A Unified Priority-Based Kernel for Ada

T.P. Baker*
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

tpb@sei.cmu.edu

Offer Pazy
Intermetrics, Inc.
733 Concord, Ave.
Cambridge, MA. 02138

pazyo@ajpo.sei.cmu.edu

March 6, 1995

Abstract

This report presents a specification of a bare-bones runtime system (RTS) kernel for real-time usage. The interface is described as an Ada package. The primary intended users of this interface are developers of Ada runtime systems and RTS extensions such as the ARTEWG CIFO. The main reason for developing "yet another kernel specification", and the unique feature of this kernel, is the goal of accommodating independently developed RTS components which share machine resources and scheduling policies. As a result, a newly developed RTS component can co-exist and be interoperable with a standard Ada RTS with no need for internal documentation or changes as long as they both conform to the conventions specified here.

1 Introduction

This report describes the semantics and interface of a Run Time Kernel (RTK) for real-time Ada applications. The purpose of this kernel is to provide a unifying execution model that can be used to explain the interaction of the standard Ada[1] tasking Run-Time System (RTS) with extended runtime library (XRTL) components[4], such as CIFO[3] and ExTRA[6, 7].

1.1 Motivation and Background

The RTK is an outgrowth of the CIFO project of the Ada Runtime Environment Working Group (ARTEWG). The CIFO is a "catalog" of application program interfaces for a collection of XRTL "features and options", which were contributed by ARTEWG participants over a period of several years. Due to the incremental nature of the CIFO development, and its being conceived as a catalog of options, it lacks a consistent execution model. This problem became very apparent at the ARTEWG meeting of August 1990, which was devoted to "cleaning up" version 3.0 of CIFO.

The draft CIFO was found to have many semantic gaps and conflicts, most of which were related to interactions between CIFO features and the standard Ada tasking model, and between different CIFO features. Questions came up, like "What does it mean to suspend a task while it is holding a lock?", and "What happens to a task's priority after it finishes a rendezvous, if the priority was changed while in the rendezvous?". There was no natural and simple way to address these questions.

*Visiting from the Florida State University. This work supported in part by the the U.S. Army CECOM Center for Software Engineering.

The source of the problem is that the CIFO adds a lot of features that are *below* the level of the standard Ada tasking model (which by itself is very consistent and comparatively well defined). This means that the terminology and semantics of the 1983 Ada standard are not adequate to define the semantics of the CIFO or to explain the interactions with Ada tasking.

Several members of the ARTEWG volunteered to attempt to address this problem by defining a low-level kernel interface that would support both the Ada RTS and the CIFO extensions. The goal was also to use the execution model of that kernel as a tool to describe not just the semantics of the higher level primitives (such as rendezvous, semaphores, etc.), but mainly their interactions. It was also envisioned that the adoption of this approach by implementors would facilitate the development of other RTS components without the need of source level integration and testing. This, in turn, would result in the emergence of higher-quality products for the Ada real-time market. The result of this effort is presented in this report.

This interface represents an attempt to minimize the dependencies on the Ada's compiler run-time model, but a complete isolation is not possible. An attempt has also been made to avoid assuming anything special about the architecture of the underlying computer hardware, other than that there is shared memory. Equal consideration has been given to single and multi-processor systems, and to homogeneous and heterogeneous memory configurations. However, this document is still immature, and it is possible (likely) that important cases have been missed.

A notable feature of the RTK is the unified treatment of priorities, which governs interruptibility, resource locking, and processor scheduling. This model, which is similar to that under consideration for Ada 9x [2], aids schedulability analysis, by reducing so-called priority inversion[8], and solves several problems encountered with dangerous race conditions due to attempts to synchronize with interrupt handlers [5, 11].

1.2 Organization

The next section contains a description of the execution model which is the basis of the RTK. Section 3 contains the Ada package specification of the RTK, and Section 4 describes the semantics of the RTK package. Section 5 provides some explanation of the rationale for the design of the RTK. Section 6 is the conclusion.

2 The Execution Model

This section describes the execution model of the RTK, including the concepts of virtual processors, priorities, interrupts, locks, suspension, dispatching and shared memory synchronization. Some rationale is also provided. (More details of the design alternatives considered in the evolution of the RTK, and reasons for the choices made, can be found in Section 5.)

2.1 Virtual Processors

A Virtual Processor (VP) is what the Ada reference manual calls a "logical processor". It is similar to what is sometimes called a "thread" [10], in that both are constructs designed to model software components that may execute in parallel, either by interleaved execution on a single processor or on a shared-memory multi-processor computer system.

The RTK is intended to allow each Ada task to be mapped exclusively and directly to a single VP

for the task's entire life-time. It is expected that when the RTK is implemented "on top" of an existing operating system with light-weight processes, most of the subprograms in this interface will simply provide a thin "facade" between the application and calls to the underlying OS services.

A consequence of this model is that there will be multiple logical "control blocks" for each Ada task, including one for the VP (managed by the RTK), one for the Ada task (managed by the Ada RTS), and possibly others for extensions to the Ada task model (managed by the XRTL). In our view, admitting this fact is not a sign of a wrong approach, but rather, we believe that any possible layering will introduce such duplication. There may be redundant information in the two control blocks, but we expect that the VP control block (VCB) will be very small and simple.

A VP is a very light-weight executing entity with a very limited context. The VP context includes:

- Identification of the VP.
- Hardware processor state, including the program counter and other registers.
- Dispatching state, composed of certain components.
- Two priorities: base and active.
- VP interrupt state and handler information.

All the VP's in a system share access to the rest of the system state, which is assumed to include a shared memory address space.

A VP may execute on different physical processors at different times. While a VP is executing on a physical processor, it inherits the physical processor's hardware interrupt state, including which interrupts are disabled or masked, and hardware interrupt handler information.

2.2 Operations on VP's

The RTK interface provides operations that affect the state of individual VP's. Several of these operations affect the state of the underlying system as well. The RTK operations on VP's include:

- Obtaining the ID of a VP.
- Creating and destroying a VP.
- Setting a user-definable attribute value for a VP, and retrieving the value.
- Suspending the currently executing VP, and causing a suspended VP to resume execution.
- Seizing and releasing non-preemptable resources.
- Obtaining and changing the priority of a VP.
- Associating handlers with interrupts.
- Locking out interrupts.
- Sending a virtual interrupt to a VP.

