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.