ORKID

Open Real-Time Kernel Interface Definition

Drafted by

The ORKID Workig Group Software

Subcommittee of VITA

Draft 1.0 for Public Comments

July 1989

Copyright 1989 by VITA, the VMEbus International Trade Association

No parts of this document may be reproduced or used in any form or any means — electronic, graphic, mechanical, electrical or chemical, photocopying, recording in any medium, taping by any computer or storage system etc without prior permission in writing from VITA, the VMEbus International Trade Association.

*Exception: This document may be reproduced or multiplied by photocopying for the exclusive purpose of soliciting public comments on the draft.*

FROM THE CHAIRMAN

Before you lies the first draft of VITA's Open Real Time Interface Definition, known as ORKID. This draft is the result of the activities of a small working group under the auspices of the Software Subcommittee of the VITA Technical Committee. It represents the view of the working group and has not yet been approved.

The working group invites you to check this draft for consistency and send in any comments and/or suggestions you may have to the working group's secretary. All comments received before September 15th, 1989 will be studied by the working group, after which a final draft will be presented to the Software Subcommittee and the Technical Committee for approval.

The members of the working group are:

Reed Cardoza Eyring Research

Alfred Chao Software Components

Chris Eck CERN

Wayne Fischer FORCE Computers

John Fogelin Wind River Systems

Zoltan Hunor VITA Europe (secretary)

Kim Kempf Microware

Hugh Maaskant Philips (chairman)

Dick Vanderlin Motorola

I would like to thank these members for their efforts. Also I would like to thank the companies they represent for providing the time and expenses of these members. Without that support this draft would not have been possible. Furthermore I would like to thank Stuart Fairful for writing up a first version of this draft.

FOREWORD

The objective of the ORKID standard is to provide a state of the art open real—time kernel interface definition that on one hand allows users to create robust and portable code, while on the other hand allowing implementors the freedom to profilate their compliant product. Borderline conditions are that the standard:

- be implementable efficiently on a wide range of microprocessors,

- imposes no unnecessary hardware or software architecture,

- be open to future developments.

Many existing kernel products have been studied to gain insight in the required functionality. As a result ORKID is, from a functional point of view, a blend of these kernels. No radical new concepts have been introduced because there would be no reasonable guarantee that these could be implemented efficiently. Also they would reduce the likelihood of acceptance in the user community. This is not to say that the functionality is meagre, on the contrary: a rich set of objects and operations has been provided.

One issue has to be addressed yet: that of MMU support. Clearly, now that new microprocessors have integrated MMUs\*and hence the cost and performance penalties of MMU support are diminishing, it will be needed in the near future. At this moment, however, it was felt that more experience is needed with MMUs in real-time environments to define a standard. It is foreseen that an addendum to this standard will address MMU support.

TABLE OF CONENTS (auto generate here)

1. INTRODUCTION

ORKID defines a standard programming interface to real-time kernels. This interface consists of a set of standard ORKID operation calls, defining operations on objects of standard types. An ORKID compliant kernel manages these objects and implements the operations.

The application area that ORKID addresses ranges from embedded systems to complex multi-processing systems with dynamic program loading. It is restricted however to real-time environments and only addresses kernel level functionality. As such it addresses a different segment than the real-time extensions to POSIX P1003.4, although some overlaps may occur.

ORKID addresses the issue of multi-processing by defining two levels of compliance: with and without support for multi-node systems. The interfaces to the operations are the same in either level.

Section 2, ORKID PRINCIPLES, contains an introduction to the concepts used in the ORKID standard. Introduced here are the standard ORKID objects and how they are identified, ORKID operations and ORKID multi-processing features. Factors affecting the portability of code developed for ORKID and implementation compliance requirements are also treated here.

Sections 3 to 12 describe in detail the various standard types of object and the operations that manipulate them. There is one section per type of object. Each section contains a general description of this type of object, followed by subsections detailing the operations. The latter are in a programming language independent format. It is foreseen that for all required programming languages, a language binding will be defined in a companion standard. The first one, introduced in conjunction with ORKID, will be for the C language. For syntax, the language binding document is the final authority.

The portability provided by the ORKID standard is at source code level. This means that, optimally, a program written for one implementation should run unmodified on another implementation, requiring only recompilation and relinking. In practice there are many reasons why this might not be true in all cases.

The syntax of ORKID operation calls in a real implementation will be defined in the appropriate language binding. There will be, however, a one to one correspondence between this standard and each language binding for all literal values, operation names and parameter names and types.

2. ORKID CONCEPTS

ORKID defines the interface to a real-time kernel by defining kernel object types and operations upon these objects. Furthermore it assumes an environment, i.e. the computer system, in which these objects exist. This chapter describes that environment, introduces the various object types, explains how objects are identified and defines the structure of the ORKID operation descriptions. Furthermore it addresses the issues of multi-processing and ORKID compatibility.

2.1. Environment

The computer system environment expected by ORKID is described by the notion of a system. A system consists of a collection of one or more interconnected nodes. Each node is a computer with an ORKID compliant kernel on which application programs can be executed. To ORKID a node is a single entity, although it may be implemented as a multi-processor computer there is only one kernel controlling that node.

2.2. ORKID Objects

The standard ORKID object types defined by ORKID are:

* tasks: single threads of program execution in a node.
* regions: memory areas for dynamic allocation of variable sized segments.
* partitions: memory areas for dynamic allocation of fixed sized blocks.
* semaphores: mechanisms used for synchronization and to manage resource allocation amongst tasks.
* queues: inter task communication mechanisms with implied synchronization.
* events: task specific event markers for synchronization.
* exceptions: task specific exceptional conditions with an asynchronous service routine.
* notepad: task specific integer locations for simple, unsynchronized data exchange.
* calendar: current date and time.
* timers: software delays and alarms.

Tasks are the active entities on a node, the CPU(s) of the node execute the task's code, or program, under control of the kernel. Many tasks may exist on a node; they may execute the same or different programs. The maximum number of tasks on a node or in a system is implementation dependent. Tasks compete for CPU time and other resources. Next to tasks interrupt service routines compete for CPU time. Although ORKID does not define how interrupt service routines are activated, it provides facilities to deal with them.

Regions are consecutive chunks of memory from which tasks may allocate segments of varying size for their own purposes. Typically a region consists of memory of one physical nature such as shared RAM, battery

backed-up SRAM etc. The maximum number of regions on a node are implementation dependent.

Partitions are consecutive chunks of memory organized as a pool of fixed sized blocks which tasks may allocate. Partitions are simpler than regions and are intended for fast dynamic memory allocation / de-allocation operations. The maximum number of partitions on a node is implementation dependent.

Semaphores provide a mechanism to synchronize the execution of a task with the execution of another task or interrupt service routine. They can be used to provide sequencing, mutual exclusion and resource management. The maximum number of semaphores on a node or in a system is implementation dependent.

Queues provide a mechanism for intertask communication, allowing tasks to send information to one another with implied synchronization. The maximum number of queues on a node or in a system is implementation dependent.

Events are task specific event markers that allow a task to block until the event, or a specific combination thereof occurs, therefore they form a simple synchronization mechanism. Each task has the same, fixed number of events. The actual number is implementation dependent, but the minimum number is fixed at sixteen.

Exceptions too are tasks specific conditions. Unlike events they are handled asynchronously by the task, meaning that when an exception is raised for a task that task's flow of control is interrupted to execute the code designated to be the exception service routine (XSR). Exceptions are intended to handle exceptional conditions without constantly having to check for them. In general exceptions should not be misused as a synchronization mechanism. Each task has the same, fixed number of exceptions. The actual number is implementation dependent, but the minimum number is fixed at sixteen.

Notepad locations are task specific integer variables that can be read or written without any form of synchronization or protection. Each task has the same, fixed number of notepads. The actual number is implementation dependent, but the minimum number is fixed at sixteen.

The calendar is a mechanism maintaining the current date and time on each node.

Timers come in two forms. The first type of timer is the delay timer that allows a task to delay its execution for a specific amount of time or until a given calendar value. The second type of timer is the event timer. This timer will, upon expiration, sent an event to the task that armed it. As with the delay timer it can expire after a specific amount of time has elapsed or when a given calendar value has passed. The maximum number of timers on a node is implementation dependent, in all cases a delay timer must be available to each task.

2.3. Naming and Identification

Tasks, regions, partitions, semaphores and queues are kernel objects dynamically created and deleted by tasks. when they are created, the task supplies a name for the object and ORKID returns an identifier, which identifies the object in subsequent ORKID operations. The syntax rules for allowable object names is implementation dependent. ORKID does not require uniqueness for object names. Conversely, an object‘s identifier must identify it uniquely within a system.

Observation:

An identifier 's uniqueness may be absolute over time, so that no two objects are ever assigned the same identifier over the lifetime of the system. Alternatively the uniqueness may be guaranteed only at the current time, so that an object may be assigned the same identifier as a previously deleted object. ORKID compliance requires at least uniqueness at the current time

Identifier uniqueness is required only within the set of objects of the same type.

Nodes have no names, but are distinguished by an identifier which must be unique within a system. This standard does not describe how node identifiers are allocated. Two aliases for node identifiers are defined by ORKID: LOCAL\_NODE and OTHER\_NODES. LOCAL\_NODE identifies the node on which the operation is performed. OTHER\_NODE defines the collection of all nodes in the system excluding LOCAL\_NODE.

One or more of a given task's events or exceptions may be specified using a bit­field. Each bit of an event bit­field specifies a single event, likewise for exceptions.

A notepad location is addressed by the combination of the task‘s identifier and an index number, starting at zero.

The calendar has no name or identifier, it is implicitly addressed by the ORKID clock operations.

Timers are created dynamically by user tasks and exist for the duration of their operation. Delay timers have no names or identifiers since they are never accessed once started. Event timers are identified uniquely within a node by a kernel assigned identifier.

2 .4. ORKID Operations

ORKID operations have the form of a function call, taking zero or more input parameters, zero or more output parameters, and returning a completion status. (The operations exception\_return and int\_return are the only two which do not return a completion status as they alter the flow of control.)

Input parameters pass data from the calling program to the kernel, and output parameters pass data from the kernel to the calling program. The physical form which the data takes, and the physical means bywhich it is passed, is implementation and language binding dependent.

The completion status may indicate success, a specific error condition such as an invalid parameter value, or a specific operational condition such as a time-out. When multiple conditions apply, only one status

is returned, defined by an implementation dependent precedence. All statuses have symbolic values the mapping of these symbols to numeric values is implementation dependent.

Each operation interface described in sections 3 to 12 defines a list of possible completion statuses. If the implementing kernel checks for these conditions it must return the appropriate completion status whenever that condition is true. In addition kernels may return statuses not listed in this standard. If the kernel implements no checks it should always return the Value OK. Each implementation must clearly specify which statuses may be returned for each operation. Appendix A gives a list of all defined completion statuses.

Some ORKID operations must be callable from Interrupt Service Routines (ISR) and/or Exception Service Routines (XSR). Kernels may support additional operations from ISRS and/or XSRS. A list of minimum requirements is defined in Appendix B and C.

2.5. Multi-processing

The ORKID standard has been defined to include facilities for multiprocessing. This means that it allows co-operating tasks to run concurrently on more than one processor, while retaining the functionality of ORKID operations. ORKID organizes this using the concepts of node and system.

Nodes

A node is defined as a computing entity addressed by a node identifier and containing a single ORKID data structure.

Systems

A system is defined as a set of one or more connected nodes. There are two basic subdivisions in the way that nodes can be connected within a system:

- A shared memory system consists of a set of nodes connected via

shared memory.

- A non-shared memory system consists of a set of nodes connected by a

network.

