapped, the 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.

22.14 qurt_mem_map_static_query_64()

22.14.1 Function Documentation

22.14.1.1 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. Pages are specified by the following attributes: 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, etc.).
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.

22.15 qurt_mem_pool_add_pages()

22.15.1 Function Documentation

22.15.1.1 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: This operation can be called 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.

22.16 qurt_mem_pool_attach()

22.16.1 Function Documentation

22.16.1.1 int qurt_mem_pool_attach (char * *name*, qurt_mem_pool_t * *pool*)

Initializes a memory pool object to be attached to a pool predefined in the system configuration file.

Memory pool objects are used to assign memory regions to physical memory in different Hexagon memory units. They are specified in memory region create operations (Section 22.39.1.1).

Note: QuRT predefines the memory pool object qurt_mem_default_pool (Section 22) 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.

22.17 qurt_mem_pool_attr_get()

22.17.1 Function Documentation

22.17.1.1 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.

22.18 qurt_mem_pool_attr_get_addr()

22.18.1 Function Documentation

22.18.1.1 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.

22.19 qurt_mem_pool_attr_get_size()

22.19.1 Function Documentation

22.19.1.1 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.

22.20 qurt_mem_pool_create()

22.20.1 Function Documentation

22.20.1.1 int qurt_mem_pool_create (char * *name*, unsigned *base*, unsigned *size*, qurt_mem_pool_t * *pool*)

Dynamically creates a memory pool object.

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.

22.21 qurt_mem_pool_remove_pages()

22.21.1 Function Documentation

22.21.1.1 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: This operation can be called 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.

22.22 qurt_mem_region_attr_get()

22.22.1 Function Documentation

22.22.1.1 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.

22.23 qurt_mem_region_attr_get_bus_attr()

22.23.1 Function Documentation

22.23.1.1 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.

22.24 qurt_mem_region_attr_get_cache_mode()

22.24.1 Function Documentation

22.24.1.1 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.

22.25 qurt_mem_region_attr_get_mapping()

22.25.1 Function Documentation

22.25.1.1 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.

22.26 qurt_mem_region_attr_get_physaddr()

22.26.1 Function Documentation

22.26.1.1 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.

22.27 qurt_mem_region_attr_get_size()

22.27.1 Function Documentation

22.27.1.1 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.

22.28 qurt_mem_region_attr_get_type()

22.28.1 Function Documentation

22.28.1.1 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.

22.29 qurt_mem_region_attr_get_virtaddr()

22.29.1 Function Documentation

22.29.1.1 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.

22.30 qurt_mem_region_attr_get_physaddr_64()

22.30.1 Function Documentation

22.30.1.1 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.

22.31 qurt_mem_region_attr_init()

22.31.1 Function Documentation

22.31.1.1 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.

22.32 qurt_mem_region_attr_set_bus_attr()

22.32.1 Function Documentation

22.32.1.1 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 be used with the memory region, expressed as a 2-bit binary number.

Returns

None.

Dependencies

None.

22.33 qurt_mem_region_attr_set_cache_mode()

22.33.1 Function Documentation

22.33.1.1 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.

22.34 qurt_mem_region_attr_set_mapping()

22.34.1 Function Documentation

22.34.1.1 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.

22.35 qurt_mem_region_attr_set_physaddr()

22.35.1 Function Documentation

22.35.1.1 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 it is automatically set by QuRT 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.

22.36 qurt_mem_region_attr_set_physaddr_64()

22.36.1 Function Documentation

22.36.1.1 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.

22.37 qurt_mem_region_attr_set_type()

22.37.1 Function Documentation

22.37.1.1 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.

22.38 qurt_mem_region_attr_set_virtaddr()

22.38.1 Function Documentation

22.38.1.1 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.

22.39 qurt_mem_region_create()

22.39.1 Function Documentation

22.39.1.1 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 memory region attribute structure is initialized by the application 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 22.31.1.1), 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 22.31.1.1). QuRT predefines the memory pool object qurt_mem_default_pool (Section 22), 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.

22.40 qurt_mem_region_delete()

22.40.1 Function Documentation

22.40.1.1 int qurt_mem_region_delete (qurt_mem_region_t *region*)

Deletes the specified memory region.

If the memory region was created by the caller application, it is removed and its assigned memory reclaimed by the system.

