ORKID

OPEN REAL-TlME KERNEL INTERFACE DEFINITION

Drafted by

The ORKID Working Group

Software Subcommittee of VITA

Draft 2.1

August 1990

Copyright 1990 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 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.

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.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 meager, on the contrary: a rich set of objects and operations has been provided.

One issue still has to be addressed: 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 required 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.

Furthermore it is foreseen that a companion driver interface definition will be published.TABLE OF CONTENTS

1. INTRODUCTION 5

2. ORKID CONCEPTS 6

2.1. Environment 6

2.2. ORKID Objects 6

2.3. Naming and Identification 8

2.4. ORKID Operations 8

2.5. Mu1ti-processing 9

2.6. ORKID conformance 11

2.7. Layout of Operation Descriptions 12

3. NODES 14

3.1. NODE\_IDENT 15

3.2. NODE\_FAIL 16

3.3. NODE\_INFO 17

4. TASKS 18

4.1. TASK\_CREATE 21

4.2. TASK DELETE 22

4.3. TASK\_IDENT 23

4.4. TASK\_START 24

4.5. TASK\_RESTART 25

4.6. TASK\_SUSPEND 26

4.7. TASK\_RESUME 27

4.8. TASK\_SET\_PRIORITY 28

4.9. TASK\_SET\_MODE 29

4.10. TASK\_READ\_NOTE\_PAD 30

4.11. TASK\_WRITE\_NOTE\_PAD 31

4.12. TASK\_INFO 32

5. REGIONS 33

5.1. REGION\_CREATE 34

5.2. REGION\_DELETE 35

5.3. REGION\_IDENT 36

5.4. REGION\_GET\_SEG 37

5.5. REGION\_RET\_SEG 38

5.6. REGION\_INFO 39

6. POOLS 40

6.1. POOL\_CREATE 41

6.2. POOL\_DELETE 42

6.3. POOL\_IDENT 43

6.4. POOL\_GET\_BLK 44

6.5. POOL\_RET\_BLK 45

6.6. POOL\_INFO 46

7. SEMAPHORES 47

7.1. SEM\_CREATE 48

7.2. SEM\_DELETE 49

7.3. SEM\_IDENT 50

7.4. SEM\_CLAIM 51

7.5. SEM\_RELEASE 52

7.6. SEM\_INFO 53

8. QUEUES 54

8.1. QUEUE\_CREATE 56

8.2. QUEUE\_DELETE 57

8.3. QUEUE\_IDENT 58

8.4. QUEUE\_SEND 59

8.5. QUEUE\_JUMP 60

8.6. QUEUE\_BROADCAST 61

8.7. QUEUE\_RECEIVE 62

8.8. QUEUE\_FLUSH 64

8.9. QUEUE\_INFO 65

9. EVENTS 66

9.1. EVENT\_SEND 67

9.2. EVENT\_RECEIVE 68

10. EXCEPTIONS 69

10.1. EXCEPTION\_CATCH 71

10.2. EXCEPTION\_RAISE 73

10.3. EXCEPTION\_RETURN 74

11. CLOCK 75

11.1. CLOCK\_SET 76

11.2. CLOCK\_GET 77

11.3. CLOCK\_TICK 78

12. TIMERS 79

12.1. TIMER\_WAKE\_AFTER 80

12.2. TIMER\_WAKE\_WHEN 81

12.3. TIMER\_EVENT\_AFTER 82

12.4. TIMER\_EVENT\_WHEN 83

12.5. TIMER\_EVENT\_EVERY 84

12.6. TIMER\_CANCEL 85

13. INTERRUPTS 86

13.1. INT\_ENTER 87

13.2. INT\_RETURN 88

14. MISCELLANEOUS 89

14.1. INT\_TO\_EXT 90

14.2. EXT\_TO\_INT 91

A. COMPLETION STATUSES 92

B. MINIMUM REQUIREMENTS FOR OPERATIONS FROM AN ISR 89

C. SUMMARY OF ORKID OPERATIONS 94

D. C LANGUAGE BINDING 96

1. INTRODUCTION

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

The application areas that ORKID aims at range 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.

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 CONCEPTS, 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 14 describe in detail the various standard types of objects 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, is 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. Nevertheless it will be possible to write ORKID compatible programs, which rely implicitly so much on the specific behavior of one implementation, that full portability might be endangered.

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 and parameter names, types and sequence.

2.0 ORKIDCONCEPTS

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 (see also 2.5 Multi-Procesing).

2.2. ORKID Objects

The standard 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.
* pools : memory areas for dynamic allocation of fixed sized buffers.
* 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 asynchronous exception service routines.
* note-pad : task specific integer locations for simple, unsynchronized data exchange.
* clock : 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. Besides task's, 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 areas of memory from which tasks may be allocated segments of varying size for their own purposes. Typically a region is defined to contain memory of one physical nature such as shared RAM, battery backed-up SRAM etc. The maximum number of regions on a node is implementation dependent.

Pools are consecutive areas of memory organized as a collection of fixed sized buffers which may be allocated to tasks. Pools are simpler than regions and are intended for fast dynamic memory allocation/de- allocation operations. In contrast to the more complex concept of a region pools can be operated on across node boundaries. The maximum number of pools on a node or in a system 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 are used 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 markers that allow a task to buffer until an event, or some combination thereof occurs, therefore they form a simple synchronization mechanism. Each task has the same, fixed number of events which is equal to the number of bits in the basic word length of the corresponding processor.

Exceptions too are task 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 routines (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 which is equal to the number of bits in the basic word length of the corresponding processor.

Note-pad locations are task specific variables that can be read or written without any form of synchronization or protection. The size of a node-pad location is equal to the basic word size of the corresponding processor. Each task has the same, fixed number of note- pads. The actual number is implementation dependent, but the minimum number is fixed at sixteen. The clock 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 clock value. The second type of timer is the event timer. This timer will, upon expiration, send 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 clock 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.

* 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.

It is also possible to have a mixture of these two schemes where a non-shared memory system may contain one or more sets of nodes connected via shared memory. These sets of nodes are called shared memory subsystems.

The behavior of a networked ORKID implementation should be consistent with the behavior of a shared memory ORKID system. Specifically, all operations on objects in remote nodes must return their completion status only after the respective operation actually completed.

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.

Globality of objects

Most objects of a node can be created with the GLOBAL option. Only global objects are visible to and accessible from other nodes. Their identifiers can be found via ident operations executed on another node. All operations on these objects, with the exception of the deletions, can equally be executed across node boundaries. Delete operations on remote objects will return the OBJECT\_NOT\_LOCAL completion status.

Remote operations on non-global objects will return the INVALID\_ID completion status.

Observation:

The above suggests that identifiers in multiple-node kernels will encode the node\_id of the node on which the object was created.

2.6 ORKID Conformance

There are several places in this standard where the exact algorithms to be used are defined by the implementer of the compliant kernel. 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 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 fulfill a segment request, but another returns an error, since it has left the region more fragmented.

Subsets and Extensions

ORKID compliant kernels must implement all operations and objects as defined in this document; no subsets are permitted. Any ORKID compliant implementation may 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.

Observation:

Books for user written extensions to the kernel will ease adaptation of ORKID compliant kernels to specific needs.

Undefined and Optional Items

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

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

ORKID does not define the word length. On this depends the size of integer parameters and bit—fields. These 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 task note-pad locations. The minimum number is sixteen.

ORKID does not define the range of priority values. But it defines the literal I-IIGH\_PRIORI'I'Y to improve portability.

