ORKID
OPEN REAL-TIME 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  Microwave
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.

Eindhoven January 1990
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 profile their compliant product. Borderline conditions are that the standard:

- be implementable efficiently on a wide range of microprocessors,
- imposes no unnecessary hardware or software architecture,
- be open to future developments.

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

One issue 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. Multi-processing ................................. 9
   2.6. ORKID conformence ............................... 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>QUEUES</td>
<td>54</td>
</tr>
<tr>
<td>8.1.</td>
<td>QUEUE_CREATE</td>
<td>56</td>
</tr>
<tr>
<td>8.2.</td>
<td>QUEUE_DELETE</td>
<td>57</td>
</tr>
<tr>
<td>8.3.</td>
<td>QUEUE_IDENT</td>
<td>58</td>
</tr>
<tr>
<td>8.4.</td>
<td>QUEUE_SEND</td>
<td>59</td>
</tr>
<tr>
<td>8.5.</td>
<td>QUEUE_JUMP</td>
<td>60</td>
</tr>
<tr>
<td>8.6.</td>
<td>QUEUE_BROADCAST</td>
<td>61</td>
</tr>
<tr>
<td>8.7.</td>
<td>QUEUE_RECEIVE</td>
<td>62</td>
</tr>
<tr>
<td>8.8.</td>
<td>QUEUE_FLUSH</td>
<td>64</td>
</tr>
<tr>
<td>8.9.</td>
<td>QUEUE_INFO</td>
<td>65</td>
</tr>
<tr>
<td>9.</td>
<td>EVENTS</td>
<td>66</td>
</tr>
<tr>
<td>9.1.</td>
<td>EVENT_SEND</td>
<td>67</td>
</tr>
<tr>
<td>9.2.</td>
<td>EVENT_RECEIVE</td>
<td>68</td>
</tr>
<tr>
<td>10.</td>
<td>EXCEPTIONS</td>
<td>69</td>
</tr>
<tr>
<td>10.1.</td>
<td>EXCEPTION_CATCH</td>
<td>71</td>
</tr>
<tr>
<td>10.2.</td>
<td>EXCEPTION_RAISE</td>
<td>73</td>
</tr>
<tr>
<td>10.3.</td>
<td>EXCEPTION_RETURN</td>
<td>74</td>
</tr>
<tr>
<td>11.</td>
<td>CLOCK</td>
<td>75</td>
</tr>
<tr>
<td>11.1.</td>
<td>CLOCK_SET</td>
<td>76</td>
</tr>
<tr>
<td>11.2.</td>
<td>CLOCK_GET</td>
<td>77</td>
</tr>
<tr>
<td>11.3.</td>
<td>CLOCK_TICK</td>
<td>78</td>
</tr>
<tr>
<td>12.</td>
<td>TIMERS</td>
<td>79</td>
</tr>
<tr>
<td>12.1.</td>
<td>TIMER_WAKE_AFTER</td>
<td>80</td>
</tr>
<tr>
<td>12.2.</td>
<td>TIMER_WAKE_WHEN</td>
<td>81</td>
</tr>
<tr>
<td>12.3.</td>
<td>TIMER_EVENT_AFTER</td>
<td>82</td>
</tr>
<tr>
<td>12.4.</td>
<td>TIMER_EVENT_WHEN</td>
<td>83</td>
</tr>
<tr>
<td>12.5.</td>
<td>TIMER_EVENT_EVERY</td>
<td>84</td>
</tr>
<tr>
<td>12.6.</td>
<td>TIMER_CANCEL</td>
<td>85</td>
</tr>
<tr>
<td>13.</td>
<td>INTERRUPTS</td>
<td>86</td>
</tr>
<tr>
<td>13.1.</td>
<td>INT_ENTER</td>
<td>87</td>
</tr>
<tr>
<td>13.2.</td>
<td>INT_RETURN</td>
<td>88</td>
</tr>
<tr>
<td>14.</td>
<td>MISCELLANEOUS</td>
<td>89</td>
</tr>
<tr>
<td>14.1.</td>
<td>INT_TO_EXT</td>
<td>90</td>
</tr>
<tr>
<td>14.2.</td>
<td>EXT_TO_INT</td>
<td>91</td>
</tr>
<tr>
<td>A.</td>
<td>COMPLETION_STATUSES</td>
<td>92</td>
</tr>
<tr>
<td>B.</td>
<td>MINIMUM REQUIREMENTS FOR OPERATIONS FROM AN ISR</td>
<td>89</td>
</tr>
<tr>
<td>C.</td>
<td>SUMMARY OF ORKID OPERATIONS</td>
<td>94</td>
</tr>
<tr>
<td>D.</td>
<td>C LANGUAGE BINDING</td>
<td>96</td>
</tr>
</tbody>
</table>
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 ORKID CONCEPTS