If the memory region was created by a different application (and shared with the caller application), then only the local memory mapping to the region is removed; the memory itself is not reclaimed by the system.

Associated data types

qurt_mem_region_t

Parameters

in	*region*	Memory region object.

Returns

QURT_EOK – Region successfully deleted.

Dependencies

None.

22.41 qurt_mem_region_query()

22.41.1 Function Documentation

22.41.1.1 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 22.39.1.1) exists for the specified virtual or physical address. Once a memory region has been determined to exist, its attributes can be accessed (Section 22.22.1.1).

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.

22.42 qurt_mem_region_query_64()

22.42.1 Function Documentation

22.42.1.1 int qurt_mem_region_query_64 (qurt_mem_region_t * *region_handle*, qurt_addr_t *vaddr*, qurt_paddr_64_t *paddr_64*)

Determines if a dynamically created memory region (Section 22.39.1.1) exists for the specified virtual or physical address. Once a memory region has been determined to exist, its attributes can be accessed (Section 22.22.1.1).

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.

22.43 qurt_mem_syncht()

22.43.1 Function Documentation

22.43.1.1 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, etc.) that originated from the current thread are completed and globally observable.

Note: This operation is implemented as a wrapper for the Hexagon syncht instruction.

Returns

None.

Dependencies

None.

22.44 Data Types

This section describes data types for memory manangement services.

- 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.

22.44.1 Define Documentation

22.44.1.1 #define QURT_POOL_REMOVE_ALL_OR_NONE 1

22.44.2 Data Structure Documentation

22.44.2.1 struct qurt_mem_region_attr_t

QuRT memory region attributes type.

22.44.2.2 struct qurt_mem_pool_attr_t

QuRT user physical memory pool type.

22.44.3 Typedef Documentation

22.44.3.1 typedef unsigned int qurt_addr_t

QuRT address type.

22.44.3.2 typedef unsigned int qurt_paddr_t

QuRT physical memory address type.

22.44.3.3 typedef unsigned long long qurt_paddr_64_t

QuRT 64-bit physical memory address type.

22.44.3.4 typedef unsigned int qurt_mem_region_t

QuRT memory regions type.

22.44.3.5 typedef unsigned int qurt_mem_fs_region_t

QuRT memory FS region type.

22.44.3.6 typedef unsigned int qurt_mem_pool_t

QuRT emory pool type.

22.44.3.7 typedef unsigned int qurt_size_t

QuRT size type.

22.44.4 Enumeration Type Documentation

22.44.4.1 enum qurt_mem_mapping_t

QuRT Memory region mapping type.

Enumerator:

QURT_MEM_MAPPING_VIRTUAL Default mode. The region virtual address range is mapped to an available contiguous area of physical memory. The base address in physical memory is chosen by the QuRT system. This makes the most efficient use of virtual memory, and works for most memory use cases.

QURT_MEM_MAPPING_PHYS_CONTIGUOUS The region virtual address space must be mapped to a contiguous area of physical memory. This is necessary when the memory region is accessed by external devices that bypass Hexagon virtual memory addressing. The base address in physical memory must be explicitly specified.

QURT_MEM_MAPPING_IDEMPOTENT The region virtual address space is mapped to the identical area of physical memory.

QURT_MEM_MAPPING_VIRTUAL_FIXED The region virtual address space is mapped either to the specified area of physical memory or (if no area is specified) to any available physical memory. This mapping is used 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 The system chooses a random virtual address and maps it to available contiguous physical addresses.

QURT_MEM_MAPPING_INVALID Reserved as an invalid mapping type.

22.44.4.2 enum qurt_mem_cache_mode_t

QuRT cache mode type.

Enumerator:

QURT_MEM_CACHE_WRITEBACK

QURT_MEM_CACHE_NONE_SHARED Normal uncached memory that can be shared with other subsystems.

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_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_INVALID Reserved as an invalid cache type.

22.44.4.3 enum qurt_perm_t

Memory access permission.

Enumerator:

QURT_PERM_READ
QURT_PERM_WRITE
QURT_PERM_EXECUTE
QURT_PERM_FULL

22.44.4.4 enum qurt_mem_cache_type_t

QuRT cache type; specifies data cache or instruction cache.