This set of operations is viewed as sufficient to serve as a basis for implementing the Ada RTS and XRTL for real-time applications. In general, the features of the Ada RTS and the XRTL are not provided directly by the RTK. For example, the CIFO options for suspending another task, waiting for events, generic critical regions, and time services are not provided at this level. Some other features which VP's do not directly support include memory management, rendezvous, queuing semaphores, time-delays, I/O buffers, and file descriptors. We believe that all these can be implemented on top of the RTK efficiently.

The nature of the operations provided by the RTK, and the concern that they be efficiently implementable, dictate that they cannot be made completely safe against incorrect usage. Only those combinations explicitly permitted are to be considered safe.

2.3 Locks

Locks are used to protect non-preemptable resources that are shared between VP's. Before using such a resource, a VP seizes the associated lock. While the VP is holding the lock, the RTK does whatever is necessary to ensure that no other executing VP's or interrupt handlers will seize the lock. When the VP is done using the resource, it releases the lock.

2.4 Unified Priority Model

A unified priority model defines the interactions of the RTK operations that affect processor scheduling, locking of non-preemptable resources, and interrupt processing, by enforcing priorities everywhere.

Every VP, lock, and interrupt handler has an integer priority. (As with Ada tasks, a numerically greater priority value means greater urgency.) The ranges of priorities assigned to VP's, locks, and interrupt handlers are permitted to overlap (e.g. a VP priority may be higher than that of an interrupt handler).

Each VP has two priorities:

1. a base priority;
2. an active priority.

The base priority of a VP is specified at the time the VP is created, and may be modified subsequently by calls to the `Set_Base_Priority` operation. The active priority of a VP is the maximum of the base priority of the VP and the priorities of the set of locks currently held by that VP, and the interrupt handlers being currently executed by that VP.

Similarly, each interrupt handler has a base priority, which is set at the time the handler is bound to the interrupt. (For the default interrupt handlers, this priority is pre-defined.) When the handler is executing, its active priority is the maximum of its own base priority and the priorities of the set of locks it is holding. The priority of an interrupt handler is usually the same as that of the hardware interrupt, if such a notion exists on the specific platform. Though it is not standard in Ada for interrupt handlers to have lower priorities than other program tasks, the choice of whether to enforce this restriction for VP's used to implement tasks is left to the Ada RTS implementation.

Note that it is a consequence of this model that a VP with a high priority can block the execution of a lower priority interrupt handler. This is intentional. If the application has defined the priorities in this manner, not blocking the interrupt would create priority inversion within the application.

Each lock has a priority associated with it, which is a *ceiling* on the active priorities of its users. That is, the priority of the lock is required to be at least as high as the maximum active priority of every VP or IH that will ever attempt to seize that lock. It is the responsibility of the person defining the priority of the lock to ensure that this rule is not violated. It is the responsibility of the RTK implementation to enforce the rule, by raising an exception if a VP attempts to seize a lock while its active priority is higher than the lock's priority.

While a VP or IH is holding a lock, its active priority is raised temporarily to that of the lock (if it is not already that high). An implementation is permitted to simplify this policy by disabling all preemption (or interruption) — which is equivalent to raising the priority ceilings of all locks to the maximum priority of all VP's (or all VP's and IH's).

The active priority is used in dispatching, to allocate physical processors to VP's. (It should also be used for allocating any implementation defined resources.) If there is more than one physical processor, it is not required that every VP be eligible to execute on every processor; however, among the set of VP's eligible to execute on a given processor at a given time the one executing shall have the highest active priority among that set. Also, a VP shall not be able to preempt a processor from another VP unless it has higher active priority than the other VP's active priority.

The unified priority scheduling model is intended to ensure that, while a VP is holding a lock, no other VP that might try to seize that lock will be allowed to execute. To enforce this in the presence of dynamic priority changes, one more restriction is required: VP's that are holding locks must always be given scheduling priority over those that are not.

Beyond this, dispatching within the same active priority is required to be run-until-blocked, with FIFO service of new arrivals. That is, a VP shall be allowed to continue executing until it suspends itself, calls `Yield`, or is preempted by a VP with a higher active priority. A VP resuming after suspension, or calling `Yield`, shall move to the tail of the queue at its priority level, and a VP that is preempted shall retain its position in the queue for its priority. Note that `Yield` operation allows a voluntary form of round-robin scheduling within the basic run-until-blocked model.

2.5 VP Dispatching State

The *dispatching state* of a virtual processor has several conceptual components, each of which may be true or false. These correspond to information about the VP that is known to the RTK implementation, but which may not be stored explicitly in this form and is not directly visible to the user. They are:

1. *Suspendable* – the VP cannot execute unless it is holding locks.
2. *Holding Locks* – the VP is holding one or more locks;
3. *Executing* – the VP is running on some processor.
4. *Destroyed* – the VP has been destroyed.

The combination of these four state components defines the dispatching state of the VP. Not all combinations of state components are possible. In particular, the set of possible states is:

Susp	Holding	Exec	Dest	Possible State
T	T	T	F	Suspendable, holding locks
T	T	F	F	Suspendable, holding locks, ready
T	F	F	F	Normal suspended
F	T	T	F	Executing, holding locks
F	F	T	F	Normal executing
F	T	F	F	Ready, holding locks
F	F	F	F	Normal ready
F	F	F	T	Destroyed

Some useful general rules follow from this table. A VP cannot be Suspendable, not Holding Locks and Executing. Destroyed implies Suspendable, not Holding Locks, and not Executing. Executing implies Holding Locks, or not Suspendable. *Ready* to execute implies not Suspendable, or Suspendable and Holding Locks.

Note that a Suspendable VP does not become *suspended* until all locks are released, and the VP stops executing. That is, for a VP holding locks, suspension is a two-stage process. The first stage is when the VP attempts to suspend itself. The second stage occurs when the VP releases its last lock. This sequence allows a VP to check a data structure under the protection of a lock, and to decide to suspend itself without the danger of a race between the suspension and resumption by another VP.

Another reason for this two-stage suspension is that it allows a VP interrupt handler to suspend the VP that it is interrupting. In this case, the second stage of the suspension takes place on return from the interrupt handler.

2.6 Dispatching Points

Any point at which a VP may change state from Executing to not Executing is called a *dispatching point* for that VP.

Certain RTK operations are defined to be dispatching points for the calling VP. At these points, the RTK is required to determine whether the calling VP is entitled to continue executing according to the dispatching policy, and to switch the processor to another VP if necessary.

