2.3. Naming and Identification

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

Observation:

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

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

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

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

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

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

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

2.4. ORKID Operations

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

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

The completion status may indicate success, a specific error condition such as an invalid parameter value, or a specific operational condition such as a time-out. When multiple conditions apply, only one status is returned, defined by an implementation dependent precedence. All statuses have symbolic values - the mapping of these symbols to numeric values is implementation dependent.

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

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

2.5. Multi-processing

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

Nodes

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

Systems

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

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

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

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

UNAPPROVED DRAFT. All rights reserved by VITA
Do not specify or claim conformance to this document.

System configuration

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

Levels of Compliance

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

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

2.6 ORKID compatibility

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

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

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

Extensions

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

Undefined Items

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

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

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

UNAPPROVED DRAFT. All rights reserved by VITA
Do not specify or claim conformance to this document.

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

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

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

ORKID does not define the range of priority values.

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

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

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

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

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

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

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

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

2.7. Layout of Operation Descriptions

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

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

UNAPPROVED DRAFT. All rights reserved by VITA
Do not specify or claim conformance to this document.

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 types are given definitively in the language bindings.

Output Parameters

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

Literal Values

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

```
<parameter name> = <literal value>   Commentary
<parameter name> + <literal value>   Commentary
```

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

Completion Status

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

Description

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

UNAPPROVED DRAFT. All rights reserved by VITA
Do not specify or claim conformance to this document.