ORKID

OPEN REAL-TlME KERNEL INTERFACE DEFINITION

Drafted by

The ORKID Working Group

Software Subcommittee of VITA

Draft 2.1

August 1990

Copyright 1990 by VITA, the VMEbus International Trade Association

No parts of this document may be reproduced or used in any form or any means - electronic, graphic, mechanical, electrical or chemical, photocopying, recording in any medium, taping by any computer or storage system etc without prior permission in writing from VITA, the VMEbus International Trade Association.

Exception:

This document may be reproduced or multiplied by photocopying for the exclusive purpose of soliciting public comments on the draft.

FROM THE CHAIRMAN

Before you lies the draft of VITA's Open Real Time Interface Definition, known as ORKID. This draft is the result of the activities of a small working group under the auspices of the Software Subcommittee of the VITA Technical Committee.

The members of the working group are:

Reed Cardoza Eyring Research

Alfred Chao Software Components

Chris Eck CERN

Wayne Fischer FORCE Computers

John Fogelin Wind River Systems

Zoltan Hunor VITA Europe (secretary)

Kim Kempf Microware

Hugh Maaskant Philips (chairman)

Dick Vanderlin Motorola

I would like to thank these members for their efforts. Also I would like to thank the companies they represent for providing the time and expenses of these members. Without that support this draft would not have been possible.FOREWORD