The behavior of a networked ORKID implementation should be consistent with the behavior of a shared memory ORKID system. It is also possible to have a mixture of these two schemes where a memory system may contain one or more sets of nodes. These sets of nodes are called shared memory subsystems.

System configuration

This standard does not specify how nodes are configured or how they are assigned identifiers. However, it is recognized that the availability of nodes in a running system can be dynamic. In addition, it is possible but not mandatory that nodes can be added to and deleted from a running system.

Levels of Compliance

ORKID defines two levels of compliance, a kernel may be either single node ORKID compliant or multiple node ORKID compliant. The former type of kernel supports systems with a single node only, while the latter supports systems with multiple nodes.

The syntax of ORKID operation calls does not change with the level of compliance. All 'node' operations must behave sanely in a single node ORKID implementation, i.e. the behavior is that of a multiple node configuration with only one active node.

2 .6. ORKID compatibility

There are several places in this standard where the exact algorithms to be used are defined by the implementor. Although each operation has a defined functionality, the method used to achieve that functionality may cause behavioral differences.

For example, ORKID does not define the kernel scheduling algorithm, especially when several ready tasks have the same priority. This may lead to tasks being scheduled completely differently in different implementations, which may lead to possible different behavior.

Another example is the segment allocation algorithm. Different kernels may handle fragmentation in different ways, leading to cases where one implementation can fulfil a segment request, but another returns an error, since it has left the region more fragmented.

Extensions

Any ORKID compliant implementation can add extensions to give functionality in addition to that defined by this standard. Clearly, a task which uses non-standard extensions is unlikely to be portable to a standard system. In all cases, a kernel which claims compliance to ORKID should have all extensions clearly marked in its documentation.

Undefined Items

There are several items which ORKID does not define but leaves up to the implementation.

ORKID does not define how system or node start-up is accomplished; this will obviously lead to differences in behavior, especially in multi-node systems.

ORKID does not define the word length. On this depends the size of

integer parameters. This latter will be defined in the language binding along with all the other data structures, and so should not cause problems. It is envisaged that ORKID should be scalable in other words it should be implementable on hardware with a different word length without loss of portability.

ORKID does not define the maximum number of events and exceptions per task. The minimum number is sixteen.

ORKID does not define the maximum number of task notepad locations. The minimum number is sixteen.

ORKID does not define the range of priority values.

ORKID defines neither inter-kernel communication methods nor kernel data structure structures. This means that there is no requirement that one implementation must co-operate with other implementations within a system. In general, all the nodes in a system will run the same kernel implementation.

ORKID does not define whether object identifiers need be unique only at the current time, or must be unique throughout the system lifetime. A task which assumes the latter may have problems with an implementation which provides the former.

ORKID does not define the size limits on granularity for regions and block size for partitions.

ORKID does not define any restrictions on the execution of operations within XSRS and interrupt handling routines (ISRS). It does however define a minimum requirement of operations that must be supported.

ORKID defines a number of completion statuses. If an implementation does check for the condition corresponding to one of these statuses, then it must return the appropriate status.

ORKID does not define which completion status will be returned if multiple conditions apply.

ORKID does not define the encoding (binary value) of completions statuses, options and other symbolic values.

ORKID defines a minimum functionality for scheduling task's Exception Service Routines.

2.7. Layout of Operation Descriptions

The remainder of this standard is divided into one section per ORKID object type. Each section contains a detailed description of this type of object, followed by subsections containing descriptions of the relevant ORKID operations.

These operation descriptions are layed out in a formal manner, and contain information under the following headings:

Synopsis

This is a pseudo­language call to the operation giving its standard name and its list of parameters. Note that the language bindings define the actual names which are used for operations and parameters, but the order of the parameters in the call is defined here.

Input Parameters

Those parameters which pass data to the operation are given here in the format:

<parameter name> : <parameter type> Commentary

The actual names to be used for parameters and types are given definitively in the language bindings.

Output Parameters

Those parameters which return data from the operation are given here in the same format as for input parameters. Note that the types given here are simply the types of the data actually passed, and take no account of the mechanism whereby the data arrives back in the calling program. The actual parameter names and types to be used are given definitively in the language bindings.

Literal Values

Under this heading are given literal values which are used with given parameters. They are presented in the following two formats:

<parameter name> = <literal value> Commentary

<parameter name> + <literal value> Commentary

The first format indicates that the parameter is given exactly the indicated literal value if the parameters should affect the function desired in the commentary. The second format indicates that more than one such literal value for this parameter may be combined (logical or) and passed to the operation. If none of the defined conditions is set, the value of the parameter should be zero.

Completion Status

Under this heading are listed all of the possible standard completion statuses that the operation may return.

Description

The last heading contains a description of the functionality of the operation. This-description should not be interpreted as a recipe for implementation

3. TASKS

Tasks are single threads of program execution. within a node, a number of tasks may run concurrently, competing for CPU time and other resources. ORKID does not define the number of tasks allowed per node. Tasks are created and deleted dynamically by existing tasks.

Tasks are allocated CPU time by a part of the kernel called the scheduler. The exact behavior of the scheduler is implementation dependent, but it must have the minimum functionality described in the following paragraphs.

Throughout its existence, each task has a current priority, a current mode and a current state, all of which may change over time. A task may also have an exception service routine which has to be declared to it at runtime.

Task Exception Service Routine

A task may designate an Exception Service Routine (XSR) to handle exceptions which have been sent to that task. A task's XSR can be changed at wili1 hat a task can have only one at any time. The purpose of an XSR is to deal with exceptions which have been sent to the task. It is recommended that exceptions be reserved for errors and other abnormal conditions which arise.

A task's XSR is activated asynchronously. This means that it is not called explicitly by the task code, but automatically by the scheduler whenever one or more exceptions are sent to the task. Thus an XSR may be entered at any time during task execution. (But see 'Task Modes' below.) A task's XSR runs at least at the same priority as the task; it only needs to be executed when the task normally would have been scheduled to the running state. Exceptions are latched on a single level. Multiple occurrences of the same exception during this time will be seen as a single exception by the XSR.

Task Priority

A task's priority determines its 'importance' in relation to the other tasks within the node. Priority is a numeric parameter and can take any value in the range 1 to HIGHP. Priority HIGHP is 'highest' or 'most important' and priority 1 is 'lowest' or 'least important'. There may be any number of tasks with the same priority.

Priorities are assigned to tasks by the tasks themselves, and affect the way in which task scheduling occurs. Although the exact scheduling algorithm is outside the scope of this standard, in general the higher the priority of a task, the more likely it is to receive CPU time.

Task Modes

A mode determines certain aspects of the behavior of the kernel in respect to the task. The mode is made up by the combination of a number of mode parameters, each of which determines a single aspect of kernel behavior.

3.1. TASK\_CREATE

Create a task .

Synopsis

task\_create( name, priority, stack\_size, mode, options, tid )

Input parameters

name : string user defined task name

priority : prio initial task priority

stack\_size : integer size in bytes of task's stack

mode : bit\_field initial task mode

options : bit\_field creation options

Output Parameters

tid : task\_id kernel defined task identifier

Literal Values

mode + NOXSR XSRs cannot be activated

+ NOTERMINATION task cannot be restarted Or deleted

+ NOPREEMPT task cannot be preempted

+ NOINTERRUPT interrupt handling routine cannot be activated

options + GLOBAL New task will be visible throughout the system.

Completion Status

OK task\_create operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_PRIORITY invalid priority value

INVALID\_MODE invalid mode value

INVALID\_OPTIONS invalid options value

TOO\_MANY\_TASKS too many tasks on the node

NO\_MORE\_MEMORY not enough memory to allocate task data structure or task stack

Description

The task\_create operation creates a new task in the kernel data structure. Tasks are always created in the node in which the call to task\_create was made. The new task does not start executing code - this is achieved with a call to the task\_start operation. The tid returned by the kernel is used in all subsequent ORKID operations (except task\_ident) to identify the newly created task. If GLOBAL is specified in the options parameter, then the tid can be used anywhere in the system to identify the task, otherwise it can be used only in the node in which the task was created.

3.2. TASK\_DELETE

Delete a task.

Synopsis

task\_delete( tid )

Input Parameters

tid : task\_id kernel defined task identifier

Output Parameters

<none>

Literal Values

tid = SELF The calling task requests its own deletion.

Completion Status

OK task\_delete operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

OBJECT\_PROTECTED task has NO\_TERMINATION parameter set

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation stops the task identified by the tid parameter and deletes it from its node's kernel data structure. If the task‘s active mode has the parameters NOTERMINATION set, then the task will not be deleted and the completion status OBJECT\_PROTECTED will be returned.

Observation:

The task\_delete operation performs no 'clean-up' of the resources allocated to the task. It is therefore the responsibility of the calling task to ensure that all segments, blocks, etc., allocated to the task to be deleted have been returned.

For situations where one task must delete another, clean­up will usually require co­operation between the tasks, typically using exceptions, or task\_restart.

3.3. TASK\_IDENT

Obtain the identifier of a task on a given node with a given name.

Synopsis

task\_ident( name, nid, tid )

Input Parameters

name : string user defined task name nid : node\_id node identifier

Output Parameters

tid : task\_id kernel defined task identifier

Literal Values

nid = LOCAL\_NODE The node containing the calling task

= OTHER\_NODES all nodes in the system except the local node

name = WHO\_AM\_I Returns tid of calling task

Completion Status

OK task\_ident operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_NODE node does not exist

NAME\_NOT\_FOUND name does not exist on node

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation searches the kernel data structure in the node(s) specified by nid for a task with the given name. If OTHER\_NODES is specified, the node search order is implementation dependent. If there is more than one task with the same name in the node(s) specified, then the tid of the first one found is returned.

3.4. TASK\_START

Start a task.

Synopsis

task\_start( tid, start\_addr, arguments )

Input Parameters

tid : task\_id kernel defined task identifier

start\_addr : \* task start address

arguments : \* arguments passed to task

Output Parameters

<none>

Completion Status

OK task\_start operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

INVALID\_ADDRESS invalid start address

INVALID\_ARGUMENTS invalid number or type or size of arguments

TASK\_ALREADY\_STARTED task has been started already

OBJECT\_PROTECTED task has NOTERMINATION parameter set

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

The task\_start operation starts a task at the given address. The task must have been previously created with the task\_create operation. The task is started with the priority and mode specified when the task was created.

\* The specification of start address and the number and type of arguments are language binding dependent. For a high level language, the start address will likely be the name of a procedure and the arguments would be passed to the procedure as parameters.

3.5. TASK\_RESTART

Restart a task.

Synopsis

task\_restart( tid, arguments )

Input Parameters

tid : task\_id kernel defined identifier

arguments : \* arguments passed to task

Output Parameters

<none>

Literal Values

tid = SELF The calling task restarts itself

Completion Status

OK task\_restart operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

INVALID\_ARGUMENTS invalid number or type or size of arguments

TASK\_NOT\_STARTED task has not yet been started

OBJECT\_PROTECTED task has NOTERMINATION parameter set

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

The task\_restart operation interrupts the current thread of execution of the specified task and forces the task to restart at the address given in the task\_start call which originally started the task. The stack pointer is reset to its original value. No assumption can be made about the original content of the stack at this time.

Any resources allocated to the task are not affected during the task\_restart operation. The tasks themselves are responsible for the proper management of such resources through task\_restart.

If the task's active mode has the parameter NOTERMINATION set, then the task will not be restarted and the completion status OBJECT\_PROTECTED will be returned.

\* The specification of the number and type of the arguments is language binding dependent. For a high level language, it is likely that these arguments will be passed as parameters to the procedure whose name was given as start address in the original task\_start call.

3.6. TASK\_SUSPEND

Suspend a task.