There is an exception to this rule when an operation which is defined to be a dispatching point is called from within an interrupt handler procedure (not directly from a VP). In this case, the dispatching point shall apply to the interrupted VP, if any. That is, the dispatching point shall be deferred until the interrupt handler procedure returns, and in the case of "nested" interrupts, the dispatching point shall be deferred until the outermost handler returns.

2.7 Shared-Memory Synchronization Points

Certain points in the execution of a VP are specified as being shared-memory *synchronization points*. These are not the same as the dispatching points. The RTK is designed to be compatible with a shared-memory multiple-processor hardware architecture. With such an architecture, it is possible for the local memory of one processor to become inconsistent with that of another processor. Even with a single processor, when variables are stored in registers the value of a shared variable seen

by one VP may be different than that seen by another VP that reads the value from memory. In order to provide some degree of consistency in the views of shared data between VP's, the concept of synchronization point is introduced.

A *synchronization point* with respect to shared memory between two VP's, T1 and T2, shall be a pair of points P1 and P2 in the executions of the respective VP's, such that the values of data read from shared memory by T2 after the point P2 shall reflect all modifications made by T1 before the point P1. Note that this definition is not symmetric.

This definition of a synchronization point with respect to shared memory is intended to be compatible with the notion of synchronization used in Ada[1] for the effect of rendezvous on shared memory. It is *not* intended to be related to the notion of "synchronization point" that is used in Ada to describe the semantics of task abortion.

2.8 Interrupts, Traps, and VP Interrupts

The term *interrupt* is used generically for three similar but distinct kinds of events:

- hardware interrupts;
- traps;
- VP interrupts.

Hardware interrupts originate from external devices, arrive asynchronously, and are targeted to a particular physical processor. "Pendingness" of a hardware interrupt is part of the state of the physical processor. When a hardware interrupt arrives, which VP it interrupts (if any) depends on chance. Therefore, the occurrence of a hardware interrupt should be transparent to the interrupted VP. A hardware interrupt may also interrupt the RTK itself, and when it does, it must return control to the RTK; an application interrupt handler cannot be allowed to propagate an exception or cause an asynchronous transfer of control directly into the interrupted thread of control (which may be the RTK).

Traps originate from the actions of the executing code (e.g. bus error, FPE, etc.), arrive synchronously, and are usually repeatable and meaningful in the context of the executing code. Like hardware interrupts, the "pendingness" of a trap is a part of the physical processor state, though traps often cannot be blocked. If a trap occurs while a VP is executing, it should not be transparent to the VP. Most such traps will probably be converted to exceptions by the action of the standard Ada RTS. The execution of the RTK implementation is required not to raise traps (at least not any that can be handled by an application); therefore, an application can treat RTK calls as atomic with respect to application trap handlers.

VP interrupts originate from a VP executing the `Interrupt_VP` operation. They are targeted to a specific VP, arrive asynchronously, and interrupt only the targeted VP. A VP interrupt may be pending for more than one VP at the same time, so the "pendingness" of a VP interrupt is a property of the VP. A VP interrupt is not transparent to that VP; in particular, it may need to cause an asynchronous transfer of control in the VP. VP interrupts cannot interrupt the execution of the RTK implementation, so an application can treat RTK calls as atomic with respect to VP interrupt handlers.

The effect of any interrupt may be delayed if the affected VP is executing at an active priority that is higher than that of the interrupt handler.

Interrupts may be handled by procedures which are bound to the interrupt by an operation performed by a VP. For the VP interrupt, the binding of handler procedure to interrupt is specific to the VP which executes the binding operation. In contrast, for hardware interrupts and traps, the binding of the handler procedure to the interrupt is global. The association of instances of interrupts to handlers (i.e. which procedure to invoke) is done at the time the interrupt is delivered, and not when it is sent. For example, suppose some interrupt has a handler `Hdl_1`, and there is an interrupt pending, but because of priority rules it cannot be delivered. If the handler for that interrupt is then replaced by `Hdl_2` before the interrupt is delivered, it is `Hdl_2` that will be invoked when the interrupt is finally delivered. If there is more than one processor, and the hardware routes certain interrupts to specific processors, the `Interrupt_ID` type shall permit separate identification of similar interrupts on different processors, so that a distinct handler may be provided for specific processors.

If an application does not specify a handler for an interrupt, the RTK provides a default handler. For hardware interrupts and traps, the action and priority of this handler shall be implementation defined. For VP interrupts, the priority of the default handler shall be `VP_Priority'first`, and the action need not be defined (see below).

Note that a VP interrupt handler with priority `VP_Priority'first` can never execute, since it will never be higher than the priority of the VP to which the interrupt is targeted, so the action of the handler does not need to be defined. If sent, such an interrupt will remain pending until the VP binds a new handler, with a higher priority.

An interrupt handler is not a VP. Several RTK operations have defined effects on the "calling VP". Therefore, every interrupt handler is viewed as preempting the processor from some VP. During the execution of an interrupt handler, the "calling VP" is considered to be the VP that the handler interrupted. For a VP interrupt, the interrupted VP is the one to which the interrupt was sent. For a trap, or a hardware interrupt handler, the VP is whichever one was executing at the time the trap or hardware interrupt preempted the processor. This raises the possibility that there may have been no application VP running at the time. For traps, this possibility can generally be ignored, since the RTK and any underlying system should be "debugged" enough not to raise traps. For a hardware interrupt, it is virtually certain that the handler will sometimes execute at times when it is not interrupting any application VP. Under these circumstances, the "current VP" is undefined.

During the time the interrupt handler is executing, the interrupted VP is still considered to be logically executing on that processor, so that it cannot be scheduled to execute on any other processor. With VP interrupts, this property is very useful, since it enforces mutual exclusion between the VP and handlers for VP interrupts that are targeted to that VP, and provides a capability for "asynchronous transfer of control". By binding appropriate handlers, one VP can use VP interrupts to signal another VP to raise an exception in itself, suspend itself, or yield control of its processor to other VP's of the same priority.

Another consequence of this model is that if a VP is at a lower active priority than the handler, sending a high-priority interrupt to the VP immediately raises the effective scheduling priority of the VP to the level of the handler, if it is not already that high. Thus, if the VP is not currently executing because it has been preempted, a high enough priority VP interrupt will cause the VP to be scheduled, so that the VP interrupt handler can execute. Of course, the VP will execute the handler, rather than resuming where it was preempted.