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

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

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

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

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

1. INTRODUCTION 5

2. ORKID CONCEPTS 6

2.1. Environment 6

2.2. ORKID Objects 6

2.3. Naming and Identification 8

2.4. ORKID Operations 8

2.5. Mu1ti-processing 9

2.6. ORKID conformance 11

2.7. Layout of Operation Descriptions 12

3. NODES 14

3.1. NODE\_IDENT 15

3.2. NODE\_FAIL 16

3.3. NODE\_INFO 17

4. TASKS 18

4.1. TASK\_CREATE 21

4.2. TASK DELETE 22

4.3. TASK\_IDENT 23

4.4. TASK\_START 24

4.5. TASK\_RESTART 25

4.6. TASK\_SUSPEND 26

4.7. TASK\_RESUME 27

4.8. TASK\_SET\_PRIORITY 28

4.9. TASK\_SET\_MODE 29

4.10. TASK\_READ\_NOTE\_PAD 30

4.11. TASK\_WRITE\_NOTE\_PAD 31

4.12. TASK\_INFO 32

5. REGIONS 33

5.1. REGION\_CREATE 34

5.2. REGION\_DELETE 35

5.3. REGION\_IDENT 36

5.4. REGION\_GET\_SEG 37

5.5. REGION\_RET\_SEG 38

5.6. REGION\_INFO 39

6. POOLS 40

6.1. POOL\_CREATE 41

6.2. POOL\_DELETE 42

6.3. POOL\_IDENT 43

6.4. POOL\_GET\_BLK 44

6.5. POOL\_RET\_BLK 45

6.6. POOL\_INFO 46

7. SEMAPHORES 47

7.1. SEM\_CREATE 48

7.2. SEM\_DELETE 49

7.3. SEM\_IDENT 50

7.4. SEM\_CLAIM 51

7.5. SEM\_RELEASE 52

7.6. SEM\_INFO 538. QUEUES 54

8.1. QUEUE\_CREATE 56

8.2. QUEUE\_DELETE 57

8.3. QUEUE\_IDENT 58

8.4. QUEUE\_SEND 59

8.5. QUEUE\_JUMP 60

8.6. QUEUE\_BROADCAST 61

8.7. QUEUE\_RECEIVE 62

8.8. QUEUE\_FLUSH 64

8.9. QUEUE\_INFO 65

9. EVENTS 66

9.1. EVENT\_SEND 67

9.2. EVENT\_RECEIVE 68

10. EXCEPTIONS 69

10.1. EXCEPTION\_CATCH 71

10.2. EXCEPTION\_RAISE 73

10.3. EXCEPTION\_RETURN 74

11. CLOCK 75

11.1. CLOCK\_SET 76

11.2. CLOCK\_GET 77

11.3. CLOCK\_TICK 78

12. TIMERS 79

12.1. TIMER\_WAKE\_AFTER 80

12.2. TIMER\_WAKE\_WHEN 81

12.3. TIMER\_EVENT\_AFTER 82

12.4. TIMER\_EVENT\_WHEN 83

12.5. TIMER\_EVENT\_EVERY 84

12.6. TIMER\_CANCEL 85

13. INTERRUPTS 86

13.1. INT\_ENTER 87

13.2. INT\_RETURN 88

14. MISCELLANEOUS 89

14.1. INT\_TO\_EXT 90

14.2. EXT\_TO\_INT 91

A. COMPLETION STATUSES 92

B. MINIMUM REQUIREMENTS FOR OPERATIONS FROM AN ISR 89

C. SUMMARY OF ORKID OPERATIONS 94

D. C LANGUAGE BINDING 96

1. INTRODUCTION

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

The application areas that ORKID aims at range from embedded systems to complex multi-processing systems with dynamic program loading. It is restricted however to real-time environments and only addresses kernel level functionality.

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

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

Sections 3 to 14 describe in detail the various standard types of objects and the operations that manipulate them. There is one section per type of object. Each section contains a general description of this type of object, followed by subsections detailing the operations. The latter are in a programming language independent format. It is foreseen that for all required programming languages, a language binding will be defined in a companion standard. The first one, introduced in conjunction with ORKID, is for the C language. For syntax, the language binding document is the final authority.

The portability provided by the ORKID standard is at source code level. This means that, optimally, a program written for one implementation should run unmodified on another implementation, requiring only recompilation and relinking. Nevertheless it will be possible to write ORKID compatible programs, which rely implicitly so much on the specific behavior of one implementation, that full portability might be endangered.

The syntax of ORKID operation calls in a real implementation will be defined in the appropriate language binding. There will be, however, a one to one correspondence between this standard and each language binding for all literal values, operation and parameter names, types and sequence.2.0 ORKIDCONCEPTS

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

2.1. Environment

The computer system environment expected by ORKID is described by the notion of a system. A system consists of a collection of one or more interconnected nodes. Each node is a computer with an ORKID compliant kernel on which application programs can be executed. To ORKID a node is a single entity, although it may be implemented as a multi-processor computer there is only one kernel controlling that node (see also 2.5 Multi-Procesing).

2.2. ORKID Objects

The standard object types defined by ORKID are:

* tasks : single threads of program execution in a node.
* regions : memory areas for dynamic allocation of variable sized segments.
* pools : memory areas for dynamic allocation of fixed sized buffers.
* Semaphores : mechanisms used for synchronization and to manage resource allocation amongst tasks.
* queues : inter task communication mechanisms with implied synchronization.
* events : task specific event markers for synchronization.
* exceptions : task specific exceptional conditions with asynchronous exception service routines.
* note-pad : task specific integer locations for simple, unsynchronized data exchange.
* clock : current date and time.
* timers : software delays and alarms.

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

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

Pools are consecutive areas of memory organized as a collection of fixed sized buffers which may be allocated to tasks. Pools are simpler than regions and are intended for fast dynamic memory allocation/de- allocation operations. In contrast to the more complex concept of a region pools can be operated on across node boundaries. The maximum number of pools on a node or in a system is implementation dependent.

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

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

Events are task specific markers that allow a task to buffer until an event, or some combination thereof occurs, therefore they form a simple synchronization mechanism. Each task has the same, fixed number of events which is equal to the number of bits in the basic word length of the corresponding processor.

Exceptions too are task specific conditions. Unlike events they are handled asynchronously by the task, meaning that when an exception is raised for a task that task's flow of control is interrupted to execute the code designated to be the exception service routines (XSR). Exceptions are intended to handle exceptional conditions without constantly having to check for them. In general exceptions should not be misused as a synchronization mechanism. Each task has the same, fixed number of exceptions which is equal to the number of bits in the basic word length of the corresponding processor.

Note-pad locations are task specific variables that can be read or written without any form of synchronization or protection. The size of a node-pad location is equal to the basic word size of the corresponding processor. Each task has the same, fixed number of note- pads. The actual number is implementation dependent, but the minimum number is fixed at sixteen. The clock is a mechanism maintaining the current date and time on each node.

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

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

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

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

System Configuration

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

Levels of Compliance

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

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

Globality of objects

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

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

Observation:

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