ORKID defines neither inter-kernel communication methods nor kernel

data structure implementations. 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 on nodes with the same integer size.

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.

ORXID does not define the size limits on granularity for regions and buffer size for pools.

ORKID does not define any restrictions on the execution of operations within Interrupt Service 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 completion statuses, options and other symbolic values. But these values must be defined in the language binding.

ORKID does not define the maximum message length supported by a given implementation.

ORKID does not define the encoding of port designations in multi-port memory.

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 pseudo—1anguage 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 their 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 or returned from the operation. If none of the defined conditions is set, the value of the parameter must be zero. The literal ZERO is defined in ORKID for initializing options and mode to this value.

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 an implementation.

3. NODES

Nodes are the building bricks of ORKID systems, referenced by a node identifier and containing a single set of ORKID data structures. Nodes will typically contain a single CPU, but multi-CPU nodes are equally possible.

Specifying how nodes are created and configured is outside the scope of this standard. However, certain basic operations for node handling will be needed in all ORKID implementations and are defined in the following sections.

3.1. NODE\_lDENT

Obtain the identifier of a node with a given name.

Synopsis

node\_ident( name, nid )

Input Parameters

name : string user defined node name

Output Parameters

nid : node\_id system defined node identifier

Literal Values

name = WHO\_AM\_I returns nid of calling task

Completion Status

OK node\_ident successful

ILLEGAL\_USE node\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

NAME\_NOT\_FOUND no node with this name

Description

This operation returns the node identifier for the node with the given name. No assumption is made on how this identifier is obtained. If there is more than one mode with the same name in the system, then the nid of the first one found is returned.4.1. TASK\_CREATE

Create a task.

Synopsis

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

Input parameters

 name : string user defined task name

 priority : integer 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 task cannot be interrupted

 = ZERO no mode parameter set

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

Completion Status

 OK task\_create successful

 ILLEGAL\_USE task\_create not callable from ISR

 INVALID\_PARAMETER a parameter refers to an invalid address

 INVALID\_PRIORITY invalid priority value

 INVALID\_MODE invalid mode value

 INVALID\_OPTIONS invalid options value

 TOO\_MANY\_OBJECTS too many tasks on the node or in the system

 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.

4.2. 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\_de1ete successful

ILLEGAL\_USE task\_delete not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

OBJECT\_NOT\_LOCAL task\_delete not allowed on non­local task

OBJECT\_PROTECTED task in NOTERMINATION mode

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 parameter NOTERMINATION set, then the task will not be deleted and the completion status OBJECT\_PROTECTED will be returned.

Observation:

The task\_delete operation deallocates the task's stack but otherwise 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, buffers, etc., allocated to the task to be deleted have been returned.

For situations where one task wants to delete another, the recommended procedure is to ask this task to delete itself, typically using exceptions, or task\_restart with a specific argument. In this way the task can release all its resources before deleting itself.

4.3. 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

= ALL\_NODES all nodes in the system

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

Completion Status

OK task\_ident successful

ILLEGAL\_USE task\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID node does not exist

NAME\_NOT\_FOUND task 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 or ALL\_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.

4.4. 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 successful

ILLEGAL\_USE task\_start not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

INVALID\_ARGUMENTS invalid number or type or size of arguments

TASK\_ALREADY\_STARTED task has been started already

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 specifications of start address and the number and type of arguments are language binding dependent.

4.5. 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 successful

ILLEGAL\_USE task\_restart not callable from ISR

INVALID PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

INVALID\_ARGUMENTS invalid number or type or size of arguments

TASK\_NOT\_STARTED task has not yet been started

OBJECT\_PROTECTED task in NOTERMINATION mode

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. The task restarts executing with the priority and mode specified at task\_create. All event and exception latches are clared and no XSRs are defined.

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.

4.6. 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 successful

 INVALID\_PARAMETER a parameter refers to an invalid address

 INVALID\_ID task does not exist

 OBJECT\_DELETED originally existing task has been deleted before operation

 OBJECT\_PROTECTED task in NOPREEMPT mode

 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 completion status OBJECT\_PROTECTED, unless the task suspends itself. In which case the operation will always be successful. 4.7. TASK\_RESUME

Resume a suspended task.

Synopsis

 task\_resume( tid )

Input Parameters

 tid : task\_íd kernel defined task identifier

Output Parameters

 <none>

Completion Status

 OK task\_resume successful

 INVALID\_PARAMETER a parameter refers to an invalid address

 INVALID\_ID task does not exist

 OBJECT\_\_DELETED originally existing task has been deleted before operation

 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.

4.8. 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 : integer task’s new priority

Output Parameters

 old\_prio : integer 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 successful

 ILLEGAL\_USE task\_set\_priority not callable from ISR

 INVALID\_PARAMETER a parameter refers to an invalid address

 INVALID\_ID task does not exist

 OBJECT\_DELETED originally existing task has been deleted before operation

 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 4. Task Priority).

4.9. 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 task cannot be interrupted

 = ZERO no mode parameter set

 old\_mode same as new\_mode

 mask + NOXSR change XSR mode bit

 + NOTERMINATION change NOTERMINATION mode bit

 + NOPREEMPT change NOPREEMPT mode bit

 + NOINTERRUPT change NOINTERRUPT mode bit

 = ALL change all mode bits

 = ZERO change no mode bits

Completion Status

 OK task\_set\_mode successful

 ILLEGAL\_USE task\_set\_mode not callable from ISR

 INVALID\_PARAMETER a parameter refers to an invalid 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.

4.10. 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 : integer note-pad location number

Output Parameters

 loc\_value : word note-pad location value

Literal Values

 tid = SELF the calling task reads its own note-pad

Completion Status

OK task\_read\_note\_pad successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

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 note-pad location of the task identified by tid (see also 4. Task Note­-Pads). ORKID compliant kernels have a minimum of 16 note-pad locations, indexed via loc\_number starting at one.

4.11. TASK\_WRITE\_NOTE\_PAD

Write one of a task's note­pad locations.

Synopsis

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

Input Parameters

tid : task\_id kernel defined task id

loc\_number : integer note-pad location number

loc\_value : word note-pad location value

Output Parameters

<none>

Literal Values

tid = SELF the calling task writes into its own note-pad.

Completion Status

OK task\_write\_note\_pad successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

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 note­-pad location of the task identified by tid (see also 4. Task Note­-Pads). ORKID compliant kernels have a minimum of 16 note-pad locations, indexed via loc\_number starting at one.

4.12. TASK\_INFO

Obtain information on a task.

Synopsis

task\_info( tid, priority, mode, options, event, exception, state )

Input Parameters

tid : task\_\_id kernel defined task id

Output Parameters

priority : integer task priority

mode : bit\_field task mode

options : bit\_field task options

event : bit\_field event(s) latched for task

exception : bìtlfield exception(s) latched for task

state : integer task's execution state

Literal Values

tid = SELF the calling task requests information on itself

state = RUNNING task is executing

 READY task is ready for execution

 BLOCKED task is blocked

 SUSPENDED task is suspended

Completion Status

OK task\_info successful

ILLEGAL\_USE task\_info not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID task does not exist

OBJECT\_DELETED originally existing task has been deleted before

operation

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

Description

This operation provides information on the specified task. It returns the task's priority, mode, options, event and exception latches and the execution state. The latched bits in the task's event and exception bit\_fields are returned without interfering with the state of these latches. The task execution state indicates the state from the scheduler's point of view. If the task is blocked and subsequently suspended the SUSPENDED state will be passed back. All return values except options reflect the dynamic state of a task and should be used with care as they are just snapshots of this state at the time of executing the operation.