Synopsis

task\_suspend( tid )

Input Parameters

tid : task\_id kernel defined task identifier

Output Parameters

<none>

Literal Values

tid = SELF The calling task suspends itself

Completion Status

OK task\_suspend operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

OBJECT\_PROTECTED task has NOPREEMPT parameter set

TASK\_ALREADY\_SUSPENDED task already suspended

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation temporarily suspends the specified task until the suspension is lifted by a call to task\_resume. While it is suspended, a task cannot be scheduled to run.

If the task's active mode has the parameter NOPREEMPT set the operation will fail and return the completions status OBJECT\_PROTECTED, unless the task suspends itself. In which case the operation will always be successful.

3.7. TASK\_RESUME

Resume a suspended task.

Synopsis

task\_resume( tid )

Input Parameters

tid : task\_id kernel defined task identifier

Output Parameters

<none>

Completion Status

OK task\_resume operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

TASK\_NOT\_SUSPENDED task not suspended

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

The task\_resume operation lifts the task’s suspension immediately after the point at which it was suspended. The task must have been suspended with a call to the task\_suspend operation.

3.8. TASK\_SET\_PRIORITY

Set priority of a task.

Synopsis

task\_set\_priority( tid, new\_prio, old\_prio)

Input Parameters

tid : task\_id kernel defined task id

new\_prio : prio task’s new priority

Output Parameters

old\_prio : prio task’s previous priority

Literal Values

tid = SELF The calling task sets its own priority

new\_prio = CURRENT There will be no change in priority

Completion Status

OK task\_set\_priority operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

INVALID\_PRIORITY invalid priority value

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation sets the priority of the specified task to new\_prio. The new\_prio parameter is specified as CURRENT if the calling task merely wishes to find out the current value of the specified task's priority. ( see also 3. Task Priority )

3.9. TASK\_SET\_MODE

Set mode of own task.

Synopsis

task\_set\_mode( new\_mode, mask, old\_mode )

Input Parameters

new\_mode : bit\_field new task mode settings

mask : bit\_field significant bits in mode

Output Parameters

old\_mode : bit\_field task's previous mode

Literal Values

new\_mode + NOXSR XSRs cannot be activated

+ NOTERMINATION task cannot be restarted or deleted

+ NOPREEMPT task cannot be preempted

+ NOINTERRUPT interrupt handling routine cannot be activated

old\_mode + NOXSR XSRs cannot be activated

+ NOTERMINATION task cannot be restarted or deleted

+ NOPREEMPT task cannot be preempted

+ NOINTERRUPT interrupt handling routine cannot be activated

mask (same as mode)

Completion Status

OK task\_set\_mode operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_MODE invalid mode or mask value

Description

This operation sets a new active mode for the task or its XSR. If called from a task's XSR then the XSR mode is changed, otherwise the main task's mode is changed.

The mode parameters which are to be changed are given in mask. If a parameter is to be set then it is also given in mode, otherwise it is left out. For both mask and mode, the logical OR (!) of the symbolic values for the mode parameters are passed to the operation.

For example, to clear NOINTERRUPT and set NOPREEMPT, mask = NOINTERRUPT! NOPREEMPT, and mode = NOPREEMPT. To return the current mode without altering it, the mask should simply be set to zero. ( see also 3. Task Modes )

3.10. TASK\_READ\_NOTE\_PAD

Read one of a task's note-pad locations.

Synopsis

task\_read\_note\_pad( tid, loc\_number, loc\_value )

Input Parameters

tid : task\_id kernel defined task id

loc\_number : lnum note-pad location number

Output Parameters

loc\_value : integer note-pad location value

Literal Values

tid = SELF The calling task reads its own notepad

Completion Status

OK task\_read\_note\_pad operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

INVALID\_LOCATION note-pad number does not exist

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation returns the value contained in the specified notepad location of the task identified by tid. ( see also 3. Task Notepads )

3.11. TASK\_WRITE\_NOTE\_PAD

Write one of a task's note-pad locations.

Synopsis

task\_write\_note\_pad (tid, loc\_number, loc\_value)

Input Parameters

tid : task\_id kernel defined task id

loc\_number : lnum note-pad location number

loc\_value : integer note-pad location value

Output Parameters

<none>

Literal Values

tid = SELF The calling task writes into its own notepad

Completion Status

OK task\_write\_note\_pad operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

INVALID\_LOCATION note-pad number does not exist

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation writes the specified value into the specified notepad location of the task identified by tid. ( see also 3. Task Notepads)

4. REGIONS

A region is an area of memory within a node which is organized by an ORKID compliant kernel into a pool of segments of varying size. The area of memory to become a region is declared to the kernel by a task when the region is created, and is thereafter managed by the kernel until it is explicitly deleted by a task.

Each region has a granularity, defined when the region is created. The actual size of segments allocated is always a multiple of the granularity, although the required segment size is given in bytes.

Once a region has been created, a task is free to claim variable sized segments from it and return them in any order. The kernel will do its best to satisfy all requests for segments, although fragmentation may cause a segment request to be unsuccessful, despite there being more than enough total memory remaining in the region. The memory management algorithms used are implementation dependent.

Regions, as opposed to partitions, tasks, etc., are only locally accessible. In other words, regions cannot be declared global and a task cannot access a region on another node. This does not stop a task from using the memory in a region on another node, for example in an area of memory shared between the nodes, but all claiming of segments must be done by a co-operating task in the appropriate node and the address passed back.

Observation:

Regions are intended to provide the first subdivisions of the physical memory available to a node. These subdivisions may reflect differing physical nature of the memory, giving for example a region of RAM, a region of ROM, a region of shared memory, etc.. Regions may also subdivide memory into areas for different uses, for example a region for kernel use and a region for user task use.

4.1. REGION\_CREATE

Create a region.

Synopsis

region\_create (name, addr, length, granularity, options, rid)

Input Parameters

name : string user defined region name

addr : address start address of the region

length : integer length of region in bytes

granularity : integer allocation granularity in bytes

options : bit\_field region create options

Output Parameters

rid : region\_id kernel defined region identifier

Completion Status

OK region\_create operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ADDRESS area given not within actual memory present

INVALID\_GRANULARITY granularity not supported

INVALID\_OPTIONS invalid options value

TOO\_MANY\_REGIONS too many regions on the node

REGION\_OVERLAP area given overlaps an existing region

Description

This operation declares an area of memory to be organized as a region by the kernel. The process of formatting the memory to operate as a region may require a memory overhead which may be taken from the new region itself. It can never be assumed that all of the memory in the region will be available for allocation. The overhead percentage will be implementation dependent.

Observation:

Currently ORKID defines no options, the parameter is there as a place holder for future extensions and implementations desiring to provide additional options.

4.2. REGION\_DELETE

Delete a region.

Synopsis

region\_delete( rid, options )

Input Parameters

rid : region\_id kernel defined region identifier

options : bit\_field region deletion options

Output Parameters

<none>

Literal Values

Options + FORCED\_DELETE deletion will go ahead even though there are unreleased segments

Completion Status

OK region\_delete operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID region does not exist

OBJECT\_DELETED region specified has been deleted

INVALID\_OPTIONS invalid options value

REGION\_IN\_USE segments from this region are still allocated

Description

Unless the FORCED\_DELETE option was specified, this operation first checks whether the region has any segments which have not been returned. If this is the case, then the REGION\_IN\_USE completion status is returned. If not, and in any case if FORCED\_DELETE was specified, then the region is deleted from the kernel data structure.

4.3. REGION\_IDENT

Obtain the identifier of a region with a given name.

Synopsis

region\_ident( name, rid )

Input Parameters

name : string user defined region name

Output Parameters

rid : region\_id kernel defined region identifier

Completion Status

OK region\_ident operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal node

NAME\_NOT\_FOUND name does not exist on node

Description

This operation searches the kernel data structure in the local node for a region with the given name, and returns its identifier if found. If there is more than one region with the same name, the kernel will return the identifier of one of them, the choice being implementation dependent.

4.4. REGION\_GET\_SEG

Get a segment from a region.

Synopsis

region\_get\_seg( rid, seg\_size, seg\_addr )

Input Parameters

rid : region\_id kernel defined region id

seg\_size : integer requested segment size in bytes

Output Parameters

seg\_addr : address address of obtained segment

Completion Status

OK region\_get\_seg operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID region does not exist

OBJECT\_DELETED region specified has been deleted

NO\_MORE\_MEMORY not enough contiguous memory in the region to allocate segment of requested size

Description

The region\_get\_seg operation is a request for a given sized segment from a given region's free memory pool. If the kernel cannot fulfil the request immediately, it returns the error completion status NO\_MORE\_MEMORY, otherwise the address of the allocated segment is returned. The allocation algorithm is implementation dependent.

Note that the actual size of the segment returned will be more than the size requested, if the latter is not a multiple of the region’s granularity.

4.5. REGION\_RET\_SEG

Return a segment to its region.

Synopsis

region\_ret\_seg( rid, seg\_addr )

Input Parameters

rid : region\_id kernel defined region id

seg\_addr : address address of segment to be returned

Output Parameters

<none>

Completion Status

OK region\_ret\_seg operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID region does not exist

OBJECT\_DELETED region specified has been deleted

INVALID\_SEGMENT no segment allocated from this region at seg\_addr

Description

This operation returns the given segment to the given region's free memory pool. The kernel checks that this segment was previously allocated from this region, and returns INVALID\_SEGMENT if it wasn’t.

4.6. REGION\_INFO

Obtain information on a region.

Synopsis

region\_info( rid, size, max\_segment, granularity )

Input Parameters

rid : region\_id kernel defined region id

Output Parameters

size : integer length in bytes of overall area in region available for segment allocation

max\_segment : integer length in bytes of maximum segment allocatable at time of call

granularity : integer allocation granularity in bytes

Completion Status

OK region\_info operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID region does not exist

OBJECT\_DELETED region specified has been deleted

Description

This operation provides information on the specified region. It returns the size of the region's area for segment allocation, which may be smaller than the region length given in region\_create due to a possible formatting overhead. It returns also the size of the biggest segment allocatable from the region. This value should be used with care as it is just a snap­-shot of the region's usage at the time of executing the operation. Finally it returns the region’s allocatable granularity.

5. PARTITIONS

Partitions are areas of memory organized by the kernel as a pool of fixed size blocks. As for regions, the creating task supplies the area of memory to be used by the partition. The task also supplies the size of the blocks to be allocated from the partition. Any restrictions imposed on the block size are implementation dependent.

Partitions are simpler structures than regions, and are intended for use where speed of allocation is essential. Partitions may also be declared global, and be operated on from more than one node. However, this makes sense only if the nodes accessing the partition are all in the same shared memory system, and the partition is in shared memory.

Once the partition created, tasks may request blocks one at a time from it, and can return them in any order. Because the blocks are all the same size, there is no fragmentation problem in partitions. The exact allocation algorithms are implementation dependent.

5.1. PARTITION\_CREATE

Create a partition.

Synopsis

partition\_create( name, addr, length, block\_size, options, pid )

Input Parameters

name : string user defined partition name

addr : address start address of partition

length : integer length of partition in bytes

block\_size : integer partition block size in bytes

options : bit\_field partition create options

Output Parameters

pid : part\_id kernel defined partition identifier

Literal Values

option: + GLOBAL partition is global within the shared memory system

Completion Status

OK partition\_create operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ADDRESS area defined is not within actual memory present

INVALID\_BLOCK\_SIZE block\_size not supported

INVALID\_OPTIONS invalid options value

TOO\_MANY\_PARTITIONS too many partitions on the node

PARTITION\_OVERLAP area given overlaps an existing partition

Description

This operation declares an area of memory to be organized as a partition by the kernel. The process of formatting the memory to operate as a partition may require a memory overhead which may be taken from the new partition. It can never be assumed that all of the memory in the partition will be available for allocation. The overhead percentage will be implementation dependent.