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

2.1. Environment

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

*Hooks 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 node 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 HIGH_PRIORITY 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.

ORKID 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-language 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 a recipe for 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_IDENT

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
    ILLEGAL_USE
    INVALID_PARAMETER
    NAME_NOT_FOUND
          node_ident successful
          node_ident not callable from ISR
          a parameter refers to an invalid address
          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. It
there is more than one node with the same name in the system, then the
nid of the first one found is returned.
3.2. NODE_FAIL

Indicates fatal node failure to the system.

Synopsis

    node_fail( nid, code, options )

Input Parameters

    nid : node_id    system defined node identifier
    code : integer   type of error detected
    options : bit_field  failure options

Output Parameters

    <none>

Literal Values

    options    + TOTAL    all nodes should be stopped

Completion Status

    OK        node_fail successful
    INVALID_PARAMETER  a parameter refers to an invalid address
    INVALID_ID     node does not exist
    OBJECT_NOT_LOCAL node_fail on remote node not allowed from ISR
    NODE_NOT_REACHABLE node is not reachable

Description

This operation indicates a fatal failure of the type given by code in the node identified by nid to the system. If the TOTAL option is set all nodes of the system should be stopped, otherwise only the node identified by nid is stopped. The operation does not return if, as a result of the operation, the local node is stopped.

Observation:

The value in code can be transferred to a certain memory location or even displayed by hardware in the failing node to ease post mortem analysis of the failure.
3.3 NODE_INFO

Obtain information on a node.

Synopsis

    node_info( nid, ticks_per_sec )

Input Parameters

    nid : node_id            system defined node identifier

Output Parameters

    ticks_per_sec: integer   number of ticks per second for node clock

Completion Status

    OK                   node_info successful
    ILLEGAL_USE   node_info not callable from ISR
    INVALID_PARAMETER  a parameter refers to an invalid address
    INVALID_ID        node does not exist
    NODE_NOT_REACHABLE node is not reachable

Description

This operation obtains the number of ticks per second for the clock on the node identified by nid.

Observation:

For efficiency all delay times are specified in ticks. The value of ticks_per_sec allows tasks to convert between seconds and ticks.
4. TASKS

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

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

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

Task Exception Service Routine

A task may designate Exception Service Routine (XSR) to handle exceptions which have been raised for that task. A task can have one XSR defined for every bit in the exception bit-field. XSRs can be redefined dynamically. The purpose of XSRs is to deal with exceptions which have been raised for the task. It is recommended that exceptions be reserved for errors and other abnormal conditions which arise.

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

Task Priority

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

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

Task Modes

A task's mode determines certain aspects of the behavior of the kernel in respect to the task. The mode is made up by the combination of a number of mode parameters, each of which determines a single aspect of kernel behavior.
Open Real-time Kernel Interface Definition
Draft 2.1

This standard defines four values for a mode parameter, and an ORKID compliant kernel may add others. A given mode is specified by a bit-field, similarly to events and exceptions. Each bit of a mode bit-field specifies a single mode value. The bit for each value is identified by a standard symbolic value - the mapping of these symbols to numeric values is implementation dependent. The four standard mode values are as follows:

+ **NOXSR**
  This value affects only tasks with defined XSRs. When it is set, the task's XSRs will not be activated when exceptions are raised. Instead, exceptions will be latched until this value is cleared, after which the XSRs will be scheduled normally. Exceptions sent to a task without defined corresponding XSRs are lost.