The operation, when called from an Exception Service Routine (XSR), returns this XSR's mode.

5. REGIONS

A region is an area of memory within a node which is organized by the kernel into a collection 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 pools, 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. This address has to be explicitly translated by the sender via int\_to\_ext and by the receiver via ext\_to\_int.

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 battery backed-­up SRAM, 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.

5.1. 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

Literal Values

options + FORCED\_DELETE deletion will go ahead even if there are

unreleased segments

Completion Status

OK region\_create successful

ILLEGAL\_USE region\_create not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_\_GRANULARITY granularity not supported

INVALID\_OPTIONS invalid options value

TOO\_MANY\_OBJECTS 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.

The FORCED\_DELETE option governs the deletion possibility of the region. (see 5.2. region\_delete)

5.2. REGION\_DELETE

Delete a region.

Synopsis

region\_delete( rid )

Input Parameters

rid : region\_id kernel defined region identifier

Output Parameters

<none>

Literal Values

options + FORCED\_DELETE deletion will go ahead even if there are

unreleased segments

Completion Status

OK region\_delete successful

ILLEGAL\_USE region\_delete not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID region does not exist

OBJECT\_DELETED originally existing region has been deleted before

operation

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

Description

Unless the FORCED\_DELETE option was specified at creation, 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.

5.3. REGION\_lDENT

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 successful

ILLEGAL\_USE region\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

NAME\_NOT\_FOUND region 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 the first one found.

5.4. REGION\_GET\_SEG

Get a segment from a region.

Synopsis

region\_get\_seg( rid, seg\_síze, 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 successful

ILLEGAL\_USE region\_get\_seg not callable from ISR

INVALID\_\_PARAMETER a parameter refers to an invalid address

INVALID\_ID region does not exist

OBJECT\_DELETED originally existing region has been deleted before

operation

NO\_MORE\_MEMORY not enough contiguous memory in the region to

allocate segment of requested size

Description

The region\_get\_seg operation requests a given sized segment from a given region's free memory. If the kernel cannot fulfill the request immediately, it returns the completion status NO\_MORE\_MEMORY, otherwise the address of the allocated segment is passed back in seg\_addr. 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.

5.5. REGION\_RET\_SEG

Return a segment to its region.

Synopsis

regíon\_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 successful

ILLEGAL\_USE region\_ret\_seg not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID region does not exist

OBJECT\_DELETED originally existing region has been deleted before

operation

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. The kernel checks that this segment was previously allocated from this region, and returns INVALID\_SEGMENT if it wasn't.

5.6. REGION\_INFO

Obtain information on a region.

Synopsis

regìon\_ínfo( rid, size, max\_segment, granularity, options )

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

options : bit\_field region create options

Completion Status

OK region\_info successful

ILLEGAL\_USE region\_info not callable from

ISR INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID region does not exist

OBJECT\_\_DELETED originally existing region has been deleted before operation

Description

This operation provides information on the specified region. It returns the size in bytes 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 in bytes 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 allocation granularity and options.

6. POOLS

A pool is an area of memory within a shared memory subsystem which is organized by the kernel into a collection of fixed size buffers. The area of memory to become a pool is declared to the kernel by a task when the pool is created, and is thereafter managed by the kernel until it is explicitly deleted by a task. The task also specifies the size of the buffers to be allocated from the pool. Any restrictions imposed on the buffer size are implementation dependent.

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

Once the pool has been created, tasks may request buffers one at a time from it, and can return them in any order. Because the buffers are all the same size, there is no fragmentation problem in pools. The exact allocation algorithms are implementation dependent. Addresses of buffers obtained vía pool\_get\_buff are translated to the callers address map for immediate use.

Observation:

Buffer addresses passed from one node to another in e.g. a message have to be explicitly translated by the sender via int\_to\_ext and by the receiver via ext\_to\_int.

6.1. POOL\_CREATE

Create a pool.

Synopsis

pool\_create( name, addr, length, buff\_size, options, pid )

Input Parameters

name : string user defined pool name

addr : address start address of pool

length : integer length of pool in bytes

buff\_size : integer pool buffer size in bytes

options : bit\_field pool create options

Output Parameters

pid : pool\_id kernel defined pool identifier

Literal Values

options + GLOBAL pool is global within the shared memory subsystem

+ FORCED\_DELETE deletion will go ahead even if there are unrealeased buffers

Completion Status

OK pool\_create successful

ILLEGAL\_USE pool\_create not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_BUFF\_SIZE buff\_size not supported

INVALID\_OPTIONS invalid options value

TOO\_MANY\_OBJECTS too many pools on the node or in the system

POOL\_OVERLAP area given overlaps an existing pool

Description

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

The FORCED\_DELETE option governs the deletion possibility of the pool (see 6.2 pool\_delete)

6.2. POOL\_DELETE

Delete a pool.

Synopsis

pool\_delete( pid )

Input Parameters

pid : pool\_id kernel defined pool identifier

Output Parameters

 <none>

Completion Status

OK pool\_delete successful

ILLEGAL\_USE pool\_delete not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID pool does not exist

OBJECT\_DELETED originally existing pool has been deleted before operation

POOL\_IN\_USE buffers from this pool are still allocated

OBJECT\_NOT\_LOCAL pool\_delete not allowed on non—local pools

Description

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

6.3. POOL\_IDENT

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

Synopsis

pool\_ident( name, nid, pid)

Input Parameters

name : string user defined pool name

nid : node\_id node identifier

Output Parameters

pid : pool\_id kernel defined pool identifier

Literal Values

nid = LOCAL\_NODE the node containing the calling task

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

= ALL\_NODES all nodes in the system

Completion Status

OK pool\_ident successful

ILLEGAL\_USE pool\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID node does not exist

NAME\_NOT\_FOUND pool does not exist on node

NODE\_NOT\_REACHABLE node is not reachable

Description

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

Observation:

This operation may return the pid of a GLOBAL pool that is not in the same shared memory subsystem as the node containing the calling task.

6.4. POOL\_GET\_BUFF

Get a buffer from a pool.

Synopsis

pool\_get\_buff( pid, buff\_addr )

Input Parameters

pid : pool\_id kernel defined pool identifier

Output Parameters

buff\_addr : address address of obtained buffer

Completion Status

OK pool\_get\_buff successful

ILLEGAL\_USE pool\_get\_buff not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID pool does not exist

OBJECT\_DELETED originally existing task has been deleted before operation

NO\_MORE\_MEMORY no more buffers available in pool

POOL\_NOT\_SHARED pool not in shared memory subsystem

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

Description

The pool\_get\_buff requests for a single buffer from the pool's free memory. If the kernel cannot immediately fulfil the request, it returns the completion status NO\_MORE\_MEMORY, otherwise the address of the allocated buffer is returned. The exact allocation algorithm is implementation dependent.

6.5. POOL\_RET\_BUFF

Return a buffer to its pool.

Synopsis

pool\_ret\_buff( pid, buff\_addr)

Input Parameters

pid : pool\_id kernel defined pool identifier

buff\_addr : address address of buffer to be returned

Output Parameters

<none>

Completion Status

OK pool\_ret\_buff successful

ILLEGAL\_USE pool\_ret\_buff not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID pool does not exist

OBJECT\_DELETED originally existing pool has been deleted before operation

POOL\_NOT\_SHARED pool not in shared memory sybsystem

INVALID\_BUFF no buffer allocated from pool at buff\_addr

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

Description

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

6.6 POOL\_INFO

Obtain information on a pool.