2.9 Resource ID's

In the RTK there are certain finite resources, that cannot safely be shared between independently developed RTS subsystems. These resources include VP interrupt ID's, suspension ID's, and VP

attribute ID's. An attempt by two independent subsystems to use such a resource will cause interference between the operations of the subsystems, with chaotic results. For example, if operations of two subsystems attempt to bind handlers for the same VP interrupt, one of the handlers will get the interrupt, and the other will fail. Similarly, if an operation implemented by one subsystem causes a VP to suspend itself, an operation of an independent subsystem should not release the VP with the same suspension ID.

For each of these ID types, the RTK implementation shall provide at least four distinct values. Each RTS subsystem should use no more than one value of each of theses types, and it should be designed to permit changing the specific value easily. For example, this might be done by providing a user-tailorable constant for each value used. This requirement includes the Ada tasking implementation.

Hardware interrupt and trap ID's are also resources that cannot be shared between RTS subsystems. Each RTS component that uses such ID's should document these uses, since they may cause conflicts when the component is combined with other RTS components. However, because hardware interrupts and traps are not interchangeable, conflicts in their usage are not likely to be resolvable by simple user tailoring.

3 Ada Package Specification

The XRTL interface to the RTK shall be by the following Ada package specification. The comments indicate the subsection of this document that describe the semantics of the RTK declarations that immediately follow each comment.

```
with System;
package RTK is
   -- § 4.1 Exceptions and Error Checking
   Optional_Checks_Performed: constant Boolean:= impl.-defined static const.;
   RTK_Error:                       exception;
   Locking_Error:                   exception;
   -- § 4.2 VP ID's
   Max_VP_IDs: constant:= impl.-defined static const.;
   type VP_ID is private;
   Null_VP: constant VP_ID;
   function Is_Valid(VP: VP_ID) return Boolean;
   function Self return VP_ID;
   -- § 4.3 VP Creation and Destruction
   type Init_State is impl.-defined type;
   subtype VP_Priority is Integer range impl.-defined static range;
   Max_VPs: constant:= impl.-defined static const.;
   procedure Create_VP
     (Priority: VP_Priority;
      Initial_State: Init_State;
      Entry_Point: in System.Address;
      VP: out VP_ID);
   procedure Destroy_VP(VP: in out VP_ID);
   -- § 4.4 User-definable VP Attributes
   Null_Attribute: System.Address:= impl.-defined static const.;
   type Attribute_ID is range impl.-defined static range;
   procedure Set_Attribute
      (VP: VP_ID;
       Attribute: Attribute_ID;
```

```
      Value: System.Address);
   function  Get_Attribute
      (VP: VP_ID;
       Attribute: Attribute_ID) return System.Address;
   -- § 4.5 Suspension and Resumption
   type Suspension_ID is range impl.-defined static range;
   procedure Suspend_Self(Suspension: Suspension_ID);
   procedure Resume_VP(VP: VP_ID; Suspension: Suspension_ID);
   -- § 4.6 Locks
   type Lock_ID is limited private;
   subtype Lock_Priority is Integer range impl.-defined static range;
   Max_Locks: constant:= impl.-defined static const.;
   procedure Initialize_Lock (Lock: in out Lock_ID; Priority: Lock_Priority);
   procedure Finalize_Lock(Lock: in out Lock_ID);
   procedure Seize_Lock(Lock: in out Lock_ID);
   procedure Release_Lock(Lock: in out Lock_ID);
   function  Self_Is_Holder(Lock: Lock_ID) return Boolean;
   -- § 4.7 Dynamic Priorities
   procedure Set_Base_Priority(VP: VP_ID; Priority: VP_Priority);
   procedure Set_Base_Priority(Priority: VP_Priority);
   function Base_Priority(VP: VP_ID) return VP_Priority;
   function Base_Priority return VP_Priority;
   procedure Yield;
   -- § 4.8 Interrupts
   type Interrupt_ID is impl.-defined type;
   subtype VP_Interrupt_ID is
      Interrupt_ID range impl.-defined static range;;
   subtype Interrupt_Priority is Integer range impl.-defined static range;
   type Interrupt_Info is impl.-defined type;
   type Handler_Info is impl.-defined type;
   function Current_Priority
     (Interrupt: Interrupt_ID) return Interrupt_Priority;
   procedure Interrupt_VP
     (VP: VP_ID;
      VP_Interrupt: VP_Interrupt_ID;
      Info: Interrupt_Info);
   procedure Attach_Interrupt_Handler
     (Interrupt: Interrupt_ID;
      Priority: Interrupt_Priority;
      Handler_Address: System.Address;
      Info: Handler_Info:= impl.-defined static const.);
    procedure Detach_Interrupt_Handler
     (Interrupt: Interrupt_ID);
private
   type VP_ID is impl.-defined type;
   Null_VP: constant VP_ID:= impl.-defined static const.;
   type Lock_ID is impl.-defined type;
end RTK;
```

4 Semantic Description

This section describes the semantics and intended usage of the RTK operations. The subsections are organized according to groups of related declarations: exceptions and error handling in Section 4.1; VP ID's in Section 4.2; VP creation and destruction in Section 4.3; user-definable VP attributes in Section 4.4; suspension and resumption in Section 4.5; locks in Section 4.6; dynamic priorities in Section 4.7; interrupts in Section 4.8; interrupt handlers in Section 4.9.

4.1 Exceptions and Error Checking

```
Optional_Checks_Performed: constant Boolean:= impl.-defined static const.;
RTK_Error:                      exception;
Locking_Error:                  exception;
```

Description

The RTK may check for certain errors, and raise certain exceptions when the errors are detected. The general meanings of these exceptions are:

1. `RTK_Error` – An RTK operation was called in an illegal context or with an illegal parameter value, or an implementation-defined limit was exceeded.
2. `Locking_Error` – A lock was used inconsistently, or not according to the priority rules.

These exceptions shall be raised only for the conditions described here.

Whenever an exception is raised by an RTK operation, the operation shall have no other effect. In particular, it shall not change the state of any VP or any lock, or consume storage.