+ **NOTERMINATION**
  When this value is set, the task is protected from forced deletion or restart by other tasks. NOTERMINATION allows a task to complete a section of code without risk of deletion or restart, and yet still allows other tasks to be scheduled.

+ **NOREPREEMP**
  When this value is set, the task will retain control of its CPU either until it clears the value, or until it blocks itself by an ORKID operation call. In this latter case, when the task is eventually re-scheduled, the NOPREEMPT value will still be set in its mode. In this mode the task is also protected from being suspended by another task. This value does not preclude activation of XSRs or ISRs.

+ **NOINTERRUPT**
  Tasks with this value set will not be interrupted.

**Observation:**

The NOINTERRUPT mode value does not preclude the execution of Interrupt Service Routines (ISR) by another processor in a multiple-processor node and therefore should not be used to obtain mutual exclusion with ISR code.

**Observation:**

A typical extension for certain processor architectures will be a SUPERVISOR mode value.

The behavior of a task is determined by the task's active mode. When a task is not executing an Exception Service Routine the mode specified in the task_create operation or the last task_set_mode operation is the active mode. Upon the activation of a task's XSR a new active mode is constructed by oring the old active mode with the mode specified in the exception catch operation.

After returning to the interrupted task this one will continue in its old active mode (see also 10. Exceptions).
Observation:

An XSR should, in general, not reset any mode value via the `task_set_mode` operation that was set at the time of its activation. This would lower the task's protection in an unforeseeable way.

Task Note-Pads

Every task has a fixed number of note-pad locations. These are simply 'word' locations which are accessible at all times by their own task, by all other tasks on the same node, and if the task was created with the GLOBAL option set, by all tasks on all nodes. The size of a note-pad location is equal to the basic word length of the corresponding processor. The note-pad is very simple, having only two operations—one to read and one to write a location.
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_delete 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 non-local 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
    start_addr : *
    arguments  : *

kernel defined task identifier

    task_start address

    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
arguments : *

kernel defined identifier
arguments passed to task

Output Parameters

<none>

Literal Values

tid = SELF

the calling task restarts itself.

Completion Status

OK
ILLEGAL_USE
INVALID_PARAMETER
INVALID_ID
OBJECT_DELETED
INVALID_ARGUMENTS
TASK_NOT_STARTED
OBJECT_PROTECTED
NODE_NOT_REACHABLE