Synopsis

pool\_info( pid, buffers, free\_buffers, buff\_size, options )

Input Parameters

pid : pool-id kernel defined pool identifier

Output Parameters

buffers : integer number of buffers in the pool

free\_buffers : integer number of free buffers in the pool

buff\_size : integer pool buffer size in bytes

options : bit\_field pool create options

Completion Status

OK pool\_info successful

ILLEGAL\_USE pool\_info not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID pool does not exist

OBJECT\_DELETED originally existing pool has been deleted before operation

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

Description

This operation provides information on the specified pool. It returns its overall number of buffers, the number of free buffers in the pool, its buffer size in bytes and options. The number of free buffers in the pool should be used with care as it is just a snap—shot of the pools's usage at the time of executing the operation.

7. 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.

During a sem\_claim operation, the semaphore count is decremented by one. If the resulting semaphore count is greater than or equal to zero, then the calling task continues to execute. If the count is less than zero, the task blocks from processor usage and is put on a waiting queue for the semaphore. During a sem\_release 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 queue for this semaphore is unblocked and is made eligible for processor 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\_claim

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\_release operation.

In situations where multiple copies of a resource exist, \_the semaphore may be created with an initial count equal to a number of copies. A resource is claimed with the sem\_claim operation. When all available copies of the resource have been claimed, a task requiring the resource will be blocked until return of one of the claimed copies is announced by a sem\_release 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\_claim operation when it reaches a synchronization point. The other task performs a sem\_release operation when it reaches its synchronization point, unblocking the pending 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 with the FIFO option set enter additional tasks at the end of their waiting queue. Without this option, the tasks are enqueued in order of task priority. ORKID does not require reordering of semaphore waiting queues when a waiting task has his priority changed.

7.1. 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 : sem\_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 successful

ILLEGAL\_USE sem\_create not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_COUNT initial count is negative

INVALID\_OPTIONS invalid options value

TOO\_MANY\_OBJECTS too many semaphores on the node or in the system

Description

This operation creates a new semaphore in the kernel data structure, and returns its identifier. The semaphore is created with its count at the value given by the init\_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.

7.2. SEM\_DELETE

Delete a semaphore.

Synopsis

sem\_delete( sid )

Input Parameters

sid : sem\_id kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK sem\_delete successful

ILLEGAL\_USE sem delete not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED originally existing semaphore has been deleted before operation

0BJECT\_NOT\_LOCAL sem\_delete not allowed on non-local semaphore

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.

7.3. 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 : sem\_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

= ALL\_NODES all nodes in the system

Completion Status

OK sem\_ident successful

ILLEGAL\_USE sem\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID node does not exist

NAME\_NOT\_FOUND semaphore does not exist on node

NODE\_NOT\_REACHABLE node 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 or ALL\_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.

7.4. SEM\_CLAIM

Claim a semaphore (P operation).

Synopsis

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

Input Parameters

sid : sem\_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\_claim successful

ILLEGAL\_USE sem\_claim not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED originally existing semaphore has been deleted before operation

TIME\_OUT sem\_c1aim timed out

SEMAPHORE\_DELETED semaphore deleted while blocked in sem\_claim

SEMAPHORE\_NOT\_AVAILABLE semaphore unavailable with NOWAIT option

SEMAPHORE\_UNDERFLOW semaphore counter under flowed

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\_AVAILABLE 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 counter underflowed the SEMAPHORE\_UNDERFLOW completion status is returned. If the semaphore is deleted while a task is waiting on its queue, then the task is unblocked and the sem\_claim operation returns the SEMAPHORE\_DELETED completion status to the task. 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\_release operation on this semaphore, leading to the return of the successful completion status

7.5. SEM\_RELEASE

Release a semaphore (V operation) .

Synopsis

sem\_release( sid )

Input Parameters

sid :sem\_id kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

OK sem\_release successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID semaphore does not exist

OBJECT\_DELTED originally existing semaphore has been deleted before operation

SEMAPHORE\_OVERFLOW semaphore counter overflowed

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

Description

This operation increments the semaphore counter 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. If the counter overflowed the SEMAPHORE\_OVERFLOW completion status is returned.

7.6. 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 successful

ILLEGAL\_USE sem\_info not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID semaphore does not exist

OBJECT\_DELETED originally existing semaphore has been deleted before operation

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 snap-shot of the semaphore's state at the time of executing the operation.

8.3. 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

= ALL\_NODES all nodes in the system

Completion Status

OK queue\_ident successful

ILLEGAL\_USE queue\_ident not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID node does not exist

NAME\_NOT\_FOUND queue name does not exist on node

NODE\_NOT\_REACHABLE node 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 or ALL\_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.

8.4. QUEUE\_SEND

Send a message to a given queue.

Synopsis

 queue\_send ( qid, msg\_buff, msg\_length )

Input Parameters

qid : queue\_id kernel defined queue identifier

msg\_buff : address message starting address

msg\_length : integer length of message in bytes

Output Parameters

<none>

Completion Status

OK queue\_send successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID queue does not exist

OBJECT\_DELETED originally existing queue has been deleted before operation

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

QUEUE\_FULL no more buffers available

NODE\_NOT\_REACHABLE node on which queue 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 removed from the wait queue, unblocked and will be returned a successful completion status along with the message. Otherwise the message is appended at the end of the queue.

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

8.5. QUEUE\_JUMP

Send a message to the head of a given queue.

Synopsis

 queue\_jump( qid, msg\_buff, msg\_length )

Input Parameters

qid : queue\_id kernel defined queue identifier

msg\_buff : address message starting address

msg\_length : integer length of message in bytes

Output Parameters

<none>

Completion Status

OK queue\_jump successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID queue does not exist

OBJECT\_DELETED originally existing queue has been deleted before operation

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

QUEUE\_FULL no more buffers available

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

Description

This operations sends a message to the head of 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 removed from the wait queue, unblocked and will be returned a successful completion status along with the message. Otherwise the message is prepended at the head of the queue.

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

8.6. QUEUE\_BROADCAST

Broadcast message to all tasks blocked on a queue.

Synopsis

queue\_broadcast( qid, msg\_buff, msg\_length, count )

Input Parameters

qid : queue\_id kernel defined queue identifier

msg\_buff : address message starting address

msg\_length : integer length of message in bytes

Output Parameters

count : integer number of unblocked tasks

Completion Status

OK queue\_broadcast successful

ILLEGAL\_USE queue\_broadcast not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID queue does not exist

OBJECT\_DELETED originally existing queue has been deleted before operation

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

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

Description

This operation sends a message to all tasks waiting on a queue.

If the wait queue is empty, then no messages are sent, no tasks are unblocked and the count passed back 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 the wait queue, unblocked and returned a successful completion status. The number of tasks unblocked is passed back in the count parameter.

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

8.7. QUEUE\_RECEIVE

Receive a message from a queue.

Synopsis

queue\_receive ( qid, msg\_buff, buff\_length, options, time\_out, msg\_length )

Input Parameters

qid : queue\_id kernel defined queue identifier

msg\_buff : address

buff\_length : integer

options : bit\_field

time\_out : integer

Output Parameters

msg\_length : integer

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 queue\_receive successful

ILLEGAL\_USE queue\_receive not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID queue does not exist

OBJECT\_DELETED originally existing semaphore has been deleted before operation

INVALID\_LENGTH receive buffer smaller than queue’s message buffer

INVALID\_OPTIONS invalid options value

TIME\_OUT queue-receive timed out

QUEUE\_DELETED queue deleted while blocked in queue\_receive

QUEUE\_EMPTY queue unavailable with NOWAIT option

NODE NOT\_REACHABLE node on which queue resides is not reachable

Description

This operation receives a message from a given queue. The operation first checks if the receive buffer is smaller than the queue's message buffer. If this is the case the INVALID\_LENGTH completion status is returned.