Enumerator:

QURT_MEM_ICACHE
QURT_MEM_DCACHE

22.44.4.5 enum qurt_mem_cache_op_t

QuRT cache operation code type.

Enumerator:

QURT_MEM_CACHE_FLUSH
QURT_MEM_CACHE_INVALIDATE
QURT_MEM_CACHE_FLUSH_INVALIDATE
QURT_MEM_CACHE_FLUSH_ALL
QURT_MEM_CACHE_FLUSH_INVALIDATE_ALL
QURT_MEM_CACHE_TABLE_FLUSH_INVALIDATE

22.44.4.6 enum qurt_mem_region_type_t

QuRT memory region type.

Enumerator:

QURT_MEM_REGION_LOCAL
QURT_MEM_REGION_SHARED
QURT_MEM_REGION_USER_ACCESS

QURT_MEM_REGION_FS
QURT_MEM_REGION_INVALID Reserved as an invalid region type.

22.44.4.7 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.

22.44.4.8 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.

22.44.5 Variable Documentation

22.44.5.1 qurt_mem_pool_t qurt_mem_default_pool

Memory pool object.

23 System Environment

Programs can access various properties of the QuRT system environment.

The system environment supports the following operations:

- Get the program heap - Return information on the program heap.
- Get the hardware timer - Return the memory address of the hardware timer.
- Get the architecture version - Return the Hexagon processor architecture version.
- Get the maximum hardware threads - Return the number of hardware threads supported in the Hexagon processor.
- Get the maximum pimutex priority - Return the maximum priority of the priority inheritance mutexes.
- Get the API version - Return the release version of the current QuRT RTOS.
- Get the process names - Return the names of processes in the system.
- Get the stack profile count - Return the stack profile count.

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 which are not using pimutexes to run with a thread priority higher than the pimutex maximum priority.

Functions
System environment services are accessed with the following QuRT functions.

- qurt_sysenv_get_app_heap()
- qurt_sysenv_get_arch_version()
- qurt_sysenv_get_hw_timer()
- qurt_sysenv_get_max_hw_threads()
- qurt_sysenv_get_max_pi_prio()
- qurt_sysenv_get_process_name()
- qurt_sysenv_get_stack_profile_count()
- 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_hw_timer()

23.3.1 Function Documentation

23.3.1.1 int qurt_sysenv_get_hw_timer (qurt_sysenv_hw_timer_t * *timer*)

Gets the memory address of the hardware timer from the kernel.

Associated data types

qurt_sysenv_hw_timer_t

Parameters

out	*timer*	Pointer to the memory address of the hardware timer.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.4 qurt_sysenv_get_max_hw_threads()

23.4.1 Function Documentation

23.4.1.1 int qurt_sysenv_get_max_hw_threads (qurt_sysenv_max_hthreads_t * *mhwt*)

Gets the number of hardware threads supported in the Hexagon processor from the kernel.

Associated data types

qurt_sysenv_max_hthreads_t

Parameters

out	*mhwt*	Pointer to the number of hardware threads supported in the Hexagon processor.

Returns

QURT_EOK – Success.
QURT_EVAL – Invalid parameter.

Dependencies

None.

23.5 qurt_sysenv_get_max_pi_prio()

23.5.1 Function Documentation

23.5.1.1 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.

23.6 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 Data Types

This section describes data types for system environment services.

23.8.1 Data Structure Documentation

23.8.1.1 struct qurt_sysenv_swap_pools_t

QuRT swap pool information type.

23.8.1.2 struct qurt_sysenv_app_heap_t

QuRT application heap information type.

23.8.1.3 struct qurt_arch_version_t

QuRT architecture version information type.

23.8.1.4 struct qurt_sysenv_max_hthreads_t

QuRT maximum hardware threads information type.

23.8.1.5 struct qurt_sysenv_max_pi_prio_t

QuRT maximum pi priority information type.

23.8.1.6 struct qurt_sysenv_hw_timer_t

23.8.1.7 struct qurt_sysenv_procname_t

QuRT process name information type.

23.8.1.8 struct qurt_sysenv_stack_profile_count_t

QuRT stack profile count information type.

23.8.1.9 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.

23.8.1.10 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. The collected data can be used to determine the CPU utilization of a QuRT thread (or the entire QuRT user program system).