task_restart successful
task_restart not callable from ISR
a parameter refers to an invalid address
task does not exist
originally existing task has been deleted
before operation
invalid number or type or size of arguments
task has not yet been started
task in NOTERMINATION mode
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 cleared 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**

```c
    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_id kernel defined task identifier

Output Parameters

<none>

Completion Status

OK
INVALID_PARAMETER
INVALID_ID
OBJECT_DELETED
TASK_NOT_SUSPENDED
NODE_NOT_REACHABLE

(task_resume successful
a parameter refers to an invalid address
task does not exist
originally existing task has been deleted
before operation
task not suspended
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  
    new_prio : integer

    kernel defined task id
    task's new priority

Output Parameters

    old_prio : integer

    task's previous priority

Literal Values

tid = SELF
    new_prio = CURRENT

    the calling task sets its own priority.
    there will be no change in priority.

Completion Status

    OK
    ILLLEGAL_USE
    INVALID_PARAMETER
    INVALID_ID
    OBJECT_DELETED
    INVALID_PRIORITY
    NODE_NOT_REACHABLE

    task_set_priority successful
    task_set_priority not callable from ISR
    a parameter refers to an invalid address
    task does not exist
    originally existing task has been deleted
    before operation
    invalid priority value
    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 note-pad 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
    INVALID_PARAMETER
    INVALID_ID
    OBJECT_DELETED
    INVALID_LOCATION
    NODE_NOT_REACHABLE

    task_write_note_pad successful
    a parameter refers to an invalid address
    task does not exist
    originally existing task has been deleted
    before operation
    note-pad number does not exist
    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 : bit_field  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_IDENT

Obtain the identifier of a region with a given name.

Synopsis

    region_ident( name, rid )

Input Parameters

    name     : string    user defined region name

Output Parameters

    rid      : region_id kernel defined region identifier

Completion Status

    OK        region_ident 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_size, seg_addr )

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rid</td>
<td>region_id</td>
<td>kernel defined region id</td>
</tr>
<tr>
<td>seg_size</td>
<td>integer</td>
<td>requested segment size in bytes</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg_addr</td>
<td>address</td>
<td>address of obtained segment</td>
</tr>
</tbody>
</table>

Completion Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>region_get_seg successful</td>
</tr>
<tr>
<td>ILLEGAL_USE</td>
<td>region_get_seg not callable from ISR</td>
</tr>
<tr>
<td>INVALID_PARAMETER</td>
<td>a parameter refers to an invalid address</td>
</tr>
<tr>
<td>INVALID_ID</td>
<td>region does not exist</td>
</tr>
<tr>
<td>OBJECT_DELETED</td>
<td>originally existing region has been deleted before operation</td>
</tr>
<tr>
<td>NO_MORE_MEMORY</td>
<td>not enough contiguous memory in the region to allocate segment of requested size</td>
</tr>
</tbody>
</table>

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

region_ret_seg( rid, seg_addr )

Input Parameters

<table>
<thead>
<tr>
<th>rid</th>
<th>region_id</th>
<th>kernel defined region id</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg_addr</td>
<td>address</td>
<td>address of segment to be returned</td>
</tr>
</tbody>
</table>

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

region_info( 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

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>region_info successful</td>
</tr>
<tr>
<td>ILLEGAL_USE</td>
<td>region_info not callable from ISR</td>
</tr>
<tr>
<td>INVALID_PARAMETER</td>
<td>a parameter refers to an invalid address</td>
</tr>
<tr>
<td>INVALID_ID</td>
<td>region does not exist</td>
</tr>
<tr>
<td>OBJECT_DELETED</td>
<td>originally existing region has been deleted before operation</td>
</tr>
</tbody>
</table>

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 via 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
                    deletion will go ahead even if there are unreleased buffers
    + FORCED_DELETE

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
    ILLEGAL_USE
    INVALID_PARAMETER
    INVALID_ID
    OBJECT_DELETED
    NO_MORE_MEMORY
    POOL_NOT_SHARED
    NODE_NOT_REACHABLE

    pool_get_buff successful
    pool_get_buff not callable from ISR
    a parameter refers to an invalid address
    pool does not exist
    originally existing task has been deleted
    before operation
    no more buffers available in pool
    pool not in shared memory subsystem
    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 fulfill 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**

```c
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
    + FIFO       throughout the system
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

```c
sem_delete( sid )
```