Otherwise, if there are one or more messages on the queue, then the message at the head of the queue is removed and copied into the receive

buffer and a successful completion status returned.

If the message queue is empty, and NOWAIT was not specified in the options, then the task is blocked and put on the queue's wait queue. At that moment the time-out period is started. If the time-out expires then the TIME\_OUT completion status is returned.

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.

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.

8.8. 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 successful

ILLEGAL\_USE queue\_flush not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address queue does not exist

INVALID\_ID queue does not exist

OBJECT\_DELETED originally existing queue has been deleted before operation

NODE\_NOT\_REACHABLE node on which queue 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.

8.9. 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 allowed in
 queue
length : integer length of message buffers in bytes
options : bit\_field queue 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 successful
ILLEGAL\_USE queue\_info not callable from ISR
INVALID\_PARAMETER a parameter refers to an invalid
 address
INVALID\_ID queue does not exist
OBJECT\_DELETED originally existing queue has been deleted
 before operation
NODE\_NOT\_REACHABLE node on which queue resides is not reachable

Description

This operation provides information on the specified message queue. It returns its maximum number of buffers, their length 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 snap-shot of the queue's state at the time of executing the operation.

9. EVENTS

Events provide a simple method of task synchronization. Each task has the same number of events which is equal to the number of bits in the basic word length of the corresponding processor. Events have no identifiers, but are referenced using a task identifier and a bit-field. The 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 latches corresponding to the asked for combination. This means that if an event is sent more than once before being received, the second and subsequent sends are lost.

9.1. 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 successful
INVALID\_PARAMETER a parameter refers to an invalid address
INVALIDIID task does not exist
OBJECT\_DELETED originally existing task has been deleted
 before operation
NODE\_NOT\_REACHABLE node on which task 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.

9.2. EVENT\_RECElVE

Receive event(s).

Synopsis

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

Input Parameters

event : bit\_field event(s) to receive
options : bit\_field receive options
time\_out : integer ticks to wait before timing out

Output Parameters

event\_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 event(s)
 set
time\_out = FOREVER wait forever - do not time out

Completion Status

OK event\_receive successful
ILLEGAL\_USE event\_receive not callable from ISR
INVALID\_PARAMETER a parameter refers to an invalid address
INVALID\_OPTIONS invalid options value
TIME\_OUT event\_receive timed out
NO\_EVENT event(s) not set and NOWAIT option given

Description

This operation blocks a task until a given combination of events occurs. By default, the task waits until all of the events have been sent. If the ANY option is set, then the task waits only until at least 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 event\_received, and the corresponding event latches are cleared. If the ANY option was set, and one or more of the specified events was sent, all the events sent, satisfying the event parameter, 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 time—out expires, the TIME\_0UT completion status is returned.

10. EXCEPTIONS

ORKID exceptions provide tasks with a method of handling exceptional conditions asynchronously. Each task has the same number of exceptions which is equal to the number of bits in the basic word length of the corresponding processor. Exceptions have no identifiers, but are referenced using a task identifier and a bit-field. The bit-field can indicate any number of a task's exceptions at once.

Using this bit field, any number of exceptions can be raised simultaneously to a task. Each exception, defined by one bit of the bit-field, is associated with one specific Exception Service Routine (XSR) . If a task has no XSR defined for any one of the raised exceptions, then the corresponding exception bits are lost and the XSR\_NOT\_SET completion status is returned for the exception\_raise operation. Otherwise, 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 from the receiving of events, and involves the automatic activation by the scheduler of the task's XSRs corresponding to every set bit. XSRs have to be declared via the exception\_catch operation by tasks after their creation. A task may change its XSRs at any time.

An XSR is activated whenever the corresponding exception is raised to a task, and the task has not set its NOXSR mode parameter in the active mode. If the NOXSR parameter was 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, the corresponding latch is cleared and the XSR entered.

XSR code is executed in exactly the same way as other parts of the task. While it is executing, an XSR has no special privileges or restrictions compared to normal task code. The kernel automatically activates an XSR as detailed above, but the XSR will actually run only when the task would normally be scheduled to run. The XSR must normally deactivate and return to the code it interrupted with a special ORKID operation: exception\_return; alternatively it may alter the flow of execution through the task\_restart operation.

Observation:

Raising an exception to a task will not unblock a waiting task.

An 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.

If several exception bits are set at the same time, the Exception Service Routine corresponding to the highest bit-number set will be

activated. After executing the exception\_return operation in this XSR the routine corresponding to the bit with the second highest bit-number will be activated etc. An XSR running without the NOXSR bit in its mode will be interrupted by an exception of higher priority, i.e. with a higher bit-number. Exceptions of equal and lower priority will be latched.

The exception\_return operation will return either to the interrupted task, reinstating its original mode, or to the interrupted XSR with its original mode. This is also true in case of explicit change of an XSR's mode via task\_set\_mode.

10.1. EXCEPTION\_CATCH

Specify a task's Exception Service Routine for a given exception bit.

Synopsis

exception\_catch( bit\_number, new\_xsr, new\_mode, old\_xsr, old\_mode )

Input Parameters

bit\_number : integer exception bit-number
new\_xsr : address address of XSR
new\_mode : bit\_field execution mode to be ored in

Output Parameters

old\_xsr : address address of old XSR
old\_mode : bit\_field mode of old XSR

Literal Values

new\_xsr = NULL\_XSR task henceforth will have no XSR
 for the given exception bit
new\_mode + NOXSR XSRs cannot be activated
 + NOTERMINATION task cannot be restarted or deleted
 + NOPREEMPT task cannot be preempted
 + NOINTERRUPT task cannot be interrupted
 = ZERO no mode set

old\_\_mode same as new\_\_mode

o1d\_xsr = NULL\_XSR task previously had no XSR for the given
 exception bit

Completion Status

OK exception\_catch successful

ILLEGAL\_\_USE exception\_catch not callable from ISR

INVALID\_\_PARAMETER a parameter refers to an invalid address

INVALID\_\_MODE invalid mode value

INVALID\_BIT invalid exception bit-number

Description

This operation designates a new Exception Service Routine (XSR) for the exception given by bit\_number for the calling task. The task supplies the start address of the XSR, and the mode which will be ored to the active mode of the interrupted task or XSR to produce the active mode of this XSR. If this operation returns a successful completion status, the exception given by bit\_number will henceforth cause the XSR at the given address to be activated, if the running task does not have the NOXSR mode set.

The kernel returns the address of the previous XSR and the mode of that

XSR for the specified exception.

Note that if a task has no XSR defined for the given exception a 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 for this exception without designating a new one.

Observation:

This operation can be used for defining the corresponding XSR for the first time and when a task wishes to use a different XSR temporarily. Once finished with the temporary XSR, the original one can be simply reinstated using the old\_xsr and old\_mode values.

10.2. EXCEPTION\_RAISE

Raise exception(s) to a task.

