FROM THE CHAIRMAN

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

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

The members of the working group are:

Reed Cardoza	Eyring Research	
Alfred Chao	Software Components	
Chris Eck	CERN	
Wayne Fischer	FORCE Computers	
John Fogelin	Wind River Systems	
Zoltan Hunor	VITA Europe	(secretary)
Kim Kempf	Microware	
Hugh Maaskant	Philips	(chairman)
Dick Vanderlin	Motorola	

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

Eindhoven July 1989

FOREWORD

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

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

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

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

TABLE OF CONTENTS

1. INTRODUCTION

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

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

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

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

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

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

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

2. ORKID CONCEPTS

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

2.1. Environment

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

2.2. ORKID Objects

The standard **ORKID** object types defined by **ORKID** are:

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

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

Regions are consecutive chunks of memory from which tasks may allocate segments of varying size for their own purposes. Typically a region consists of memory of one physical nature such as shared RAM, battery backed-up SRAM etc. The maximum number of regions on a node are implementation dependent.

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

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

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

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

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

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

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

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

2.3. Naming and Identification

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

Observation:

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

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

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

One or more of a given task's events or exceptions may be specified using a 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.

System configuration

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

Levels of Compliance

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

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

2.6 ORKID compatibility

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

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

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

Extensions

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

Undefined Items

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

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

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

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

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

ORKID does not define the range of priority values.

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

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

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

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

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

ORKID 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:

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.

3. TASKS

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

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

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

Task Exception Service Routine

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

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

Task Priority

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

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

Task Modes

A 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.

3.1. TASK_CREATE

Create a task.

Synopsis

```
   task_create( name, priority, stack_size, mode, options, tid )
```

Input parameters

name	: string	user defined task name
priority	: prio	initial task priority
stack_size	: integer	size in bytes of task's stack
mode	: bit_field	initial task mode
options	: bit_field	creation options

Output Parameters

tid	: task_id	kernel defined task identifier

Literal Values

mode	+ NOXSR	XSRs cannot be activated
	+ NOTERMINATION	task cannot be restarted or deleted
	+ NOPREEMPT	task cannot be preempted
	+ NOINTERRUPT	interrupt handling routine cannot be activated
options	+ GLOBAL	New task will be visible throughout the system.

Completion Status

OK	task_create operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_PRIORITY	invalid priority value
INVALID_MODE	invalid mode value
INVALID_OPTIONS	invalid options value
TOO_MANY_TASKS	too many tasks on the node
NO_MORE_MEMORY	not enough memory to allocate task data structure or task stack

Description

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

3.2. TASK_DELETE

Delete a task.

Synopsis

```
task_delete( tid )
```

Input Parameters

tid	: task_id	kernel defined task identifier

Output Parameters

<none>

Literal Values

tid	= SELF	The calling task requests its own deletion.

Completion Status

OK	task_delete operation successful
ILLEGAL_USE	operation not callable from ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_ID	task does not exist
OBJECT_DELETED	task specified has been deleted
OBJECT_PROTECTED	task has NO_TERMINATION parameter set
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

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

Observation:

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

For situations where one task must delete another, clean-up will usually require co-operation between the tasks, typically using exceptions, or task_restart.

3.3. TASK_IDENT

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

Synopsis

```
   task_ident( name, nid, tid )
```

Input Parameters

name	: string	user defined task name
nid	: node_id	node identifier

Output Parameters

tid	: task_id	kernel defined task identifier

Literal Values

nid	= LOCAL_NODE	The node containing the calling task
	= OTHER_NODES	all nodes in the system except the local node
name	= WHO_AM_I	Returns tid of calling task

Completion Status

OK	task_ident operation successful
ILLEGAL_USE	operation not callable from XSR or ISR
INVALID_PARAMETER	a parameter refers to an illegal address
INVALID_NODE	node does not exist
NAME_NOT_FOUND	name does not exist on node
NODE_NOT_REACHABLE	node on which task resides is not reachable

Description

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