Profiling is performed with the following operations:

- Enable profiling – Enable counting of running and idle processor cycles.
- Reset profile thread ID processor cycles – Set the per-hardware-thread running processor cycle counts to zero for the specified QuRT thread.
- Get profile thread ID processor cycles – Get the current per-hardware-thread running processor cycle counts for the specified QuRT thread.
- Get profile thread processor cycles – Get the current running processor cycle count for the current QuRT thread.
- Get profile thread thread cycles – Get the current running thread cycle count for the current QuRT thread.
- Reset profile idle processor cycles – Set the per-hardware-thread idle processor cycle counts to zero.
- Get profile idle processor cycles – Get the current per-hardware-thread idle processor cycle counts.
- Get core processor cycles – Get the current running processor cycle count since the processor was reset.

Profiling supports thread-specific cycle counting for both the running (i.e., executing) and idle (i.e., not executing) cycles. The counts can be reset, which 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 (i.e, the number of actual processor cycles executed by all hardware threads) as opposed to thread cycles (i.e., the number of cycles executed by a specific hardware thread). Assuming six hardware threads, the relation between these two cycle types is expressed in the following equation:

```
thread_cycles = processor_cycles / 6
```

The enable profiling operation is used to selectively enable or disable profiling (which is disabled by default).

Note: Enabling profiling does not automatically reset the cycle counts – this must be done explicitly by calling the reset operations before starting cycle counting.

The get profile thread ID processor cycles operation returns the current per-hardware thread running cycle counts for the specified QuRT thread (Section 4). 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.

The get profile thread processor cycles operation returns the current running cycle count for the specified QuRT thread (Section 4). 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.

The get profile thread cycles operation returns the current running cycle counts for the current QuRT thread, expressed in terms of thread cycles.

The get profile idle processor cycles operation returns the current idle cycle count (i.e., 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 get classified as idle or running according to the state that the current thread was in (i.e., idle or running) when the transition to the kernel occurred.

The get core processor cycles operation 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 which differs from the system clock described in Section 17).

In a given 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 total_PCYCLES is the value returned by the get core processor cycles operation.