5.2. PARTITION\_DELETE

Delete a partition.

Synopsis

partition\_delete( pid, options )

Input Parameters

pid : part\_id kernel defined partition identifier

options : bit\_field partition deletion options

Output Parameters

<none>

Literal Values

options + FORCED\_DELETE deletion will go ahead even though there are unreleased blocks

Completion Status

OK partition\_delete operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID partition does not exist

OBJECT\_DELETED partition specified has been deleted

INVALID OPTIONS invalid options value

PARTITION\_IN\_USE blocks from this partition are still allocated

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

Unless the FORCED\_DELETE option was specified, this operation first checks whether the partition has any blocks which have not been returned. If this is the case, then the PARTITION\_IN\_USE completion status is returned. If not, and in any case if FORCED\_DELETE was specified, then the partition is deleted from the kernel data structure.

5.3. PARTITION\_IDENT

Obtain the identifier of a partition on a given node with a given name.

Synopsis

partition\_ident( name, nid, pid, )

Input Parameters

name : string user defined partition name

nid : node\_id node identifier

Output Parameters

pid : part\_id kernel defined partition identifier

block\_size : integer the partition's block size

Literal Values

nid = LOCAL\_NODE the node containing the calling task

= OTHER\_NODES all nodes in the system except the local node

Completion Status

OK partition\_ident operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_NODE node does not exist

NAME\_NOT\_FOUND name does not exist on node

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a partition with the given name, and returns its identifier and block size if found. If OTHER\_NODES is specified, the node search order is implementation dependent, but will include only those nodes in the shared memory system or subsystem containing the partition. If there is more than one partition with the same name, then the pid of the first one found is returned.

5.4. PARTITION\_GET\_BLK

Get a block from a partition.

Synopsis

partition\_get\_blk( pid, blk\_addr )

Input Parameters

pid : part\_id kernel defined partition identifier

Output Parameters

blk\_addr : address address of obtained block

Completion Status

OK partition\_get\_blk operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID partition does not exist

OBJECT\_DELETED partition specified has been deleted

NO\_MORE\_MEMORY no more blocks available in partition

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation is a request for a single block from the partition's free block pool. If the kernel cannot immediately fulfil the request, it returns the error completion status NO\_MORE\_MEMORY, otherwise the address of the allocated block is returned. The exact allocation algorithm is implementation dependent.

5.5. PARTITION\_RET\_BLK

Return a block to its partition.

Synopsis

partition\_ret\_blk( pid, blk\_addr )

Input Parameters

pid : part\_id kernel defined partition identifier

blk\_addr : address address of block to be returned

Output Parameters

<none>

Completion Status

OK partition\_ret\_blk operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID partition does not exist

OBJECT\_DELETED partition specified has been deleted

INVALID\_BLOCK no block allocated from partition at blk\_addr

NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation returns the given block to the given partition's free block pool. The kernel checks that the block was previously allocated from the partition and returns INVALID\_BLOCK if it wasn't.

5.6. PARTITION\_INFO

Obtain information on a partition.

Synopsis

partition\_info( pid, blocks, free\_blocks, block\_size )

Input Parameters

pid : partition-id kernel defined region id

Output Parameters

blocks : integer number of blocks in the partition

free\_blocks : integer number of free blocks in the partition

block\_size : integer partition block size in bytes

Completion Status

OK partition\_info operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID partition does not exist

OBJECT\_DELETED partition specified has been deleted

NODE\_NOT\_REACHABLE node on which the partition resides is not reachable

Description

This operation provides information on the specified partition. It returns its overall number of blocks, the number of free blocks in the partition, and the block size. The number of free blocks in the partition should be used with care as it is just a snap-shot of the partitions's usage at the time of executing the operation.

6. SEMAPHORES

The semaphores defined in ORKID are standard Dijkstra counting semaphores. Semaphores provide for the fundamental need of synchronization in multi-tasking systems, i.e. mutual exclusion, resource management and sequencing.

Semaphore Behavior

The following should not be understood as a recipe for implementations.

The behavior of counting semaphores can be described as follows:

During a sem\_p operation, the semaphore count is decremented by one. If the resulting semaphore count is greater than or equal to zero, than the calling task continues to execute. If the count is less than zero, the task blocks from CPU usage and is put on a waiting list for the semaphore.

During a sem\_v operation, the semaphore count is incremented by one. If the resulting semaphore count is less than or equal to zero then the first task in the waiting list for this semaphore is unblocked and is made eligible for CPU usage.

Semaphore Usage

Mutual exclusion is achieved by creating a counting semaphore with an initial count of one. A resource is guarded with this semaphore by requiring all operations on the resource to be proceeded by a sem\_p operation. Thus, if one task has claimed a resource, all other tasks requiring the resource will be blocked until the task releases the resource with a sem\_v operation.

In situations where multiple instantiations of a resource exist, the semaphore may be created with an initial count equal to a number of instantiations. A resource is claimed from the pool with the sem\_p operation. When all available copies of the resource have been claimed, a task requiring the resource will be blocked until one of the claimed resources is returned to the pool by a sem\_v operation.

Sequencing is achieved by creating a semaphore with an initial count of zero. A task may pend the arrival of another task by performing a sem\_p operation when it reaches a synchronization point. The other tasks performs a sem\_v operation when it reaches its synchronization point, unblocking the pended task.

Semaphore Options

ORKID defines the following option symbols, which may be combined.

\* GLOBAL Semaphores created with the GLOBAL option set are

visible and accessible from any node in the system.

\* FIFO Semaphores created with the FIFO option set enqueue

blocked tasks in order of arrival of the sem\_p

operations. Without this option, the tasks are enqueued in order of task priority.

6.1. SEM\_CREATE

Create a semaphore.

Synopsis

sem\_create( name, init\_count, options, sid )

Input Parameters

name : string user defined semaphore name

init\_count : integer initial semaphore count

options : bit\_field semaphore create options

Output Parameters

sid : sema\_id kernel defined semaphore identifier

Literal Values

options + GLOBAL the new semaphore will be visible throughout the system

+ FIFO tasks will be queued in first in first out order

Completion Status

OK sem\_create operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_COUNT init count is negative

INVALID\_OPTIONS invalid options value

TOO\_MANY\_SEMAPHORES too many semaphores on node

Description

This operation creates a new semaphore in the kernel data structure, and returns its identifier. The semaphore is created with its counter at the value given by the count parameter. The task queue, initially empty, will be ordered by task priority, unless the FIFO option is set, in which case it will be first in first out.

6.2. SEM\_DELETE

Delete a semaphore.

Synopsis

sem\_delete( sid )

Input Parameters

sid : sema\_id kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK sem\_delete operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID semaphore does not exist

INVALID\_OPTIONS semaphore specified has been deleted

TOO\_MANY\_SEMAPHORES node on which semaphore resides is not reachable

Description

The sem\_delete operation deletes a semaphore from the kernel data structure. The semaphore is deleted immediately, even though there are tasks waiting in its queue. These latter are all unblocked and are returned the SEMAPHORE\_DELETED completion status.

6.3. SEM\_IDENT

Obtain the identifier of a semaphore on a given node with a given name.

Synopsis

sem\_ident( name, nid, sid )

Input Parameters

name : string user defined semaphore name

nid : node\_id node identifier

Output Parameters

sid : sema\_id kernel defined semaphore identifier

Literal Values

nid = LOCAL\_NODE the node containing the calling task

= OTHER\_NODES all nodes in the system except the local node

Completion Status

OK sem\_ident operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_NODE node does not exist

NAME\_NOT\_FOUND name does not exist on node

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a semaphore with the given name, and returns its identifier if found. If OTHER\_NODES is specified, the node search order is implementation dependent. If there is more than one semaphore with the same name in the node(s) specified, then the sid of the first one found is returned.

6.4. SEM\_P

Perform P operation (take) on a semaphore.

Synopsis

sem\_p( sid, options, time\_out )

Input Parameters

sid : sema\_id kernel defined semaphore identifier

options : bit\_field semaphore wait options

time\_out : integer ticks to wait before timing out

Output Parameters

<none>

Literal Values

Options + NOWAIT do not wait - return immediately if

semaphore not available

time\_out = FOREVER wait forever - do not time out

Completion Status

OK sem\_p operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED semaphore specified has been deleted

TIME\_OUT sem\_p operation timed out

SEMAPHORE\_DELETED semaphore deleted while blocked in sem\_p

operation

SEMAPHORE\_NOT\_AVAILABLE semaphore unavailable with NOWAIT option

NODE\_NOT\_REACHABLE node on which semaphore resides is not

reachable

Description

This operation performs a claim from the given semaphore. It first checks if the NOWAIT option has been specified and the counter is zero or less, in which case the SEMAPHORE\_NOT\_AVAIIABLE completion status is returned. Otherwise, the counter is decreased. If the counter is now zero or more, then the claim is successful, otherwise the calling task is put on the semaphore queue.

If the semaphore is deleted while the task is waiting on its queue, then the task is unblocked and this operation returns the SEMAPHORE\_DELETED completion status. Otherwise the task is blocked either until the timeout expires, in which case the TIME\_OUT completion status is returned, or until the task reaches the head of the queue and another task performs a sem\_v operation on this semaphore.

6.5. SEM\_V

Perform a V operation (give) on a semaphore.

Synopsis

sem\_v( sid )

Input Parameters

sid : sema\_id kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK sem\_v operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED semaphore specified has been deleted

SEM\_OVERFLOW the counter of semaphore overflows

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation increments the semaphore count by one. If the resulting semaphore count is less than or equal to zero then the first task in the semaphore queue is unblocked, and returned the successful completion status

6.6. SEM\_INFO

Obtain information on a semaphore.

Synopsis

sem\_info( sid, options, count, tasks\_waiting )

Input Parameters

sid : sem-id kernel defined semaphore identifier

Output Parameters

options : bit\_field semaphore create options

count : integer semaphore count at time of call

tasks\_waiting : integer number of tasks waiting in the semaphore queue

Completion Status

OK sem\_info operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED semaphore specified has been deleted

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation provides information on the specified semaphore. It returns its create options, the value of it's counter, and the number of tasks waiting on the semaphore queue. The latter two values should be used with care as they are just a snap­shot of the semaphores's state at the time of executing the operation.

7. QUEUES

Queues permit the passing of messages amongst tasks. Queues contain a variable number of messages, all of which have the same user task defined length. The queues normally behave first in first out, with messages sent to a queue being appended at the tail, and messages received from a queue being taken from the head. Urgent messages can be inserted at the head of the queue, i.e. they are prepended. Several urgent messages prepended without an intervening receive will be received last in first out.

Queue Behavior

The following should not be understood as a recipe for implementations.

When a queue contains no messages, a task which receives from it is blocked (unless it specified the NOWAIT option) and is put on the queue's wait queue. This queue of waiting tasks is ordered either by task priority or as first in first out.

A task may broadcast a message to all tasks on a wait queue, which unblocks all of them and returns them all the same message. This latter operation is atomic with respect to any other operation on this queue.

All messages in a queue may be flushed with a single operation that is atomic with respect to any other operation on this queue.

Observation:

It can be seen that there is more than one way to use a queue. At one extreme, many tasks feed messages onto a queue and a single task receives them, creating a many to one data flow. At the other extreme, many tasks wait for a message and one task broadcasts a message synchronously to all of them, creating a one to many data flow.