In order not to impose excessive overhead on RTK implementations, most error checks are specified as being *optional*. A check is specified as being optional by the phrase "if optional checks are performed" in the corresponding "Error Handling" specification. The RTK implementation shall either perform all the optional checks, and the associated error handling, or shall perform none of them. The value of the `Optional_Checks_Performed` constant shall specify whether the optional checks are performed. If an optional check is not performed, and the error occurs, the effect on the system is undefined. That is, the entire system may fail. If an optional check is performed, the error handling shall be as specified for the operation. For most errors the handling is to raise a specified exception. By using an unchecked RTK implementation, the user shall accept the full burden of using RTK operations correctly.

If an exception propagates to the end of the (main) procedure executed by a VP, it shall result in the destruction of the VP. (See definition of the Destroyed state in Section 2.5, and `Destroy_VP` in Section 4.3.)

4.2 VP ID's

```
Max_VP_IDs: constant:= impl.-defined static const.;
type VP_ID is private;
Null_VP: constant VP_ID;
```

```
function Is_Valid(VP: VP_ID) return Boolean;
function Self return VP_ID;
```

Description

The `VP_ID` type shall be used to identify virtual processors. Each value of the `VP_ID` type shall identify a unique virtual processor, except for the `Null_VP` constant, which does not identify any VP. Variables of the `VP_ID` type shall be implicitly initialized to the `Null_VP` value. Other values of the `VP_ID` type shall (only) be obtained by the `Create_VP` operation.

`Max_VP_IDs` shall be a lower bound on the number of VP's that may be created, *serially*, over the life of a system. Values of the `VP_ID` type shall not be re-used by the RTK implementation; that is, each value of the `VP_ID` type shall not identify more than one VP between system start-up and system shut-down. The range of values of the `VP_ID` type shall be sufficiently large to guarantee that at least `Max_VP_IDs` VP's can be created.

Note that `Max_VP_IDs` is distinct from `Max_VPs`, which is lower bound on the number of VP's that may exist at one instant.

The `Is_Valid` function shall return the value True if and only if the specified ID is valid. A value of the `VP_ID` type shall be *valid* if and only if it identifies some VP that has been created by the `Create_VP` operation, and has not subsequently been destroyed. Note that the value `Null_VP` is not a valid VP ID.

The `Self` function shall return the ID of the calling VP. If `Self` is called from within an interrupt handler it shall return the ID of the VP whose execution was interrupted by the handler, if any. If self is called from within a hardware interrupt handler and there was no VP executing when the interrupt handler was invoked, `Self` shall return an invalid VP ID (which may be `Null_VP`). For efficiency, it is recommended that this operation be implemented as a short in-line operation.

Error Handling

There are no error conditions for these operations.

4.3 VP Creation and Destruction

```
type Init_State is impl.-defined type;
subtype VP_Priority is Integer range impl.-defined static range;
Max_VPs: constant:= impl.-defined static const.;
procedure Create_VP
  (Priority: VP_Priority;
   Initial_State: Init_State;
   Entry_Point: in System.Address;
   VP: out VP_ID);
procedure Destroy_VP(VP: in out VP_ID);
```

Description

The `Init_State` type is used to specify the initial hardware state of a VP. This will be machine-specific, and will typically be a record type including at least the addresses to be loaded into the stack pointer and base registers.

The `Max_VPs` shall be a lower bound on the number of VP's that the system is capable of supporting at one time, if and only if there is such a predefined limit. Otherwise, `Max_VPs` shall be zero.

The `Create_VP` operation shall create a virtual processor. The `Priority` parameter specifies the base priority of this VP. The `Entry_Point` parameter specifies the address of the instruction where the VP is to start executing. This shall be a parameterless Ada procedure. Ordinarily, the procedure should be defined as a library unit, or immediately within a package that is a library unit. This restriction is to prevent "dangling reference" problems, and would not need to be followed where the RTK operations are used by an Ada tasking implementation, when the compiler and tasking implementations are able to ensure against dangling references. The `Initial_State` parameter specifies the values of any other registers that need to be initialized when the VP starts execution, including any information needed by the compiler to reference objects declared in enclosing lexical scopes. The ID of the new VP is returned via the `VP` parameter. This shall be a new value of the `VP_ID` type, not used previously for any other VP.

A newly created VP shall have the state components:

1. Suspendable = true;
2. Holding Locks = false;
3. Executing = false;
4. Destroyed = false.

The suspension ID (see Section 4.5) of the newly created VP shall be a special value such that `Resume_VP` with any suspension ID shall resume that VP. This special value shall not correspond to any value of the type `Suspension_ID`.

The `Destroy_VP` operation shall destroy the VP specified by the `VP` parameter, release any locks held by that VP, and shall recover any system resources allocated by the `Create_VP` operation that created that VP. Before the operation, the specified VP ID should be valid. After the operation, the value of the `VP` parameter shall be set to `Null_VP`, and the specified VP ID value shall become invalid.

Note that destroying a VP asynchronously is possible, but is hazardous, in the same way as is Ada task abortion. The VP may be holding pointers to dynamically allocated (i.e. heap) objects. The storage occupied by such objects may never be recovered.

Note that a VP may destroy itself. An interrupt handler may also destroy the VP that it interrupts. When a VP destroys itself the effect shall be immediate.

Error Handling

`Create_VP` will fail if there are insufficient RTK storage resources available to create a new VP. (This refers to data structures used by the RTK implementation, and does not include stack storage for the VP, which is not a responsibility of the RTK.) `Storage_Error` shall be raised for this error.

`Create_VP` will fail if the range of possible VP ID values has been exhausted (see `Max_VP_IDs`). In this case, `RTK_Error` shall be raised.

`Destroy_VP` should not be called while the VP to be destroyed has local lock ID variables that have been initialized but have not yet been finalized. (This may create dangerous dangling references to lock variables, or reclaimed system storage.) It is not considered feasible to check for this error without special compiler support, so checking is not required, even if optional checks are performed.

`RTK_Error` shall be raised for this error, if the system is able to detect it; otherwise, the effect is undefined.

`Destroy_VP` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the result is undefined.

In general, after `Destroy_VP`, no further reference should be made to the the ID of the destroyed VP, which has now become invalid. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise the result is undefined.