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;
```

Because the cycle counts are summed on a per-thread basis, it is necessary in the above code to convert each processor cycle count to a thread cycle count (by dividing by 6).

Note: Because the hardware core clock stops running when the Hexagon processor shuts down (due to all its hardware threads being idle), the cycle values returned by this operation should be treated 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 were executed in a given period of time by the Hexagon processor 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 the get profile thread processor cycles operation (described above).

total_PCYCLES is the value returned by the get core processor cycles operation (also described above).

Note, however, that the Hexagon processor may 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, total_PCYCLES must be adjusted by the core clock shutdown time. The shutdown time (also called the ALL_WAIT period) can be computed 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 17).

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 need to be corrected because the weight of each sample is different.

For more information on profiling QuRT threads, see Appendix A.

Functions
Profiling services are accessed with the following QuRT functions.

- qurt_get_core_pcycles()
- qurt_profile_enable()
- qurt_profile_get_idle_pcycles()
- qurt_profile_get_thread_pcycles()
- qurt_profile_get_thread_tcycles()
- qurt_profile_get_threadid_pcycles()
- qurt_profile_reset_idle_pcycles()
- qurt_profile_reset_threadid_pcycles()

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), the cycle values returned by this operation should be treated as relative rather than absolute.

Thread cycle counts are valid only in the V4 Hexagon processor version.

Returns

Integer – Current 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 – this 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_get_idle_pcycles()

24.3.1 Function Documentation

24.3.1.1 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.

24.4 qurt_profile_get_thread_pcycles()

24.4.1 Function Documentation

24.4.1.1 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.

24.5 qurt_profile_get_thread_tcycles()

24.5.1 Function Documentation

24.5.1.1 unsigned long long int qurt_profile_get_thread_tcycles (void)

Gets the count of running thread cycles for the current thread.

Returns the current running thread cycle count for the current QuRT thread.

Returns

Integer – Running thread cycle count for current thread.

Dependencies

None.

24.6 qurt_profile_get_threadid_pcycles()

24.6.1 Function Documentation

24.6.1.1 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.

24.7 qurt_profile_reset_idle_pcycles()

24.7.1 Function Documentation

24.7.1.1 void qurt_profile_reset_idle_pcycles (void)

Sets the per-hardware-thread idle cycle counts to zero.

Returns

None.

Dependencies

None.

24.8 qurt_profile_reset_threadid_pcycles()

24.8.1 Function Documentation

24.8.1.1 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.

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:

- Enable performance monitor – Enable the performance monitor unit.
- Get PMU register – Get the current value of the specified PMU register.
- Set PMU register – Set the value of the specified PMU register.

Functions
Performance monitor services are accessed with the following QuRT functions.

- qurt_pmu_enable()
- qurt_pmu_get()
- qurt_pmu_set()

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 performance monitor unit (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 • #QURT_PMUSTID0 • #QURT_PMUSTID1

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 • #QURT_PMUSTID0 • #QURT_PMUSTID1
in	*reg_value*	Register value.

Returns

None.

Dependencies

None.

26 Error Results

26.1 Overview

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.

26.1.1 Define Documentation

26.1.1.1 #define QURT_EOK 0

Operation was successfully performed.

26.1.1.2 #define QURT_EVAL 1

Wrong values for the parameters. The specified page does not exist.

26.1.1.3 #define QURT_EMEM 2

Not enough memory to perform the operation.

26.1.1.4 #define QURT_EINVALID 4

Invalid argument value; invalid key.

26.1.1.5 #define QURT_EFAILED 12

Operation failed.

26.1.1.6 #define QURT_ENOTALLOWED 13

Operation not allowed.

26.1.1.7 #define QURT_ETLSAVAIL 23

No free TLS key is available.

26.1.1.8 #define QURT_ETLSENTRY 24

TLS key is not already free.

26.1.1.9 #define QURT_EINT 26

Invalid interrupt number (not registered).

26.1.1.10 #define QURT_ESIG 27

Invalid signal bitmask (cannot set more than one signal at a time).

26.1.1.11 #define QURT_ENOTHREAD 30

Thread no longer exists.

26.1.1.12 #define QURT_EALIGN 32

Not aligned.

26.1.1.13 #define QURT_EDEREGISTERED 33

Interrupt is already deregistered.

26.1.1.14 #define QURT_ECANCEL 37

A cancellable request was cancelled due to the associated process being asked to exit.

26.1.1.15 #define QURT_EFATAL -1

Fatal error.

26.1.1.16 #define QURT_FP_EXCEPTION_ALL 0x1F << 25

26.1.1.17 #define QURT_FP_EXCEPTION_INEXACT 0x1 << 29

26.1.1.18 #define QURT_FP_EXCEPTION_UNDERFLOW 0x1 << 28

26.1.1.19 #define QURT_FP_EXCEPTION_OVERFLOW 0x1 << 27

26.1.1.20 #define QURT_FP_EXCEPTION_DIVIDE0 0x1 << 26

26.1.1.21 #define QURT_FP_EXCEPTION_INVALID 0x1 << 25

27 Function Tracing

27.1 Overview

QuRT supports function tracing to assist in debugging programs.

- qurt_trace_changed()
- qurt_trace_get_marker()
- Macros

27.2 qurt_trace_changed()

27.2.1 Function Documentation

27.2.1.1 int qurt_trace_changed (unsigned int *prev_trace_marker*, unsigned int *trace_mask*)

Determines whether specific kernel events have occurred.

Returns a value indicating whether the specified kernel events have been 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: This function is used with qurt_trace_get_marker to determine 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 indicating the 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.

27.3 qurt_trace_get_marker()

27.3.1 Function Documentation

27.3.1.1 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 various kernel events.

Note: This function is used with qurt_trace_changed() to determine whether certain kernel events occurred in a block of code.

Returns

Integer – Kernel trace marker.

Dependencies

None.

27.4 Macros

This section describes macros for function tracing services.

27.4.1 Define Documentation

27.4.1.1 #define QURT_TRACE(*str, ...*) __VA_ARGS__

Function tracing is implemented with a debug macro (QURT_TRACE), which optionally generates printf statements both before and after every function call that is passed as a macro argument.

For example, the following macro call 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 itself.

The debug output is generated using the library function qurt_printf. The symbol QURT_DEBUG 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 QuRT Callbacks

28.1 Overview

The QuRT RTOS defines a callback function that enables users to perform program-specific operations during certain QuRT system events.

Note: These callbacks are invoked only if their symbol names are defined as functions in the program code.

- __hexagon_bsp_init()

28.2 __hexagon_bsp_init()

void __hexagon_bsp_init(void)

Initializes the Hexagon processor.

__hexagon_bsp_init is called by the QuRT boot code during system startup. It enables the program system to perform program-specific system initialization. The callback function has the following properties:

- It must be implemented in assembly language.
- It can access device memory (I/O registers) only through physical addresses.
- No stack is available.
- Registers R24-27 and R31 are callee-saved registers.
- Register R31 contains the function return address.
- It executes in Supervisor mode. The function is called before the Hexagon memory management unit (MMU) is enabled, and before the kernel is started.

Note: __hexagon_bsp_init is invoked only if its symbol name is defined as a function in the program code.

The function code has the following form:

```
.falign
.global __hexagon_bsp_init
.type __hexagon_bsp_init, @function
// function assembly code goes here
jumpr r31
.size __hexagon_bsp_init, .-__hexagon_bsp_init
```

Parameters

None.

Returns

None.

Dependencies

None.

29 Predefined Symbols

29.1 Overview

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.

The symbol can be used 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 (e.g., 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.

29.1.1 Define Documentation

29.1.1.1 #define QURT_API_VERSION 11

QURT API version.

A Thread-level Profiling

A.1 Overview

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.2 Server Behavior

The server receives and processes the following events:

- Start command from client
- Timer expiry
- Stop command from client

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 16) with the period specified by the sampling period received from the start command.
- Enable QuRT profiling using qurt_profile_enable(1).

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.

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.3 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.4 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 References

B.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

B.2 Acronyms and Terms

Acronym or term	Definition
API	application programming interface
Application	A category of user program (multimedia, modem firmware, modem software).
Barrier	QuRT object used to synchronize threads so they meet at a specific point in the program.
BSP	board support package
Cache	Memory subsystem, which stores frequently accessed code or data.
Call tracing	Debug feature, which generates a list of all function calls performed while executing the target application system.
Condition Variable	QuRT object used to synchronize threads based on the value of a data item.
EBI	external bus interface (memory type)
Exception	Special condition that changes the normal flow of program execution.
ISDB	in-silicon debugger
Edge-triggered	Interrupt triggered by a rising or falling transition on the interrupt request line.
Interrupt	Externally generated processor event, which interrupts the normal flow of program control.
IST	interrupt service thread
Kernel	Library, which implements the core QuRT system operations (including thread and memory management).
L2VIC	vector interrupt controller – interrupt system used in the V5 Hexagon processor.
Level-triggered	Interrupt triggered by a high or low level on the interrupt request line.
Lock	See Mutex.
LPM	low-power memory (memory type)
MMU	memory management unit
Mutex	QuRT object used to provide a thread with exclusive access to a resource shared with other threads (short for mutual exclusion).
NMI	non-maskable interrupt
Object	User-created instance of an arbitrary QuRT service.

Acronym or term	Definition
Pipe	QuRT object used to perform synchronized data exchange between threads.
PMU	performance monitor unit – Hexagon processor feature used to measure code performance
Polarity	Whether a signal is defined to be active on a high or low level.
Priority	User-defined thread attribute used to prioritize thread execution.
Process	A grouping of an executable program, an address space, and one or more threads.
QDI	Set of facilities, which support the implementation of device drivers in the QuRT system (short for QuRT driver invocation)
QuRT	Real time operating system for the Hexagon processor.
RTOS	real-time operating system
SMI	stack memory interface
Semaphore	QuRT object used to synchronize threads to restrict access to shared resources.
Signal	QuRT object used to synchronize threads on sets of mutex-like signals.
SSR	supervisor status register
TCB	task control block – Kernel data structure for storing thread state
TCM	tightly coupled memory (memory type)
Thread	Sequence of instructions, which can execute in parallel with other threads (short for thread of execution).
Thread local storage	RTOS feature, which supports the allocation of global storage that is private to a given thread.
TID	trace identifier – Numeric identifier used to trace a thread during hardware debugging
TLB	translation lookaside buffer
TLS	thread lock storage
User process	process
User program	A complete program, which makes calls to the QuRT API to perform various RTOS operations.
VMA	virtual memory area