Queue Options

A queue's options are set by the creating task. They define various aspects of the behavior of the kernel with respect to queues. ORKID defines the following option symbols, which may be combined unless otherwise stated. An implementation may define additional options.

- GLOBAL Queues created with the GLOBAL option set are visible and accessible from any node in the system. When a

message is sent to a queue in another node, the message is physically copied to that other node. In non-shared

memory systems, it is not guaranteed that a message has arrived in the destination node before the operation

returns a successful completion status.

- FIFO With this option set, the tasks waiting for messages from the queue will be queued first in first out. The tasks

are by default queued in order of task priority

7.1. QUEUE\_CREATE

Create a message queue.

Synopsis

queue\_create( name, max\_buff, length, options, qid )

Input Parameters

name : string user defined queue name

max\_buff : integer maximum number of buffers allowed in queue

length : integer length of message buffers in bytes

options : bit\_field queue create options

Output Parameters

qid : queue\_id kernel defined queue identifier

Literal Values

options + GLOBAL the new queue will be visible throughout the system

+ FIFO tasks waiting on a message will be queued first in first out

Completion Status

OK queue\_create operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_LENGTH buffer length not supported

INVALID\_OPTIONS invalid options value

TOO\_MANY\_QUEUES too many queues on node

NO\_MORE\_MEMORY not enough memory to allocate message buffer(s)

Description

This operation creates a new queue in the kernel data structure. The given number of buffers of the given length are allocated by the kernel. If the kernel cannot find sufficient memory it returns the NO\_MORE\_MEMORY completion status.

The maximum possible length of messages is implementation dependent, but an ORKID compliant kernel is required to support message lengths of up to 32 bytes.

7.2. QUEUE\_DELETE

Delete an existing queue.

Synopsis

queue\_delete( qid )

Input Parameters

qid : queue\_id kernel defined queue identifier

Output Parameters

<none>

Completion Status

OK queue\_delete operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED specified has been deleted

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This option deletes the given queue from the kernel data structure. If any tasks were waiting for a message from the queue, they are unblocked and returned the QUEUE\_DELETED completion status. If there were any messages in the queue, they are lost and the buffers deallocated.

7.3. QUEUE\_IDENT

Obtain the identifier of a queue on a given node with a given name.

Synopsis

queue\_ident( name, nid, qid )

Input Parameters

name : string user defined queue name

nid : node\_id node identifier

Output Parameters

qid : queue\_id kernel defined queue identifier

Literal Values

nid = LOCAL\_NODE the node containing the calling task

= OTHER\_NODES all nodes in the system except the local node.

Completion Status

OK queue\_ident operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_NODE node does not exist

NAME\_NOT\_FOUND name does not exist on node

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation searches the kernel data structure in the node(s) specified for a queue with the given name, and returns its identifier if found. If OTHER\_NODES is specified, the node Search order is implementation dependent. If there is more than one queue with the same name in the node(s) specified, then the qid of the first one found is returned.

7.4. QUEUE\_SEND

Send a message to a given queue.

Synopsis

queue\_send( qid, message, length )

Input Parameters

qid : queue\_id kernel defined queue identifier

message : address message starting address

length : integer length of message in bytes

Output Parameters

<none>

Completion Status

OK queue\_send operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does exist

OBJECT\_DELETED queue specified has been deleted

INVALID\_LENGTH message length greater than queue’s buffer length

QUEUE\_FULL no more buffers available

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operations sends a message to a queue. If the queue‘s wait queue contains a number of tasks waiting on messages, then the message is delivered to the task at the head of the wait queue. This task is then wait queue, unblocked and will be returned a successful completion status along with the message. Otherwise the message is put on the queue.

If the maximum queue length has been reached, then the QUEUE\_FULL completion status is returned.

7.5. QUEUE\_URGENT

Send a message to head of queue.

Synopsis

queue\_urgent( qid, message, length )

Input Parameters

qid : queue\_id kernel defined queue identifier

message : address message starting address

length : integer message length in bytes

Output Parameters

<none>

Completion Status

OK queue\_broadcast operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED queue specified has been deleted

INVALID\_LENGTH message length greater than queue’s buffer length

QUEUE\_FULL no more buffers available

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation sends a priority message to a queue.

If the queue’s wait queue contains a number of tasks waiting on messages, then the action is exactly the same as for queue send. The message is delivered to the task at the head of the wait queue. This task is then removed from the wait queue, unblocked and will be returned a successful completion status along with the message.

Otherwise the message is inserted at the head of the message queue. If there is no memory available for the buffer, then the NO\_MORE\_MEMORY completion status is returned.

7.6. QUEUE\_BROADCAST

Broadcast message to all tasks blocked on a queue.

Synopsis

queue\_broadcast( qid, message, length, count )

Input Parameters

qid : queue\_id kernel defined queue identifier

message : address message starting address

length : integer message length in bytes

Output Parameters

count : integer number of unblocked tasks

Completion Status

OK queue\_broadcast operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED queue specified has been deleted

INVALID\_LENGTH message length greater than queue’s buffer length

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation sends a message to all tasks waiting on the queue. If the wait queue is empty, then no messages are sent, no tasks are unblocked and the count returned will be zero. If the wait queue contains a number of tasks waiting on messages, then the message is delivered to each task in the wait queue. All tasks are then removed from the wait queue, unblocked and returned a successful completion status. The number of tasks unblocked is returned in the count parameter.

This operations is atomic with respect to other operations on the queue.

7.7. QUEUE\_RECEIVE

Receive a message from a queue.

Synopsis

queue\_receive( qid, message, options, time\_out )

Input Parameters

qid : queue\_id kernel defined queue identifier

message : address address to put message

options : bit\_field queue receive options

time\_out : integer max number of ticks to wait

Output Parameters

<none>

Literal Values

options + NOWAIT do not wait return immediately if no message in queue

time\_out = FOREVER wait forever ~ do not time out

Completion Status

OK queue receive operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED queue specified has been deleted

INVALID\_ADDRESS message refers to an illegal address

INVALID\_OPTIONS invalid options value

TIME\_OUT queue\_receive operation timed out

QUEUE\_DELETED queue deleted while blocked in queue\_receive operation

QUEUE\_EMPTY queue empty with NOWAIT option

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation receives a message from a given queue. If there are one or more messages on the queue, then the buffer at the head is removed from the queue, its message is copied into the given area, the buffer is deallocated, and a successful completion status returned.

If the queue is empty, and NOWAIT was not specified in the options, then the task is blocked and put on the queue's wait queue in order of task priority or first in first out. If NOWAIT was specified and the queue is empty, then the QUEUE\*EMPTY completion status is returned. If the queue is deleted while the task is waiting on a message from it, then the QUEUE\_DELETED completion status is returned. If the

timeout expires, then the TIME\_OUT completion status is returned. Otherwise, when the task reaches the head of the queue and a message is sent, or if a message is broadcast while the task is anywhere in the queue, then the task receives the message and is returned a successful completion status.

7.8. QUEUE\_FLUSH

Flush all messages on a queue.

Synopsis

queue\_flush( qid, count )

Input Parameters

qid : queue\_id kernel defined queue identifier

Output Parameters

count : integer number of flushed messages

Completion Status

OK queue\_flush operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETE.R a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED queue specified has been deleted

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

If there were one or more messages in the specified queue, then they are removed from the queue, their buffers deallocated and their number returned in count. If there were no messages in the queue, then a count of zero is returned.

7.9. QUEUE\_INFO

Obtain information on a queue.

Synopsis

queue\_info( qid, max\_buff, length, options, messages\_waiting

tasks\_waiting )

Input Parameters

qid : queue\_id kernel defined queue identifier

Output Parameters

max\_buff : integer maximum number of buffers in queue

length : integer length of message buffers in bytes

options : bit\_field semaphore create options

tasks\_waiting : integer number of tasks waiting on the message queue

messages\_waiting : integer number of messages waiting in the message queue

Completion Status

OK queue\_info operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID queue does not exist

OBJECT\_DELETED queue specified has been deleted

NODE\_NOT\_REACHABLE node on which the queue resides is not reachable

Description

This operation provides information on the specified message queue. It returns its maximum number of buffers in bytes, its create options, and the number of tasks waiting for messages on this queue, respectively the number of messages waiting in the queue to be read.

The latter two values should be used with care as they are just a snapshot of the semaphores's state at the time of executing the operation.

8. EVENTS

Events provide a simple method of task synchronization. Each task has the same number of events. The maximum number of these is implementation dependent, but the minimum number is fixed at sixteen. Events have no identifiers, but are addressed using a task identifier and a bit-field. A bit-field can indicate any number of a task's events at once.

A task can wait on any combination of its events, requiring either all specified events to arrive, or at least one of them, before being unblocked. Tasks can send any combination of events to a given task. If the receiving task is not in the same node as the sending task, then the receiving task must be global.

Sending events in effect sets a one bit latch for each event. Receiving a combination of events clears the appropriate latches. This means that if an event is sent more than once before being received, the second and subsequent sends are not seen.

8.1. EVENT\_SEND

Send event(s) to a task.

Synopsis

event\_send( tid, event )

Input Parameters

tid : task\_id kernel defined task identifier

event : bit\_field event(s) to be sent

Output Parameters

<none>

Completion Status

OK event\_send operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation sends the given event(s) to the given task. The appropriate task event latches are set. If the task is waiting on a combination of events, a check is made to see if the currently set latches satisfy the requirements. If this is the case, the given task receives the event(s) it is waiting on and the appropriate bits are cleared in the latch.

8.2. EVENT\_RECEIVE

Receive event(s).

Synopsis

event\_receive( events, options, time\_out, events\_received )

Input Parameters

events : bit\_field event(s) to receive

options : bit\_field receive options

time\_out : integer max no of tricks to wait

Output Parameters

events\_received : bit\_field event(s) received

Literal Values

options + ANY return when any of the events is sent

+ NOWAIT do not wait - return immediately if no events set

time\_out = FOREVER wait forever - do not time out

Completion Status

OK event\_receive operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_OPTIONS invalid options value

TIME\_OUT event\_receive operation timed out

NO\_EVENTS event(s) not set and NOWAIT option given

Description

This operation waits on a given combination of events to occur. By default, the operation waits until all of the events have been sent. If the ANY option is set, then the operation waits only until any one of the events has been sent.

The operation first checks the task’s event latches to see if the required event(s) have already been sent. In this case the task receives the events, which are returned in events\_caught, and the appropriate event latches are cleared. If the ANY option was set, and more than one of the specified events was sent, all the events sent, satisfying the events, are received.

If the required event(s) have yet to be sent, and the NOWAIT option has been specified, the NO\_EVENTS completion status is returned. If NOWAIT is not specified then the task is blocked, waiting on the appropriate events to be sent. A timeout is initiated, unless the time\_out value supplied is FOREVER. If all required events are sent before the timeout expires, then the events are received and a successful completion status returned. If the timeout expires, the TIME\_OUT completion status is returned.

9. EXCEPTIONS

ORKID exceptions provide tasks with a method of handling exceptional conditions asynchronously. Each task has the same number of exceptions. The maximum number of these is implementation dependent, but the minimum number is fixed at sixteen. Exceptions have no identifiers, but are addressed using a task identifier and a bit field, which can indicate any number of exceptions at once.

Exceptions are identified in the same manner as events. Using a bit field, any number of exceptions can be raised simultaneously to a task. Raising an exception sets a one bit latch for each exception. If the same exception is raised more than once to a task before the task can catch them, then the second and subsequent raisings are ignored. If the target task is not in the same node as the raising task, then the target task must be global.

The 'catching' of exceptions is quite different than that of events, and involves the activation of the task's Exception Service Routine (XSR). XSRS have to be declared via the exception\_catch operation to tasks after their creation. A task may change its XSR at any time.

