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.

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 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 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

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 accross 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 Conformence

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	node_ident successful
ILLEGAL_USE	node_ident not callable from ISR
INVALID_PARAMETER	a parameter refers to an invalid address
NAME_NOT_FOUND	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 mode 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.

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.

+ NOPREEMPT — When this value is set, the task will retain control of it's 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 therefor 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 it's 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	kernel defined task identifier
start_addr	: *	task start address
arguments	: *	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.