4.4 User-Definable VP Attributes

```
Null_Attribute: System.Address:= impl.-defined static const.;
type Attribute_ID is range impl.-defined static range;
procedure Set_Attribute
   (VP: VP_ID;
    Attribute: Attribute_ID;
    Value: System.Address);
function  Get_Attribute
   (VP: VP_ID;
    Attribute: Attribute_ID) return System.Address;
```

Description

The `Attribute_ID` type shall be used to identify a specific user-definable attribute of a VP. The range of this type is implementation-defined, but shall include at least four values. Each RTS subsystem that makes use of a value of this type shall document this use, and should be designed to permit changing the specific value easily. For example, this might be done by providing a user-tailorable constant for each value used. This requirement includes the Ada tasking implementation.

`Null_Attribute` shall be the default value of every user-definable VP attribute.

The `Set_Attribute` operation shall associate `Value` as the specified user-definable attribute of the specified VP. This value shall replace the previous user-definable attribute of that VP.

The `Get_Attribute` operation shall return the value of the specified user-definable attribute of the specified VP. If the `Set_Attribute` operation has not been used to associate a user-definable attribute value for that VP and attribute ID, `Null_Attribute` shall be returned.

Error Handling

`Set_Attribute` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Set_Attribute` should not be called with an invalid attribute ID. `Constraint_Error` shall be raised for this error, if optional checks are performed (whether or not the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Get_Attribute` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Get_Attribute` should not be called with an invalid attribute ID. `Constraint_Error` shall be raised for this error, if optional checks are performed (whether or not the compiler has suppressed constraint checks); otherwise, the effect is undefined.

4.5 Suspension and Resumption

```
type Suspension_ID is range impl.-defined static range;
procedure Suspend_Self(Suspension: Suspension_ID);
procedure Resume_VP(VP: VP_ID; Suspension: Suspension_ID);
```

Description

The `Suspension_ID` type is used to identify the subsystem (or reason) that caused a VP to suspend itself and to ensure that only a `Resume_VP` operation with a matching ID can undo the suspension.

The `Suspend_Self` operation shall cause the calling VP to become *suspendable*, and associate the specified suspension ID with that VP. If the VP is holding locks, it shall continue executing; otherwise, it shall stop executing immediately. If a VP is suspendable and releases the last lock it is holding, it shall stop executing immediately. If the VP is already suspendable, the operation shall still be permitted, and the new suspension ID shall replace the suspension ID previously associated with that VP. If the VP is not holding any locks this operation is a dispatching point for that VP, at which the VP shall change state from Executing to not Executing. If this operation is called from within an interrupt handler, the effect shall be to suspend the interrupted VP, if any, at its next dispatching point. If the interrupt handler did not interrupt any VP, calling this operation is an error.

The `Resume_VP` operation shall cause the Suspendable state component of the VP specified by the `VP` parameter to be set to False, if the VP is suspendable with the specified suspension ID, or it's suspension ID is the "special" value that is assigned to VP's when they are first created. This operation shall be a dispatching point for the calling VP. If the VP is not suspendable, resuming it is not an error; the operation shall simply have no effect.

Error Handling

`Suspend_Self` should not be called from a context where `Self` is undefined. (This may only happen inside a hardware interrupt handler.) `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Suspend_Self` should not be called with an invalid suspension ID. `Constraint_Error` shall be raised for this error if optional checks are performed (even if the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Resume_VP` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Resume_VP` should not be called with an invalid suspension ID. `Constraint_Error` shall be raised for this error if optional checks are performed (even if the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Resume_VP` should not be called if the VP is suspendable with a different suspension ID, other than the special value assigned to VP's when they are first created. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

4.6 Locks

```
type Lock_ID is limited private;
subtype Lock_Priority is Integer range impl.-defined static range;
Max_Locks: constant:= impl.-defined static const.;
procedure Initialize_Lock (Lock: in out Lock_ID; Priority: Lock_Priority);
procedure Finalize_Lock(Lock: in out Lock_ID);
procedure Seize_Lock(Lock: in out Lock_ID);
procedure Release_Lock(Lock: in out Lock_ID);
function  Self_Is_Holder(Lock: Lock_ID) return Boolean;
```

Description

The `Lock_ID` type shall be used to identify locks. A lock is used to control concurrent access to certain system resources. Each lock shall be in one of the following states:

1. Uninitialized; – same as finalized
2. Locked;
3. Unlocked.

Each object of type `Lock_ID` shall be implicitly initialized to the Uninitialized state. It can make the transition to the Locked state only via the `Seize_Lock` operation, and can become unlocked only through the `Release_Lock` and `Destroy_VP` operations. Each lock that is in the Locked state shall be held by a unique VP. A lock shall only be released by the VP that is holding it. Each lock shall have a fixed priority associated with it. Whenever a VP is holding a lock, its active priority shall not be lower than that of the lock. In particular, the VP holding the lock shall not be able to be preempted from its processor by any other VP that may attempt to seize the same lock. It is an error for a VP to attempt to seize a lock when its active priority is higher than that of the lock. While a VP is holding a lock, it shall remain eligible to execute, even if it is suspendable. If a suspendable VP releases the last lock it is holding, it shall stop executing.

Note that operations which modify lock objects must be implemented atomically, and therefore require the address of the lock object. This implicitly requires that the Ada implementation of "in out" parameters for the Lock type must pass the address of the actual parameter. Insuring atomicity requires that the bodies of such operations must be implemented in a fashion that depends on the underlying machine or operating system.

`Max_Locks` shall be the maximum number of locks that the system is capable of supporting at one time, if and only if there is such a predefined limit. Otherwise, it shall be zero.

The `Initialize_Lock` operation shall initialize the `Lock` to a the Unlocked state, with the specified priority. If `Finalize_Lock` has been call previously for `Lock` and `Finalize_Lock` has not been subsequently called for it, this is an error. Creating a lock object may consume system resources, in addition to local memory resources in the environment where it is declared. Responsibility for the recovery of such resources, via the `Finalize_Lock` operation, is with the user.

The `Finalize_Lock` operation shall restore the `Lock` to the Uninitialized state, and recover all system resources that may have been allocated by the preceding `Initialize_Lock` operation. Performing this operation on a lock that is in the Locked or Uninitialized state is an error.

The `Seize_Lock` operation shall change the state of the lock L to the Locked state, and raise the active priority of the calling VP to the priority of the lock, if it is not already that high. It shall

be implemented in such a way as not to cause the calling VP to stop executing. If the lock is in the Uninitialized state, or is already held by the same VP, or the active priority of the calling VP is higher then the priority of `Lock`, it is an error. This operation is a not a dispatching point. This operation shall be a synchronization point with respect to shared memory, between the VP seizing the lock and the VP that previously held the lock.