An XSR is activated whenever one or more exceptions are raised to a task, and the task has not set its NOXHR modal parameter in the active mode. If the NOXHR parameter is set, the XSR will be activated as soon as it is cleared. When an XSR is activated, the task's current flow of execution is interrupted and the XSR entered. The XSR is passed the bit field indicating which exceptions have been sent as a parameter. The exact way how to accomplish this is defined in the language binding. The XSR always catches all exceptions which have been raised, and all the latches are cleared.

An XSR is treated by the scheduler in exactly the same way as other parts of the task. The kernel automatically activates a task's current XSR as detailed above, but the XSR is actually required to execute only when the task would normally be scheduled to run. The XSR must deactivate and return to the code which it interrupted with a special ORKID operation: EXCEPTION\_RETURN. While it is active, an XSR has no special privileges or restrictions other than those necessitated by its asynchronous execution.

A XSR has its own mode with the same four mode parameters as tasks: NOXSR, NOTERMINATION, NOPREEMPT and NOINTERRUPT. The mode parameter given in the exception\_catch operation is ored with the active mode at the time of the XSR's activation. The XSR will enter execution with this mode, which now becomes the active mode.

An active XSR can itself be interrupted by an exception being raised. In this case, unless the XSR's modal parameter NOXHR was set, the XSR is immediately reentered to handle the new exception. Theoretically, XSR activation can be thus nested to any depth. The kernel only considers the active mode when making scheduling decisions.

9.1. EXCEPTION\_CATCH

Specify a task's asynchronous exception handling routine.

Synopsis

exception\_catch( new\_XSR, mode, old\_XSR, old\_mode )

Input Parameters

new\_XSR : address address of exception handling routine

mode : bit\_field startup execution mode of XSR

Output Parameters

old\_XSR : address address of previous XSR

old\_mode : bit\_field mode associated with old XSR

Literal Values

new\_XSR = NULL\_XSR task henceforth will have no XSR

mode + NOXHR XSR cannot be activated

+ NOTERMINATION task cannot be restarted or deleted

+ NOPREEMPT task cannot be preempted

+ NOINTERRUPT interrupt handling routine cannot be activated

old\_XSR = NULL\_XSR task previously had no XSR

Completion Status

OK exceptions\_catch operation successful

ILLEGAL\_USE operation not callable from ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ADDRESS new\_XSR refers to an illegal address

INVALID\_MODE invalid mode value

Description

This operation designates a new exception handling routine (XSR) for the current task. The task supplies the start address of the XSR, and the mode in which it will be started. If this operation returns a successful completion status, an exception sent to the task will henceforth cause the XSR at the given address to be activated.

The kernel returns the address of the previous XSR and the mode associated with that XSR.

Observation:

This can be used when a task wishes to use a different XSR temporarily. Once finished with the temporary XSR, the original one can be simply reinstated.

Note that if tasks are created without an XSR in a particular

implementation, the first call to exception\_catch will return the symbolic value NULL\_XSR in old\_XSR. This same value can be passed as the new\_XSR input parameter, which removes the current XSR from the task without designating a new one.

9.2. EXCEPTION\_RAISE

Raise exceptions to a task.

Synopsis

exception\_raise( tid, exceptions )

Input Parameters

tid : task\_id kernel defined task id

exceptions : bit\_field exceptions to be raised

Output Parameters

<none>

Completion Status

OK exceptions\_send operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID task does not exist

OBJECT\_DELETED task specified has been deleted

XSR\_NOT\_SET task has no exception handler routine

NODE\_NOT\_REACHABLE node on which semaphore resides is not reachable

Description

This operation raises one or more exceptions to a task. If the task in question has an XSR, then unless it has the NOXHR modal parameter set, the XSR will be activated immediately and run not later than the task would normally be scheduled. If NOXHR is set, the XSR will be activated as soon as the task clears this parameter.

If the task has no current XSR, then this operation returns the XSR\_NOT\_SET completion status.

9.3. EXCEPTION\_RETURN

Return from Asynchronous Exception Handling Routine.

Synopsis

exception\_return( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

<not applicable>

Description

This operation transfers control from an XSR back to the code which it interrupted. It has no parameters and does not produce a completion status. This operation must be used to deactivate an XSR.

The behavior of exception\_return when not called from an XSR is undefined.

10. CLOCK

Each ORKID kernel maintains a node clock. This is a single data object in the kernel data structure which contains the current date and time. The clock is updated at every tick, the frequency of which is node dependent. The range of dates the clock is allowed to take is implementation dependent.

In a multi-node system, the different node clocks will very likely be synchronized, although this is not necessarily done automatically by the kernel. Since nodes could be in different time zones in widely distributed systems, the node clock specifies the local time zone, so that all nodes can synchronize their clocks to the same absolute time.

The data structure containing the clock value passed in clock operations is language binding dependent. It identifies the date and time down to the nearest tick, along with the local time zone. The time zone value is defined as the number of hours ahead (positive value) or behind (negative value) Greenwich Mean Time (GMT).

When the system starts up, the clock may be uninitialised. If this is the case, attempts at reading it before it has been set result in an error completion status, rather than returning a random value.

10.1. CLOCK\_SET

Set node time and date.

Synopsis

clock\_set( clock )

Input Parameters

clock : clock\_buf current time and date

Output Parameters

<none>

Completion Status

OK clock\_set operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_CLOCK invalid clock value

Description

This operation sets the node clock to the specified value. The kernel checks the supplied date and time in clock\_buf to ensure that they are legal. This is purely a syntactic check - the operation will accept any legal value. The exact structure of the data supplied is language binding dependent.

10.2. CLOCK\_GET

Get node time and date.

Synopsis

clock\_get( clock )

Input Parameters

<none>

Output Parameters

clock : clock\_buf current time and date

Completion Status

OK clock\_get operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

CLOCK\_NOT\_SET clock has not been initialized

Description

This operation returns the current date and time in the node clock. If the node clock has not yet been set, then the CLOCK\_NOT\_SET completion status is returned. The exact structure of the clock\_buf data returned is language binding dependent.

10.3. CLOCK\_TICK

Announce a tick to the clock.

Synopsis

clock\_tick( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK clock\_tick operation successful

Description

This operation increments the current node time by one tick. There are no parameters and the operation always succeeds. Every node must contain a mechanism which keeps the node clock up to date by calling upon CLOCK\_TICK.

11. TIMERS

ORKID defines two types of timers. The first type is the sleep timer. This type allows a task to sleep either for a given period, or up until a given time, and then wake and continue. Obviously a task can set only one such timer in operation at a time, and once set, it cannot be cancelled. These timers have no identifier.

The second type of timer is the event timer. This type allows a task to send events to itself either after a given period or at a given time. A task can have more than one event timer running at a time. Each event timer is assigned an identifier by the kernel when the event is set. This identifier can be used to cancel the timer.

Timers are purely local objects. They affect only the calling task, either by putting it to sleep or sending it events. Timers exist only while they are running. When they expire or are cancelled, they are deleted from the kernel data structure.

11.1. TIMER\_WAKE\_AFTER

Wake after a specified time interval.

Synopsis

timer\_wake\_after( ticks )

Input Parameters

ticks : integer number of ticks to wait

Output Parameters

<none>

Completion Status

OK timer\_wake\_after operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

Description

This operation causes the calling task to be blocked for the given number of ticks. The task is woken after this interval has expired, and is returned a successful completion status. If the node clock is set using the clock\_set operation during this interval, the number of ticks left does not change.

11.2. TIMER\_WAKE\_WHEN

Wake at a specified wall time.

Synopsis

timer\_wake\_when( clock )

Input Parameters

clock : clock\_buf time and date to wake

Output Parameters

<none>

Completion Status

OK timer\_wake\_when operation successful

ILLEGAL\_USE operation not callable from XSR or ISR

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_CLOCK invalid clock value

Description

This operation causes the calling task to be blocked up until a given date and time. The task is woken at this time, and is returned a successful completion status. The kernel checks the supplied clock\_buf data for validity. The exact structure of that data is language binding dependent.

If the node clock is set while the timer is running, the wall time at which the task is woken remains valid. If the node time is set to after the timer wake time, then the timer is deemed expired and the task is woken immediately and returned a successful completion status.

11.3. TIMER\_EVENT\_AFTER

Send event after a specified time interval.

Synopsis

timer\_event\_after( ticks, event, tmid )

Input Parameters

ticks : integer number of ticks to wait

event : bit\_field event to send

Output Parameters

tmid : timer\_id kernel defined timer identifier

Completion Status

OK timer\_event\_after operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

TOO\_MANY\_TIMERS too many timers on the node

Description

This operation starts an event timer which will send the given events to the calling task after the specified number of ticks. The kernel returns an identifier which can be used to cancel the timer. If the node clock is set using the clock\_set operation during this interval, the number of ticks left does not change.

11.4. TIMER\_EVENT\_WHEN

Send event at the specified wall time and date.

Synopsis

timer\_event\_when( clock, event, tmid )

Input Parameters

clock : clock\_buf time and date to send event

event : bit\_field event(s) to send

Output Parameters

tmid : timer\_id kernel defined timer identifier

Completion Status

OK timer\_event\_when operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_CLOCK invalid clock value

TOO\_MANY\_TIMERS too many timers on node

Description

This operation starts an event timer which will send the given events to the calling task at the given date and time. The kernel returns an identifier which can be used to cancel the timer.

If the node clock is set while the timer is running, the wall time at which the task is woken remains valid. If the node time is set to after the timer wake time, then the timer is deemed expired and the events are sent to the calling task immediately.

11.5. TIMER\_CANCEL

Cancel a running event timer.

Synopsis

timer\_cancel( tmid )

Input Parameters

tmid : timer\_id kernel defined timer identifier

Output Parameters

<none>

Completion Status

OK timer\_cancel operation successful

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_ID timer does not exist

Description

This operation cancels an event timer previously started using the timer\_event\_after or timer\_event\_when operations. The user specifies the timer using the identifier returned by these operations. If the given timer has expired or has been cancelled, the INVALID\_ID completion status is returned.

12. INTERRUPTS

ORKID defines two operations which bracket interrupt handler code. It is up to each implementor to decide what functionality, to put in these operations.

Observation:

The kernel may use int\_enter and int\_exit with an Interrupt Service Routine code or task code is being executed. Typically int\_exit will be used to decide if a scheduling action must take place in pre­emptive kernels.

12.1. INT\_ENTER

Announce interrupt handler entry.

Synopsis

int\_enter( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK int\_enter operation successful

Description

This operation call announces the start of an interrupt handling routine to the kernel. Its functionality is implementation dependent. The operation takes no parameters and always returns a successful completion status. It is up to a user task to set up vectors to the handler which makes this call.

12.2. INT\_EXIT

Exit from an interrupt handler.

Synopsis

int\_exit( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

<not applicable>

Description

This operation announces the end of an interrupt handling routine to the kernel. Its exact functionality is implementation dependent, but will involve returning to interrupted code or scheduling another task. The operation takes no parameters and does not return to the calling code.

The behavior of int\_return when not called from an Interrupt Service Routine is undefined.