Synopsis

exception\_raise( tid, exception )

Input Parameters

tid : task\_id kernel defined task id
exception : bit\_field exception(s) to be raised

Output Parameters

<none>

Completion Status

OK exception\_raise successful
INVALID\_PARAMETER a parameter refers to an invalid address
INVALID\_ID task does not exist
OBJECT\_DELETED originally existing task has been deleted
 before operation
XSR\_NOT\_SET no handler routine for given exception(s)
NODE\_NOT\_REACHABLE node on which task resides is not reachable

Description

This operation raises one or more exceptions to a task. If the task in question has XSR(s) defined for the given exception(s), then unless it has the NOXSR mode value set, the highest priority XSR will be activated immediately and will run when the task would be normally scheduled. If NOXSR is set, this XSR will be activated as soon as the task clears this parameter.

If the task has no XSR(s) for the given exception(s), then this operation returns the XSR\_NOT\_SET completion status.

10.3. EXCEPTION\_RETURN

Return from Exception Service 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.

11. 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.

11.1. CLOCK\_SET

Set node time and date.

Synopsis

clock\_set( clock )

Input Parameters

clock : clock\_buff current time and date

Output Parameters

<none>

Completion Status

OK clock\_set successful

ILLEGAL\_USE clock\_set not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid 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\_buff 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.

11.2. CLOCK\_GET

Get node time and date.

Synopsis

clock\_get( clock )

Input Parameters

<none>

Output Parameters

clock : clock\_buff current time and date

Completion Status

OK clock\_get successful

INVALID\_PARAMETER a parameter refers invalid 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 and the contents of clock are undetermined. The exact structure of the clock\_buff data returned is language binding dependent.

11.3. CLOCK\_TlCK

Announce a tick to the clock.

Synopsis

clock\_tick( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK clock\_tick successful

Description

This operation increments the current node time by one tick. There are no parameters and the operation always succeeds. Nevertheless, the operation can be meaningless if the clock was not initialized beforehand. Every node must contain a mechanism which keeps the node clock up to date by calling upon clock\_tick.

12. 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.

12.1. 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 successful

ILLEGAL\_USE timer\_wake\_after not callable from ISR

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.

12.2. TlMER\_WAKE\_WHEN

Wake at a specified wall time and date.

Synopsis

timer\_wake\_when( clock )

Input Parameters

clock : clock\_buff time and date to wake

Output Parameters

<none>

Completion Status

OK timer\_wake\_when successful

ILLEGAL\_USE timer\_wake\_when not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_CLOCK invalid clock value

CLOCK\_NOT\_SET clock has not been initialized

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.

12.3. 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 successful

ILLEGAL\_USE timer\_event\_after not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

TOO\_MANY\_OBJECTS 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.

12.4. TlMER\_EVENT\_WHEN

Send event at the specified wall time and date.

Synopsis

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

Input Parameters

clock : clock buff 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 successful

ILLEGAL\_USE timer\_event\_when not callable from ISR

INVALID\_PARAMETER A parameter refers to an invalid address

INVALID\_CLOCK invalid clock value

TOO MANY OBJECTS too many timers on the node

CLOEK\_NOT\_SET clock has not been initialized

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 event(s) are sent remains valid. If the node time is set to after the value specified in the clock parameter, then the timer is deemed expired and the events are sent to the calling task immediately.

12.5. TIMER\_EVENT\_EVERY

Send periodic event.

Synopsis

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

Input Parameters

ticks :integer number of ticks to wait between events

event :bit\_field event to send

Output Parameters

tmid : timer\_id kernel defined timer identifier

Completion Status

OK timer\_event\_every successful

ILLEGAL\_USE timer event\_every not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

TOO\_MANY\_OBJECTS too many timers on the node

Description

This operation starts an event timer which will periodically send the given events to the calling task with the periodicity specified by the 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 the life time of the timer, the number of ticks left until the next event does not change.

Observation:

This provides a drift—free mechanism for sending an event at periodic intervals.

12.6. TlMER\_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 successful

ILLEGAL\_USE timer\_cancel not callable from ISR

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_ID timer does not exist

OBJECT\_DELETED originally existing timer expired or has been cancelled before ions

Description

This operation cancels an event timer previously started using the timer\_event\_after, timer\_event\_when or timer\_event\_every operations.

13. INTERRUPTS

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

Observation:

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

13.1. lNT\_ENTER

Announce Interrupt Service Routine entry.

Synopsis

int\_enter( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

OK int\_enter successful

Description

This operation announces the start of an Interrupt Service 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.

13.2. INT\_RETURN

Exit from an Interrupt Service Routine

Synopsis

int\_return ( )

Input Parameters

<none>

Output Parameters

<none>

Completion Status

<not applicable>

Description

This operation announces the return from an ISR 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 ISR is undefined.

14. MISCELLANEOUS

This chapter contains the descriptions of miscellaneous operations.

In the current revision of ORKID these are restricted to address translation operations. These operations translate addresses of multiported memory from local processor addresses to the corresponding addresses on other ports and vice-versa.

14.1. INT\_TO\_EXT

Translate processor address to external port address.

Synopsis

int\_to\_ext( int\_addr, port, ext\_addr )

Input Parameters

int\_addr : address processor address to be translated

port : integer port designation

Output Parameters

ext\_addr :address corresponding address for designated port

Completion Status

OK int\_to\_ext successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_PORT port does not exist

NO\_TRANSLATION int\_addr cannot be accessed through port

Description

This operation translates a processor address of a multi—port memory location to the address accessing the same location via the given port. The port parameter encodes the bus and address space to be used, e.g. VMEbus with a certain address modifier. If the given port does not exist the INVALID\_PORT completion status is returned. If the given location cannot be accessed via the port the NO\_TRANSLATION completion status is returned.

Observation:

It is assumed that the various bus standard authorities will define literals for the encoding of ports for their respective bus architectures.

14.2. EXT\_TO\_lNT

Translate external port address to processor address.

Synopsis

ext\_to\_int( ext\_addr, port, int\_addr )

Input Parameters

ext\_addr: address port address to be translated

port: integer port designation

Output Parameters

int\_addr: address corresponding processor address

Completion Status

OK ext\_to\_int successful

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_PORT port does not exist

NO\_TRANSLATION ext\_addr can not be accessed by processor

Description

This operation translates an external port address of a multi-port memory to the processor address accessing the same location. The port parameter encodes the bus and address space to be used, e.g. VMEbus with a certain address modifier. If the given port does not exist the INVALID\_PORT completion status is returned. If the given location can not be accessed by the processor the NO\_TRANSLATION completion status is returned (see also 14.1. Observation).

A. COMPLETION STATUSES

CLOCK\_NOT\_SET clock has not been initialized

ILLEGAL\_USE operation not callable from ISR

INVALID\_ARGUMENTS invalid number or type or size of arguments

INVALID\_BIT invalid exception bit-number

INVALID\_BUFF no buffer allocated from partition at buff\_adr

INVALID\_BUFF\_SIZE buff\_size not supported

INVALID\_CLOCK invalid clock value

INVALID\_COUNT initial count is negative

INVALID\_GRANULARITY granularity not supported

INVALID\_ID object does not exist

INVALID\_LENGTH buffer length not supported

INVALID\_LOCATION note-pad number does not exist

INVALID\_MODE invalid mode or mask value

INVALID\_OPTIONS invalid options value

INVALID\_PARAMETER a parameter refers to an invalid address

INVALID\_PRIORITY invalid priority value

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

NAME\_NOT\_FOUND object name does not not exist on node

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

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

NO\_MORE\_MEMORY not enough memory to satisfy request

OBJECT\_DELETED originally existing task has been deleted before operation

OBJECT\_NOT\_LOCAL operation not allowed on non-local object

OBJECT\_PROTECTED task in NOTERMINATION mode

OK operation successful

POOL\_IN\_USE buffers from this pool are still allocated

POOL\_NOT\_SHARED pool not in shared memory subsystem

POOL\_OVERLAP area given overlaps an existing pool

QUEUE\_DELETED queue deleted while blocked in queue\_receive

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\_claim

SEMAPHORE\_NOT\_AVAILABLE semaphore unavailable with NOWAIT option

SEMAPHORE\_OVERFLOW semaphore counter overflowed

SEMAPHORE\_UNDERFLOW semaphore counter underflowed

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\_OBJECTS too many objects of given type on the node or in the system

XSR\_NOT\_SET no handler routine for given exception(s)

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 OBJECT\_NOT\_LOCAL completion status must be returned.