Input Parameters

```c
typed tai
```

kernel defined semaphore identifier

Output Parameters

<none>

Completion Status

```c
OK
ILLEGAL_USE
INVALID_PARAMETER
INVALID_ID
OBJECT_DELETED
OBJECT_NOT_LOCAL
```

sem_delete successful
sem_delete not callable from ISR
a parameter refers to an invalid address
semaphore does not exist
originally existing semaphore has been deleted before operation
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
        = 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_claim timed out
SEMAPHORE_DELETED  semaphore deleted while blocked in
                   sem_claim
SEMAPHORE_NOT_AVAILABLE  semaphore unavailable with NOWAIT option
SEMAPHORE_UNDERFLOW  semaphore counter underflowed
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**

```c
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_DELETED**  
  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 its 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
    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 operation 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
- **msg_buff**: address
- **msg_length**: integer

kernel defined queue identifier
message starting address
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 operation 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  message length 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
from 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        starting address of receive buffer
    buff_length: integer        length of receive buffer in bytes
    options     : bit field      queue receive options
    time_out    : integer        ticks to wait before timing out

Output Parameters

    msg_length : integer        received message length in bytes

Literal Values

    options    + NOWAIT     do not wait - return immediately if no
                            message in queue
    time_out   = FOREVER    wait forever - do not time out

Completion Status

    OK           queue_receive 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 queue 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 empty 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
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
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 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_RECEIVE

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
                    do not wait - return immediately if no
                    event(s) set
    + NOWAIT
    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_OUT 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 or'd 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
         + NOXSR  for the given exception bit
         + NOTERMINATION  XSRs cannot be activated
         + NOPREEMPT  task cannot be restarted or deleted
         + NOINTERRUPT  task cannot be preempted
         = ZERO  no mode set

old_mode  same as new_mode

old_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
INVALID_PARAMETER
INVALID_ID
OBJECT_DELETED
XSR_NOT_SET
NODE_NOT_REACHABLE

exception_raise successful
a parameter refers to an invalid address
task does not exist
originally existing task has been deleted
before operation
no handler routine for given exception(s)
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 to an 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_TICK

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

    timerWakeAfter( ticks )

Input Parameters

    ticks : integer    number of ticks to wait

Output Parameters

    <none>

Completion Status

    OK          timerWakeAfter successful
    ILLEGAL_USE timerWakeAfter 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. TIMER_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. TIMER_EVENT_WHEN

Send event at the specified wall time and date.

Synopsis

timer_event_when( clock, event, tmid )

Input Parameters

clock : clock_buff
event : bit_field

Reference:  time and date to send event
event(s) to send

Output Parameters

tmid : timer_id

Reference:  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
CLOCK_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. TIMER_CANCEL

Cancel a running event timer.

Synopsis

timer_cancel( tmid )

Input Parameters

tmid : timer_id  kernel defined timer identifier

Output Parameters

<none>

Completion Status

OK  timer_cancel 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 canceled before operation

Description

This operation cancels an event timer previously started using the timer_event_after, timer_event_when or timer_event_every operations.
13. INTERRUPTS

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

Observation:

The kernel may use int_enter and int_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. **INT_ENTER**

Announce Interrupt Service Routine entry.

**Synopsis**

    int_enter( )

**Input Parameters**

    <none>

**Output Parameters**

    <none>

**Completion Status**

    OK  \hspace{1cm} \textit{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 multi-ported 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 can not 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_INT

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_addr
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 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_sec)

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_resume (tid, arguments)
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_delete (rid)
region_ident (name, rid)
region_get_seg (rid, seg_size, seg_addr)
region_ret_seg (tid, seg_addr)
region_info (tid, size, max_segment, granularity, options)

Pool Operations

pool_create (name, addr, length, buff_size, options, pid)
pool_delete (pid)
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, tasks_waiting)

Queue Operations

queue_create (name, max_buff, length, options, qid)
Open Real-time Kernel Interface Definition
Draft 2.1

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, msg_length, 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, event, tmid )
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 )
Open Real-time Kernel Interface Definition
Draft 2.1

 ifdef 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 versions will be used. Full prototyping has been
 employed for the ANSI function declarations.
 */
/*
ORKID TYPE SPECIFICATIONS

This section of the ORKID C language binding contains typedef definitions for the types used in operation arguments in the main ORKID standard. The names are the same as those in the ORKID standard. Only the names, and in clock_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 arg1, 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:

```c
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 */

#define __ANSI__

extern int oktcre( char *name, prio priority, int stacksize, bit_field
task_id tid );
extern int oktdel( task_id tid );
extern int oktgid( 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

define oktcre() ;
edefine oktdel() ;
edefine oktgid() ;
edefine oktsta() ;
edefine oktrst() ;
edefine oktsus() ;
edefine oktrsm() ;
edefine oktspr() ;
edefine oktsmd() ;
edefine oktrnp() ;
edefine oktwnp() ;
edefine oktinf() ;
#endif

#define task_create oktcre
#define task_delete oktdel
#define task_ident oktgid
#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 */

#define _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);

#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 okscr( 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 okscr( ) ;
extern int oksdel( ) ;
extern int oksidt( ) ;
extern int okstak( ) ;
extern int okssig( ) ;
extern int oksinf( ) ;

#endif

#define sem_create okscr  
#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 *qid );
extern int okqdel( queue_id qid );
extern int okqidt( char *name, node_id nid, queue_id *qid );
extern int okqsend( 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 okqsend( );
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 okqsend
#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 */

#ifndef __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 okxrase( task_id tid, bit_field exception );
extern void okxret( void );

#else

extern int okxcat( );
extern int okxrase( );
extern void okxret( );

#endif

#define exception_catch         okxcat
#define exception_raise         okxrase
#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_get 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_field event, 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_SEGMENT
#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 */

#undef