Note that locking out VP's executing on the same processor is accomplished by the priority scheme, which prevents the dispatching of all VP's that may contend for the lock. Locking out interrupt handlers must be accomplished by disabling or masking the interrupts whose handlers may contend for the lock. Finally, on multi-processors, a VP that attempts to seize a lock that is held by a VP executing on another processor will be required to "busy-wait" until the lock is released.

The `Release_Lock` operation shall change the state of `Lock` to Unlocked, and restore the active priority of the calling VP to the maximum of its base priority and the priorities of any locks the VP is still holding. If the lock is in the Unlocked state, or the calling VP is not holding the lock, or the lock is in the Uninitialized state, or the lock was not the last one to be seized by the calling VP, this is an error. Locks are required to be released in the reverse order in which they are seized (i.e. LIFO order). This operation shall be a dispatching point for the calling VP, at which the VP may change state from Executing to not Executing. This operation shall be a synchronization point with respect to shared memory, between the VP releasing the lock and the next VP to seize it.

Note that consideration should always be given to protecting the section of code between `Seize_Lock` and `Release_Lock` operations, with an exception handler that contains a `Release_Lock` operation on that lock. Otherwise, there is risk that an exception will be raised and the lock will not be released, leading to system failure. The exception handler should use the `Self_Is_Holder` operation to determine whether to call `Release_Lock`.

The `Self_Is_Holder` function shall return the value `True` if and only if the calling VP is the holder of the specified lock. It may be used in an exception handler, to determine whether a lock was held when the exception was raised.

Error Handling

`Initialize_Lock` will fail if the system is unable to initialize the lock, due to there being insufficient system resources. In this case, `Storage_Error` shall be raised.

`Initialize_Lock` should not be called for a lock that is not in the Uninitialized state. `RTK_Error` shall be raised for this error, if optional checks are performed; otherwise, the effect is undefined.

`Finalize_Lock` should not be called if the lock is in the Locked or Uninitialized state. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Seize_Lock` should not be called if the lock is in the Uninitialized state. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Seize_Lock` should not be called if the lock is already held by the calling VP, or the active priority of the calling VP is higher than the priority of the lock. `Locking_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Seize_Lock` may fail due to a deadlock. (This can happen if there are multiple processors, or due to a previously undetected error.) Even if optional checks are performed, the system is not required to detect this error. `Locking_Error` shall be raised if the system detects this error. Otherwise, the VP's involved in the deadlock will wait, preventing lower priority VP's (and possibly also higher priority VP's) from executing on the same processors.

`Release_Lock` should not be called for a lock that is in the Uninitialized or Unlocked state. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Release_Lock` should not be called if the lock is not held by the calling VP, or the lock is not the last to be seized by that VP, or the lock was seized within an interrupt handler and was not released before return from the handler. `Locking_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Self_Is_Holder` should not be called for a lock ID that is in the Uninitialized state or if `Self` is undefined. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

4.7 Dynamic Priorities

```
procedure Set_Base_Priority(VP: VP_ID; Priority: VP_Priority);
procedure Set_Base_Priority(Priority: VP_Priority);
function Base_Priority(VP: VP_ID) return VP_Priority;
function Base_Priority return VP_Priority;
procedure Yield;
```

Description

The `Set_Base_Priority` operations set the base priority of a VP to a specified value. These operations affect the VP's in whatever state they are, without changing the rest of the state. The version with `VP` parameter sets the priority of the VP specified by the `VP` parameter to the priority specified by `Priority`. The version without the `VP` parameter sets the priority of the calling VP.

The base priorities of the affected VP's, as perceived by the calling VP via the `Base_Priority` function, shall be changed before the operation returns. If the affected VP is the same as the caller, or is not executing and is eligible to execute on the same processor, any effect on its active priority shall also be felt before the operation returns. If the affected VP is executing on a different processor, or is not eligible to execute on the same processor, such an effect shall not occur later than the next time the affected VP reaches a dispatching point. If the affected VP is currently interrupted, and the handler has a lower priority, then the priority shall be raised to the new value in order to complete the handler execution.

A `Set_Base_Priority` operation shall implicitly raise the active priority of the calling VP so that it is not lower than the specified new priority for the duration of the operation. After the operation has completed, the active priority shall revert to its previous value, or to its newly set value if the executing VP set its own priority. Completion of `Set_Base_Priority` shall be a dispatching point for the calling VP. Calling the version of `Set_Base_Priority` that implicitly applies to the calling VP from an interrupt handler is an error if `Self` at that point is undefined.

Note that if the base priority of a VP is lowered while it is holding locks or executing an interrupt handler, this change in the base priority does not affect the active priority of the VP until it releases all locks and returns from any interrupt handlers. Note also, that the rules in Section 2.5 apply to the scheduling of a VP whose priority has been changed, in relation to other VP's of the same active priority.

The `Base_Priority` functions shall return the base priority of the specified VP. For the form without the `VP` parameter, the priority returned shall be that of the calling VP. If the value of `VP` is invalid, it is an error.

Note that the `Base_Priority` functions are dangerous. A concurrent `Set_Base_Priority` operation performed on another processor or by an interrupt handler may render the value returned by either of these functions invalid. Therefore, they should only be used in situations where the application design guarantees that such concurrent changes cannot occur or will not cause errors.

The `Yield` operation is intended to permit voluntary round-robin sharing of processors between VP's of equal priority, by causing the calling VP to yield its physical processor to an eligible VP if one exists. If the calling VP is not holding any locks, the effect shall be to move the calling VP to the end of the queue of VP's with the same active priority. If the calling VP is holding locks, the effect of the `Yield` is deferred until the last held lock is released. This operation does not suspend the VP, or change its base or active priority. It operation shall be a dispatching point for the calling VP. If this operation is called from within an interrupt handler, the effect shall be to cause the interrupted VP, if any, to execute the `Yield` operation at its next dispatching point. If `Yield` is called where `Self` is undefined it is an error.