Observation:

The SELF literal is meaningless for ORKID operations called from an ISR and will lead to the INVALID\_ID completion status.

NODE OPERATIONS

node\_fail ( nid, code, options )

Task Operations

task\_suspend ( tid )

task\_resume ( tid )

task\_read\_note-pad ( tid, loc\_number, loc\_value )

task\_write\_note-pad ( tid, loc\_number, loc\_value )

Semaphore Operations

sem\_release ( sid )

Queue Operations

queue\_send ( qid, msg\_buff, msg\_length )

queue\_jump ( qid, msg\_buff, msg\_length )

Event Operations

event\_send ( tid, event )

Exception Operations

exception\_raise ( tid, exception )

Clock Operations

clock\_get ( clock )

clock\_tick ( )

Interrupt Operations

int\_enter ( )

int\_return ( )

C. SUMMARY OF ORKID OPERATIONS

In the following, output parameters are printed in bold characters.

Node Operations

node\_ident ( name, nid )

node\_fail ( nid, code, options )

node\_info ( nid, ticks\_per\_second )

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 )

task\_suspend ( tid )

task\_resume ( tid )

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

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

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

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

task\_info ( tid, priority, mode, options, event, exception)

Region Operations

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

region\_de1ete ( sid )

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, options )

Pool Operations

pool\_create ( name, addr, length, buff\_size, options, pid )

pool\_delete ( sid )

pool\_ident ( name, nid, pid )

pool\_get\_buff ( pid, buff\_addr )

pool\_ret\_buff ( pid, buff\_addr )

pool\_info ( pid, buffers, free\_buffers, buff\_size, options )

Semaphore Operations

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

sem\_delete ( sid )

sem\_ident ( name, nid, sid )

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

sem\_release ( sid )

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

Queue Operations

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

queue\_delete ( qid )

queue\_ident ( name, nid, qid )

queue\_send ( qid, msg\_buff, msg\_length )

queue\_jump ( qid, msg\_buff, msg\_length )

queue\_broadcast ( qid, msg\_buff, msglength, count )

queue\_receive ( qid, msg\_buff, buff\_length, options, time\_out, msg length )

queue\_flush ( qid, count )

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

Event Operations

event\_send ( tid, event )

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

Exception Operations

exception\_catch ( bit\_number, new\_xsr, new\_mode, old\_xsr, old\_mode)

exception\_raise ( tid, exception )

exception\_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\_event\_every ( ticks )

timer\_cancel ( tmid )

Interrupt Operations

int\_enter ( )

int\_return ( )

Miscellaneous Operations

int\_to\_ext ( int\_addr, port, ext\_addr )

ext\_to\_int ( ext\_addr, port, int\_addr )

#ifndef ORKID\_H

#define ORKID\_H 1

/\*

D. ORKID: C LANGUAGE BINDING

This file contains the C language binding standard for VITA's "Open Real-time 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.

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 versTons 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\_buff the order of the structure members, are defined by this standard. The actual types are implementation dependent.

\*/

typedef unsigned int prio ;

typedef unsigned int word ;

typedef unsigned int bit\_field ;

typedef ??? task\_id ;

typedef ??? node\_id ;

typedef ??? region\_id ;

typedef ??? pool\_id ;

typedef ??? sema\_id ;

typedef ??? queue\_id ;

typedef ??? timer\_id ;

typedef ??? cb\_year ;

typedef ??? cb\_month ;

typedef ??? cb\_day ;

typedef ??? cb\_hours ;

typedef ??? cb\_minutes ;

typedef ??? cb\_seconds ;

typedef ??? cb\_ticks ;

typedef ??? cb\_time\_zone ;

typedef ??? clock\_buff ;

/\*

ORKID OPERATION DECLARATIONS

This section of the ORKID C language binding 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 \_\_ANSI\_\_ 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 exception\_catch 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 argl, arg2, arg3 } argblock ; /\*can contain

argblock \*argp ; anything\*/

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

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

/\*start the task\*/

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

void taskxsr( bit\_field exception\_caught ){...}; /\*XSR

declaration\*/

void (\*oldxsr) () ;

status = okxcat( taskxsr, NOXSR, oldxsr ) ; /\*set up taskxsr as XSR\*/

 with NOXSR mode parameter

\*/

/\* Task Operations \*/

#ifdef \_\_ANSI\_\_

extern int oktcre( char \*name, prio priority, int stacksize, bit\_field

 mode, bit\_field options, task\_id tid ) ;

extern int oktdel( task\_id tid ) ;

extern int oktidt( char \*name, node id nid, 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 \*old\_prio ) ;

extern int oktsmd( bit\_field new\_mode, bit\_field mask, bit\_field

 \*old\_mode ) ;

extern int oktrnp( task\_id tid, int loc\_number, word \*loc\_value ) ;

extern int oktwnp( task\_id tid, int loc\_number, word loc\_value ) ;

extern int oktinf( task\_id tid, prio \*priority, bit\_field \*mode,

 bit\_field \*options, bit\_field \*event, bit\_field

 \*exception, int state );

#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 oktrnp( ) ;

extern int oktwnp( ) ;

extern int oktinf( ) ;

#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\_note\_pad oktrnp

#define task\_write\_note\_pad oktwnp

#define task info oktinf

/\* 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 ) ;

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 ) ;

extern int okrinf( region\_id rid, int size, int max\_segment,

 int granularity, bit\_field options)

#else

extern int okrcre( ) ;

extern int okrdel( ) ;

extern int okridt( ) ;

extern int okrgsg( ) ;

extern int okrrsg( ) ;

extern int okrinf( ) ;

#endif

#define region\_create okrcre

#define region\_delete okrdel

#define region\_ident okridt

#define region\_get\_seg okrgsg

#define region\_ret\_set okrrsg

#define region\_info okrinf

/\* Pool Operations \*/

#ifdef \_\_ANSI\_\_

extern int okpcre( char \*name, void \*addr, int length, int block\_size,

 bit\_field options, pool\_id \*pid ) ;

extern int okpdel( pool\_id pid ) ;

extern int okpidt( char \*name, node\_id nid, pool\_id \*pid);

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

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

extern int okpinf( pool\_id pid, int buffers, int free\_buffers,

 int buff\_size, bit\_field options)

#else

extern int okpcre( ) ;

extern int okpdel( ) ;

extern int okpidt( ) ;

extern int okpgbl( ) ;

extern int okprbl( ) ;

extern int okpinf( ) ;

#endif

#define pool\_create okpcre

#define pool\_delete okpdel

#define pool\_ident okpidt

#define pool\_get\_blk okpgbl

#define pool\_ret\_blk okprbl

#define pool\_info okpinf

/\* Semaphore Operations \*/

#ifdef \_\_ANSI\_\_

extern int okscre( char \*name, int init\_count, bit\_field options, sem\_id

 \*sid ) ;