A. RETURN CODES

CLOCK\_NOT\_SET clock has not been initialized

ILLEGAL\_USE operation not callable from XSR or ISR invalid options value

INVALID\_OPTIONS invalid options value

INVALID\_ADDRESS a specific parameter refers to an illegal address

INVALID\_ARGUMENTS invalid number or type or size of arguments

I NVALID\_BLOCK no block allocated from partition at blk\_addr

INVALID\_BLOCK\_SIZE block\_size not supported

INVALID\_CLOCK invalid clock value

INVALID\_COUNT init count is negative

INVALID\_GRANULARITY granularity not supported

INVALID\_ID object does not exist

INVALID\_LEINGTH buffer length not supported

INVALID\_LOCATION note-pad number does not exist

INVALID\_MODE invalid mode or mask value

INVALID\_NODE node does not exist

INVALID\_OPTIONS invalid options value

INVALID\_PARAMETER a parameter refers to an illegal address

INVALID\_PRIORITY invalid priority value

INVALID\_SEGMENT no segment allocated from this region at seg\_addr

NAME\_NOT\_FOUND name does not exist on node

NODE\_NOT\_REACHABLE node on which object resides is not reachable

NO\_EVENTS event(s) not set and NOWAIT option given

NO\_MORE\_MEMORY not enough memory to satisfy request

OBJECT\_DELETED specified object has been deleted

OBJECT\_PROTECTED task has NOPREEMPT or NOTERMINATION parameter set

OK operation successful

PARTITION\_IN\_USE blocks from this partition are still allocated

PARTITION\_OVERLAP Area given overlaps an existing partition

QUEUE\_DELETED queue deleted while blocked in queue\_receive operation

QUEUE\_EMPTY queue empty with NOWAIT option

QUEUE\_FULL no more buffers available

REGION\_IN\_USE segments from this region are still allocated

REGION\_OVERLAP area given overlaps an existing region

SEMAPHORE\_DELETED semaphore deleted while blocked in sem\_p operation

SEIMAPHORE\_NOT\_AVAILABLE semaphore unavailable with NOWAIT option

SEM\_OVERFLOW the counter of semaphore overflows

TASK\_ALREADY\_STARTED task has been started already

TASK\_ALREADY\_SUSPENDED task already suspended

TASK\_NOT\_STARTED task has not yet been started

TASK\_NOT\_SUSPENDED task not suspended

TIME\_OUT operation timed out

TOO\_MANY\_PARTITIONS too many partitions on the node

TOO\_MANY\_QUEUES too many queues on node

TOO\_MANY\_REIGIONS too many regions on the node

TOO\_MANY\_SEMAPHORES too many semaphores on node

TOO\_MANY\_TASKS too many tasks on the node

TOO\_MANY\_TIMERS too many timers on node

XSR\_NOT\_SET task has no exception handler routine

B. MINIMUM REQUIREMENTS FOR OPERATIONS FROM AN ISR.

ORKID requires that at least the following operations are supported from an Interrupt Service Routine. Only operations on local objects need to be supported. If the object resides on a remote node and remote operations are not supported, then the INVALID\_ID completion status must be returned.

Task Operations

task\_suspend ( tid )

task\_resume ( tid )

task\_read\_notepad ( tid, loc\_number, loc\_value )

task\_write\_notepad ( tid, loc\_number, loc\_value )

Semaphore Operations

sem\_v ( sid )

Queue Operations

queue\_send ( qid, message, length )

queue\_urgent ( qid, message, length )

Event Operations

event\_send ( tid, event )

Exception Operations

exceptions\_raise ( tid, exceptions )

Clock Operations

clock\_tick ( )

clock\_get ( clock )

Interrupt Operations

int\_enter ( )

int\_exit ( )

C. MINIMUM REQUIREMENTS FOR OPERATIONS FROM AN XSR.

ORKID requires that at least the following operations are supported from an Exception Service Routine.

Task Operations

task\_delete ( tid )

task\_start ( tid, start\_addr, arguments )

task\_restart ( tid, arguments )

task\_suspend ( tid )

task\_resume ( tid )

task\_set\_priority ( tid, new\_prio, old\_prio )

task\_set\_mode ( mode, mask, old\_mode )

task\_read\_notepad ( tid, loc\_number, loc\_value )

task\_write\_notepad ( tid, loc\_number, loc\_value )

Region Operations

region\_delete ( rid, options )

region\_get\_seg ( rid, seg\_size, seg\_addr )

region\_ret\_seg ( rid, seg\_addr )

region\_info ( rid, size, max\_segment, granularity )

Partition Operations

partition\_delete ( pid, options )

partition\_get\_blk ( pid, blk\_addr )

partition\_ret\_blk ( pid, blk\_addr )

partition\_info ( pid, blocks, free\_blockS, block\_size )

Semaphore Operations

sem\_delete ( sid )

sem\_p ( sid, time\_out )

sem\_v ( Sid )

sem\_info ( sit, options, count, tasks\_waiting )

Queue Operations

queue\_delete ( qid )

queue\_send ( qid, message, length )

queue\_urgent ( qid, message, length )

queue\_broadcast ( qid, message, length, count )

queue\_receive ( qid, message, time\_out )

queue\_flush ( qid, count )

queue\_info ( qid, max\_buf, length, options, messages\_waiting, tasks\_waiting )

Event Operations

event\_send ( tid, event )

event\_receive ( events, options, time\_out, events\_caught )

Exception Operations

exceptions\_send ( tid, exceptions )

exceptions\_return ( )

Clock Operations

clock\_set ( clock )

clock\_get ( clock )

clock\_tick ( )

Timer Operations

timer\_wake\_after ( ticks )

timer\_wake\_when ( clocks )

timer\_event\_after ( ticks, event, tmid )

timer\_event\_when ( clock, event, tmid )

timer\_cancel ( tmid )

D. SUMMARY OF ORKID OPERATIONS

In the following summary, output parameters are underlined.

Task Operations

task\_create ( name, priority, stack\_size, mode, options, tid )

task\_delete ( tid )

task\_ident ( name, nid, tid )

task\_start ( tid, start\_addr, arguments )

task\_restart ( tid, arguments )

task\_suspend ( tid )

task\_resume ( tid )

task\_setpriority ( tid, new\_prio, old\_prio )

task\_set\_mode ( mode, mask, old\_mode )

task\_read\_notepad ( tid, loc\_number, loc\_value )

task\_write\_notepad ( tid, loc\_number, loc\_value )

Region Operations

region\_create ( name, addr, length, granularity, options, rid )

region\_delete ( rid, options )

region\_ident ( name, rid )

region\_get\_seg ( rid, seg\_size, seg\_addr )

region\_ret\_seg ( rid, seg\_addr )

region\_info ( rid, size, max\_segment, granularity )

Partition Operations

partition\_create ( name, addr, length, block\_size, options, pid )

partition\_delete ( pid, options )

partition\_ident ( name, nid, pid, block\_size )

partition\_get\_blk ( pid, blk\_addr )

partition\_ret\_blk ( pid, blk\_addr )

partition\_info ( pid, blocks, free\_blocks, block\_size )

Semaphore Operations

sem\_create ( name, count, options, sid )

sem\_delete ( sid )

sem\_ident ( name, nid, sid )

sem\_p ( sid, time\_out )

sem\_v ( sid )

sem\_info ( sit, options, count, tasks\_waiting )

Queue Operations

queue\_create ( name, priv\_buff, max\_buff, length, options, qid )

queue\_delete ( qid )

queue\_ident ( name, nid, qid )

queue\_send ( qid, message, length )

queue\_urgent ( qid, message, length )

queue\_broadcast ( qid, message, length, count )

queue\_receive ( qid, message, time\_out )

queue\_flush ( qid, count )

queue\_info ( qid, max\_buf, length, options, messages\_waiting, tasks\_waiting )

Event Operations

event\_send ( tid, event )

event\_receive ( events, options, time\_out, events\_caught )

Exception Operations

exceptions\_catch ( new\_XSR, mode, old\_XSR, old\_mode )

exceptions\_send ( tid, exceptions )

exceptions\_return ( )

Clock Operations

clock\_set ( clock )

clock\_get ( clock )

clock\_tick ( )

Timer Operations

timer\_wake\_after ( ticks )

timer\_wake\_when ( clock )

timer\_event\_after ( ticks, event, tmid )

timer\_event\_when (clock, event, tmid )

timer\_cancel ( tmid )

Interrupt Operations

int\_enter ( )

int\_exit ( )

#ifndef ORKID\_H

#define ORKID\_H 1

/\*

E. ORKID: C LANGUAGE BINDING

This file contains the C language binding standard for VITA's "Open Real­time Kernel Interface Definition", henceforth called ORKID. The file is in the format of a C language header file, and is intended to be a common starting point for system developers wishing to produce an ORKID compliant kernel.

The ORKID C language binding consists of four sections, containing type specifications, function declarations, completion status codes and special symbol codes. The character sequence ??? has been used throughout wherever the coding is implementation dependent.

Of the four sections in this standard, only the function declarations are completely defined. In the other sections, only the type names and constant symbols are defined by this standard - all types and values are implementation dependent. Nevertheless, where possible, example values have been given.

Both ANSI C and non-ANSI C have been used for this header file. Defining the symbol \_\_ANSI\_\_ will cause the ANSI versions to be used, otherwise the non-ANSI versions will be used. Full prototyping has been employed for the ANSI function declarations.

\*/

/\*

ORKID TYPE SPECIFICATIONS

This section of the ORKID C language binding contains typedef definitions for the types used in operation arguments in the main ORKID standard. The names are the same as those in the ORKID standard. Only the names, and in clock\_buf the order of the structure members, are defined by this standard. The actual types are implementation dependent.

\*/

typedef unsigned int prio ;

typedef unsigned int lnum ;

typedef unsigned int bit\_field ;

typedef struct { ??? } task\_id ;

typedef struct { ??? } node\_id ;

typedef struct { ??? } region\_id ;

typedef struct { ??? } part\_id ;

typedef struct { ??? } sema\_id ;

typedef struct { ??? } queue\_id ;

typedef struct { ??? } timer\_id ;

typedef struct {

??? cb\_year ;

??? cb\_month ;

??? cb\_day ;

??? cb\_hours ;

??? cb\_minutes ;

??? cb\_seconds ;

??? cb\_tick ;

??? cb\_time\_zone ; } clock\_buf ;

/\*

ORKID OPERATION DECLARATIONS

This section of the the ORKID C language binding is the largest and contains function declarations for all the operations defined in the main ORKID standard, and is subdivided according to the subsections in this standard.

Each subdivision contains a list of function declarations and a list of symbol definitions. The function names have been kept to six characters for the sake of linker compatibility. Of these six characters, the first two are always 'OK', and the third designates the ORKID object type on which the operation works. The symbol definitions link the full names of the operations given in the ORKID standard (in lower case) to the appropriate abbreviation.

The lists of function declarations are split in two. If the symbol \_\_ANSI\_\_ has been defined, then all the functions are declared to the ANSI C standard using full prototyping, with parameter names also included. This latter is not necessary, but not illegal. It shows the correspondence between arguments in this and the main ORKID standard, the names being identical. If the symbol has not been defined, then the functions are declared without prototyping.

The correspondence between the C types and arguments and those defined in the ORKID standard are mostly obvious. However, the following comments concerning task\_start/restart and exceptionäcatch are perhaps necessary.