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

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

extern int okstak( sem\_id \*sid, bit\_field options, int time\_out ) ;

extern int okssig( sem\_id \*sid ) ;

extern int oksinf( sem\_id \*sid, bit\_field options, int count,

 int tasks\_waiting)

#else

extern int okscre( ) ;

extern int oksdel( ) ;

extern int oksidt( ) ;

extern int okstak( ) ;

extern int okssig( ) ;

extern int oksinf( ) ;

#endif

#define sem\_create okscre

#define sem\_delete oksdel

#define sem\_ident oksidt

#define sem\_take okstak

#define sem\_signal okssig

#define sem\_info oksinf

/\* Queue Operations \*/

#ifdef \_\_ANSI\_\_

extern int okqcre( char \*name, int max\_buff, int length,

 bit\_field options, queue\_id \*gid ) ;

extern int okqdel( queue\_id qid ) ;

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

extern int okqsnd( queue\_id qid, void \*msg\_buff, int msg\_length ) ;

extern int okqjmp( queue\_id qid, void \*msg\_buff, int msg\_length );

extern int okqbro( queue\_id qid, void \*msg\_buff, int msg\_length,

 int \*count ) ;

extern int okqrcv( queue\_id qid, void \*msg\_buff, int buff\_length,

 bit\_field options, int time\_out, int length );

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

extern int okqinf( queue\_id qid, int max\_buff, int length,

 bit\_field options, int messages\_waiting,

 int tasks\_waiting)

#else

extern int okqcre( ) ;

extern int okqdel( ) ;

extern int okqidt( ) ;

extern int okqsnd( ) ;

extern int okqbro( ) ;

extern int okqjmp( ) ;

extern int okqrcv( ) ;

extern int okqflu( ) ;

extern int okqinf( ) ;

#endif

#define queue\_create okqcre

#define queue\_delete okqdel

#define queue\_ident okqidt

#define queue\_send okqsnd

#define queue\_broadcast okqbro

#define queue\_jump okqjmp

#define queue\_receive okqrcv

#define queue\_flush okqflu

#define queue\_info okqinf

/\* Event Operations \*/

#ifdef \_\_ANSI\_\_

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

extern int okercv( bit\_field event, bit\_field options, int time\_out,

bit\_field \*event\_received ) ;

#else

extern int okesnd( ) ;

extern int okercv( ) ;

#endif

#define event\_send okesnd

#define event\_receive okercv

/\* Exception Operations \*/

#ifdef \_\_ANSI\_\_

extern int okxcat( int bit\_number, void new\_xsr(bit\_field), bit\_field

new\_mode, void (\*old\_xsr) (bit\_field), bit\_field

\*old\_mode );

extern int okxrse( task\_id tid, bit\_field exception ) ;

extern void okxret( void ) ;

#else

extern int okxcat( ) ;

extern int okxrse( ) ;

extern void okxret( ) ;

#endif

#define exception\_catch okxcat

#define exception\_raise okxrse

#define exception\_return okxret

/\* Clock Operations \*/

#ifdef \_\_ANSI\_\_

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

extern int okcget( clock\_buff \*clock )

extern int okctik( void ) ;

#else

extern int okcset( );

extern int okcget( );

extern int okctik( );

#endif

#define clock\_set okcset

#define clock\_set okcget

#define clock\_tick okctik

/\* Timer Operations \*/

#ifdef \_\_ANSI\_\_\_

extern int oktmwa( int ticks ) ;

extern int oktmww( clock\_buff \*clock ) ;

extern int oktmea( int ticks, bit\_fie1devent, timer\_\_id \*tmid ; extern int oktmew( clock buff \*clock, bit\_field event, timer\_id \*tmid ( );

extern int oktmee( int ticks, bit\_field event, timer\_\_id \*tid );

extern int oktmca( timer\_id \*tmid\_) ;

#else

extern int oktmwa( );

extern int oktmww( );

extern int oktmea( );

extern int oktmew( );

extern int oktmee( );

extern int oktmca( );

#endif

#define timer\_wake\_after oktmwa

#define timer\_wake\_when oktmww

#define timer event after oktmea

#define timer:event:when oktmew

#define timer\_event\_every oktmee

#define timer\_cancel oktmca

/\* Interrupt Operations \*/

#ifdef ANSI

extern int okient( void ) ;

extern void okiret( void ) ;

#else

extern int okient( )

; extern void okiret( ) ;

#endif

#define int\_enter okient

#define int\_return okiret

/\*

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. \*/

#define OK ???

#define CLOCK NOT SET ???

#define ILLEGAL\_USE ???

#define INVALID\_ARGUMENT ???

#define INVALID\_BIT ???

#define INVALID\_BUFF ???

#define INVALID\_BUFF\_SIZE ???

#define INVALID\_CLOCK ???

#define INVALID\_COUNT ???

#define INVALID-GRANULARITY ???

#define INVALID-ID ???

#define INVALID\_LENGTH ???

#define INVALID\_LOCATION ???

#define INVALID-NODE ???

#define INVALID\_OPTIONS ???

#define INVALID\_PARAMETER ???

#define INVALID\_PRIORITY ???

#define INVALID\_SEGEMENT ???

#define NAME\_NOT\_FOUND ???

#define NODE\_NOT\_REACHABLE ???

#define NO\_EVENT ???

#define NO\_MORE\_MEMORY ???

#define OBJECT\_DELETED ???

#define OBJECT\_NOT\_LOCAL ???

#define OBJECT\_PROTECTED ???

#define POOL\_IN\_USE ???

#define POOL\_NOT\_SHARED ???

#define POOL\_OVERLAP ???

#define QUEUE\_DELETED ???

#define QUEUE\_EMPTY ???

#define QUEUE\_FULL ???

#define REGION\_IN\_USE ???

#define REGION\_OVERLAP ???

#define SEMAPHORE\_DELETED ???

#define SEMAPHORE\_NOT\_AVAILABLE ???

#define SEMAPHORE\_OVERFLOW ???

#define SEMAPHORE\_UNDERFLOW ???

#define TASK-ALREADY\_STARTED ???

#define TASK\_ALREADY-SUSPENDED ???

#define TASK\_NOT\_STARTED ???

#define TASK\_NOT\_SUSPENDED ???

#define TIME\_OUT ???

#define TOO\_MANY\_OBJECTS ???

#define XSR\_NOT\_SET ???

/\*

Literal Values

This section of the Orkid C language binding contains definitions for all special symbols used as 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. \*/

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

#define OTHER\_NODES ???

#define ALL\_NODES ???

#define WHO\_AM\_I ??? /\* name \*/

#define SELF ??? /\* tid \*/

#define RUNNING ??? /\* state \*/

#define READY ???

#define BLOCKED ???

#define SUSPENDED ???

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

#define HIGHP ??? /\* new\_prio, old\_prio \*/

#define NOXSR ??? /\* new\_mode, mode, mask, old\_mode \*/

#define NOTERMINATION ???

#define NOPREEMPT ???

#define NOINTERRUPT ???

#define ALL ??? /\* mask \*/

#define GLOBAL ??? /\* options \*/

#define FORCED\_DELETE ???

#define FIFO ???

#define ANY ???

#define NOWAIT ???

#define URGENT ???

#define ZERO ??? /\* options, mask, modes \*/

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

#define NULL\_XSR ??? /\* new\_xsr, old\_xsr \*/

#endif