A task start address is translated into a function with one argument -a pointer to anything. The task‘s startup arguments are given as a pointer to anything and a length. The actual arguments will be contained in a programmer defined data type, a copy of which will be passed to the new task. The following is an example of a declaration of a task‘s main program and a call to start that task (the necessary task creation call is not included):

typedef struct { int arg1, arg2, arg3 } argblock ; /\* can contain anything \*/

argblock \*argp ;

void taskmain ( argblock \*taskargs ) { . . . } ; /\* main task program \*/

status = oktsta ( tid, taskmain, \*argp, sizeof( argblock ) ) ;

/\* start the task \*/

An XHR address also becomes a function with one argument - this time a bitfield. The previous XHR address output parameter becomes a pointer to such a function. The following is an example of the declaration of an XHR and a call to exception\_catch to set it up:

void taskxhr ( bit\_field exceptions\_caught ) { . . . } /\* XHR declaration \*/

void (\*prevxhr)() ;

status = okxcat ( taskxhr, NOXHR, prevxhr ) ; /\* set up taskxhr as XHR \*/

\*/

/\* TASK OPERATONS \*/

#ifdef \_\_ANSI\_\_

extern int oktcre( char \*name, prio priority, int stacksize, bit\_field mode, bit\_field options, task\_id \*tid ) ;

extern int oktdel( tasi\_id \*tid ) ;

extern int oktidt( char \*name, node\_id node, task\_id \*tid ) ;

extern int oktsta( task\_id \*tid, void start(void \*), void \*arguments, int arg\_length ) ;

extern int oktrst( task\_id \*tid, void \*arguments, int arg\_length ) ;

extern int oktsus( task\_id \*tid ) ;

extern int oktrsm( task\_id \*tid ) ;

extern int oktspr( task\_id \*tid, prio new\_prio, prio \*prev\_prio ) ;

extern int oktsmd( bit\_field mode, bit\_field mask, bit\_field \*prev\_mode ) ;

extern int oktrdl( task\_id \*tid, lnum loc\_number, int \*loc\_value ) ;

extern int oktwrl( task\_id \*tid, lnum loc\_number, int loc\_value ) ;

#else

extern int oktcre( ) ;

extern int oktdel( ) ;

extern int oktidt( ) ;

extern int oktsta( ) ;

extern int oktrst( ) ;

extern int oktsus( ) ;

extern int oktrsm( ) ;

extern int oktspr( ) ;

extern int oktsmd( ) ;

extern int oktrdl( ) ;

extern int oktwrl( ) ;

#endif

#define task\_create oktcre

#define task\_delete oktdel

#define task\_ident oktidt

#define task\_start oktsta

#define task\_restart oktrst

#define task\_suspend oktsus

#define task\_resume oktrsm

#define task\_set\_priority oktspr

#define task\_set\_mode oktsmd

#define task\_read\_location oktrdl

#define task\_write\_location oktwrl

/\* Region Operations \*/

#ifdef \_\_ANSI\_\_

extern int okrcre( char \*name, void \*addr, int length, int granularity, bit\_field options, region\_id \*rid ) ;

extern int okrdel( region\_id \*rid, bit\_field options ) ;

extern int okridt( char \*name, region\_id \*rid ) ;

extern int okrgsg( region\_id \*rid, int seg\_size, void \*\*seg\_addr ) ;

extern int okrrsg( region\_id \*rid, void \*seg\_addr ) ;

#else

extern int okrcre( ) ;

extern int okrdel( ) ;

extern int okridt( ) ;

extern int okrgsg( ) ;

extern int okrrsg( ) ;

#endif

#define region\_create okrcre

#define region\_delete okrdel

#define region\_ident okridt

#define region\_get\_seg okrgsg

#define region\_reg\_seg okrrsg

/\* Partition Operations \*/

#ifdef \_\_ANSI\_\_

extern int okpcre( char \*name, void \*addr, int length, int block\_size, bit\_field options, part\_id \*pid ) ;

extern int okpdel( part\_id \*pid, bit\_field options ) ;

extern int okpidt( char \*name, node\_id \*nid, part\_id \*pid, int block\_size ) ;

extern int okpgbl( part\_id \*pid, void \*\*blk\_addr ) ;

extern int okprbl( part\_id \*pid, void \*blk\_addr ) ;

#else

extern int okpcre( ) ;

extern int okpde( ) ;

extern int okpidt( ) ;

extern int okpgbl( ) ;

extern int okprbl( ) ;

#endif

#define partition\_create okpcre

#define partition\_delete okpdel

#define partition\_ident okpidt

#define partition\_get\_blk okpgbl

#define partition\_ret\_blk okprbl

/\* Semaphore Operations /\*

#ifdef \_\_ANSI\_\_

extern int okscre( char \*name, int count, bit\_field options, sema\_id \*sid ) ;

extern int oksdel( sema\_id \*sid ) ;

extern int oksidt( char \*name, node\_id \*nid, sema\_id \*sid ) ;

extern int oksemp( sema\_id \*sid, int time\_out ) ;

extern int oksemv( sema\_id \*sid ) ;

#else

extern int okscre( ) ;

extern int oksdel( ) ;

extern int oksidt( ) ;

extern int oksemp( ) ;

extern int oksemv( ) ;

#endif

#define sem\_create okscre

#define sem\_delete oksdel

#define sem\_ident oksidt

#define sem\_p oksemp

#define sem\_v oksemv

/\* Queue Operations \*/

#ifdef \_\_ANSI\_\_

extern int okqcre( char \*name, int priv\_buff, int max\_buff, int length, bit\_field options, queue\_id \*qid ) ;

extern int okqdel( queue\_id \*qid ) ;

extern int okqidt( char \*name, node\_id nid, queue\_id \*qid ) ;

extern int okqsnd( queue\_id \*qid, void \*message, int length ) ;

extern int okqurg( queue\_id \*qid, void \*message, int length ) ;

extern int okqbro( queue\_id \*qid, void \*message, int length, int \*count ) ;

extern int okqrcv( queue\_id \*qid, void \*message, int time\_out ) ;

extern int okqflu( queue\_id \*qid, int \*count ) ;

#else

extern int okqcre( ) ;

extern int okqdel( ) ;

extern int okqidt( ) ;

extern int okqsnd( ) ;

extern int okqurg( ) ;

extern int okqbro( ) ;

extern int okqrcv( ) ;

extern int okqflu( ) ;

#endif

#define queue\_create okqcre

#define queue\_delete okqdel

#define queue\_ident okqidt

#define queue\_send okqsnd

#define queue\_urgent okqurg

#define queue\_broadcast okqbro

#define queue\_receive okqrcv

#define queue\_flush okqflu

/\* Event Operations \*/

#ifdef \_\_ANSI\_\_

extern int okesnd( task\_id \*tid, bit\_field event ) ;

extern int okercv( bit\_field events, bit\_field options, int timeout, bit\_field \*events\_caught ) ;

#else

extern int okesnd( ) ;

extern int okercv( ) ;

#endif

#define event\_send okesnd

#define event\_receive okercv

/\* Exception operations \*/

#ifdef \_\_ANSI\_\_

extern int okxcat( void new\_xhr(bit\_field), bit\_field mode, void (\*old\_xhr)(bit\_field), bit\_field \*old\_mode ) ;

extern int okxsnd( task\_id \*tid, bit\_field exceptions ) ;

extern void okxret( void ) ;

#else

extern int okxcat( ) ;

extern int okxsnd( ) ;

extern void okxret( ) ;

#endif

#define exceptions\_catch okxcat

#define exceptions\_send okxsnd

#define exceptions\_return okxret

/\* Clock Operations \*/

#ifdef \_\_ANSI\_\_

extern int okcset( clock\_buf \*clock ) ;

extern int okcget( clock\_buf \*clock ) ;

extern int okctik( void ) ;

#else

extern int okcset( ) ;

extern int okcget( ) ;

extern int okctik( ) ;

#endif

#define clock\_set okcset

#define clock\_get okcget

#define clock\_tick okctik

/\* Timer Operations \*/

#ifdef \_\_ANSI\_\_

extern int oktmwa( int ticks ) ;

extern int oktmww( clock\_buf clock ) ;

extern int oktmea( int ticks, bit\_field event, timer\_id \*tmid ) ;

extern int oktmew( clock\_buf clock, bit\_field event, timer\_id \*tmid ) ;

extern int oktcan( timer\_id \*tmid ) ;

#else

extern int oktmwa( ) ;

extern int oktmww( ) ;

extern int oktmea( ) ;

extern int oktmew( ) ;

extern int oktcan( ) ;

#endif

#define timer\_wake\_after oktmwa

#define timer\_wake\_when oktmww

#define timer\_event\_after oktmea

#define timer\_event\_when oktmew

#define timer\_cancel oktmca

/\* Interrupt Operations \*/

#ifdef \_\_ANSI\_\_

extern int okient( void ) ;

extern void okiexi( void ) ;

#else

extern int okient( ) ;

extern void okiexi( ) ;

#endif

#define int\_enter okient

#define int\_exit okiexi

/\*

COMPLETION STATUS CONSTANTS

This section of the ORKID C language binding contains definitions for

all the completion status values used in the main ORKID standard. The

symbols used are the same as those given in the main standard, and are defined for C by this standard. Of the values, only the value O for the completion status ‘OK’ is defined here - the other values are given only as examples.

\*/

#define OK 0

#define CLOCK\_NOT\_SET 1

#define COUNT\_T0O\_HIGH 2

#define ILLEGAL\_USE 3

#define INVALID\_ADDRESS 4

#define INVALID\_ARGUMENT 5

#define INVALID\_BLOCK 6

#define INVALID\_BLOCK\_SIZE 7

#define INVALID\_CLOCK 8

#define INVALID\_COUNT 9

#define INVALID\_GRANULARITY 10

#define INVALID\_ID 11

#define INVALID\_LENGTH 12

#define INVALID\_LOCATION 13

#define INVALID\_MAX\_BUFF 14

#define INVALID\_MODE 15

#define INVALID\_NAME 16

#define INVALID\_NODE 17

#define INVALID\_OPTIONS 18

#define INVALID\_PRIORITY 19

#define INVALID\_SEGMENT 20

#define NAME\_NOT\_FOUND 21

#define N0\_EVENTS 22

#define NO\_MORE\_MEMORY 23

#define NODE\_NOT\_REACHABLE 24

#define 0BJECT\_DELETED 25

#define OBJECT\_NOT\_GLOBAL 26

#define PARTITION\_IN\_USE 27

#define PARTITION\_OVERLAP 28

#define QUEUE\_DELETED 29

#define QUEUE\_EMPTY 30

#define QUEUE\_FULL 31

#define REGION\_IN\_USE 32

#define REGION\_OVERLAP 33

#define SEMAPHORE\_DELETED 34

#define SEMAPHORE\_NOT\_AVAILABLE 35

#define TASK\_ALREADY\_STARTED 36

#define TASK\_ALREADY\_SUSPENDED 37

#define TASK\_MARKED\_FOR\_DELETE 38

#define TASK\_MARKED\_FOR\_RESTART 39

#define TASK\_NOT\_SUSPENDED 40

#define TIME\_OUT 41

#define TOO\_MANY\_PARTITIONS 42

#define T00\_MANY\_QUEUES 43

#define TOO\_MANY\_REGIONS 44

#define TOO\_MANY\_SEMAPHORES 45

#define TOO\_MANY\_TASKS 46

#define TOO\_MANY\_TIMERS 47

#define XHR\_NOT\_SET 48

/\*

LITERAL VALUES

This section of the RKID C language binding contains definitions for all special symbols used in argument values in the main ORKID standard. The symbols used are the same as those given in the main standard, and are defined for C by this standard. None of the values given here are defined by this standard — they are included as examples only.

\*/

#define SELF 0 /\* tid \*/

#define LOCAL\_NODE 0 /\* nid \*/

#define OTHER\_NODES -1

#define CURRENT 0 /\* new\_prio \*/

#define HIGHP 63 /\* new\_prio, prev\_prio, priority \*/

#define NOXHR 0x1 /\* mode, mask, prev\_mode \*/

#define NOTERMINATION 0x2

#define NOPREEMPT 0x4

#define NOINTERRUPT 0x8

#define GLOBAL 0x000l /\* options \*/

#define FORCED\_DELETE 0x0002

#define FIFO 0x0004

#define ANY 0x0008

#define NOWAIT 0x00l0

#define FOREVER 0 /\* time\_out \*/

#define NULL\_XHR 0 /\* new\_xhr, prev\_xhr \*/

#endif