Error Handling

`Set_Base_Priority` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Set_Base_Priority` should not be called with an invalid priority. `Constraint_Error` shall be raised for this error if optional checks are performed (regardless of whether the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Set_Base_Priority` should not be called without a `VP` parameter when `Self` is undefined. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Base_Priority` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Base_Priority` should not be called without a `VP` parameter when `Self` is undefined. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Yield` should not be called when `Self` is undefined. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

4.8 Interrupts

```
type Interrupt_ID is impl.-defined type;
subtype VP_Interrupt_ID is
   Interrupt_ID range impl.-defined static range;;
subtype Interrupt_Priority is Integer range impl.-defined static range;
type Interrupt_Info is impl.-defined type;
type Handler_Info is impl.-defined type;
function Current_Priority
  (Interrupt: Interrupt_ID) return Interrupt_Priority;
procedure Interrupt_VP
  (VP: VP_ID;
   VP_Interrupt: VP_Interrupt_ID;
   Info: Interrupt_Info);
procedure Attach_Interrupt_Handler
  (Interrupt: Interrupt_ID;
   Priority: Interrupt_Priority;
```

```
    Handler_Address: System.Address;
    Info: Handler_Info:= impl.-defined static const.);
  procedure Detach_Interrupt_Handler
   (Interrupt: Interrupt_ID);
```

Description

Values of the `Interrupt_ID` type shall be used to identify interrupts.

The `VP_Interrupt_ID` subtype shall identify a range of (conceptual) interrupts that can be raised by a VP in itself or another VP, but cannot be raised by the hardware.

The `Interrupt_Priority` subtype shall identify the range of priorities that may be associated with interrupts. Each interrupt shall have an associated priority, which shall be defined by the hardware and system. In some extreme situations VP's may be able to have higher base priority than hardware interrupts, or all interrupts may have a single priority, which is higher than that of any VP.

The `Interrupt_Info` type shall be used to pass additional machine dependent and interrupt dependent information to the interrupt handler.

The `Handler_Info` type shall be used to specify additional machine-dependent and interrupt-dependent information about the interrupt handler. For example, this type may be used to specify whether the handler is to execute in privileged mode, or whether specific interrupts are to be masked or unmasked.

The `Current_Priority` function shall return the priority of the current handler which is bound to the specified interrupt. This shall be the priority at which the interrupt attempts to preempt a physical processor, and shall be the same as the active priority at which a handler for that interrupt executes, subject to the further priority effects of locks which that handler may acquire. In the absence of any other active binding, the priority of the default binding for that interrupt shall be returned.

The `Interrupt_VP` operation shall cause the specified VP interrupt to be raised in the specified VP, subject to the rules of priorities. The `Info` parameter shall specify machine-dependent and interrupt-dependent information to the interrupt handler; it may be omitted for certain implementations. If the interrupt cannot be handled immediately, because the specified VP is not executing or has higher active priority than the interrupt, the interrupt shall remain pending until it can be handled. If the specified VP has not attached any handler for the interrupt, the default handler shall apply (which has priority too low to interrupt the VP). While an interrupt is pending, any subsequent arrivals of instances of the same interrupt shall be ignored; that is, the RTK shall not provide any queuing of interrupts. If more than one VP interrupt ID is pending, they shall be handled in an order consistent with interrupt priorities. If the value of the `VP` parameter is invalid, it is an error. In this case, if optional checks are performed, the operation shall simply be a dispatching point, and shall have no other effect. In any case, this operation shall be a dispatching point for the calling VP.

The `Attach_Interrupt_Handler` operation shall bind the subprogram with the entry point specified by `Handler_Address` as the handler for Interrupt. The `Interrupt_Priority` parameter specifies the base priority of the handler execution, as well as the ability of the interrupt to cause preemption. The `Info` parameter may be used to specify additional machine-dependent information about the handler, such as whether it is to execute in privileged mode or whether certain interrupts are to be masked or unmasked; this parameter may be omitted, for certain implementations. If there is a handler already attached to the interrupt, it is replaced by the new handler, with the new priority and other information. Multiple interrupts may be attached to the same handler. The combinations of priorities and interrupts supported will depend on the hardware architecture. All such limitations

shall be documented. The system should (but is not required to) raise an exception whenever these limitations are violated.

The `Detach_Interrupt_Handler` operation shall restore the system default handler for the specified interrupt. Any pending interrupts shall be preserved.

Error Handling

`Interrupt_VP` should not be called with an invalid VP ID. `RTK_Error` shall be raised for this error if optional checks are performed; otherwise, the effect is undefined.

`Interrupt_VP` should not be called with an invalid VP interrupt ID. `Constraint_Error` shall be raised for this error if optional checks are performed (regardless of whether the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Attach_Interrupt_Handler` may fail if the RTK does not permit a user handler to be attached to the specified interrupt. `RTK_Error` shall be raised for this error (regardless of whether optional checks are performed).

`Attach_Interrupt_Handler` should not be called with an invalid VP interrupt ID. `Constraint_Error` shall be raised for this error if optional checks are performed (regardless of whether the compiler has suppressed constraint checks); otherwise, the effect is undefined.

`Attach_Interrupt_Handler` may fail if the RTK does not support the specified handler priority for the specified interrupt. `Constraint_Error` shall be raised for this error (regardless of whether optional checks are performed).

4.9 Interrupt Handlers

```
procedure Interrupt_Handler
     (Interrupt: Interrupt_ID;
      Info: Interrupt_Info);
```

Description

An interrupt handler procedure shall have the form specified above. It shall not be declared within another procedure, task, or block statement; that is, it may only be declared as a separate compilation unit, or within a package that is a library unit. Additional compiler-specific measures, such as the use of pragmas, may need to be taken to ensure: that the handler procedure can be called via its address; that there is no problem due to elaboration-checking code associated with the procedure; that the procedure code is not deleted during linkage, even though it may not appear to be called from within the Ada program.

When the handler is invoked, the `Interrupt` parameter shall be the ID of the interrupt that caused the handler to be invoked, and the `Info` parameter shall be any implementation-defined data associated with the interrupt. The `Info` parameter may be omitted, for certain implementations.

When invoked, the handler shall execute with active priority at least equal to the priority of the interrupt; in effect, it is the same as if it is holding a lock of the same priority as the interrupt while it is executing. One consequence of this rule is that, an interrupt is not permitted to interrupt a handler that is executing for a preceding occurrence of the same interrupt. The system shall manage preemption, and disabling and masking of interrupts to ensure this behavior. If the physical processor

makes a distinction between user mode and other (privileged) modes, it shall be implementation-defined for each hardware interrupt and trap whether the handler shall execute in a normal user mode or a privileged mode. This may depend on the `Info` parameter of `Attach_Interrupt_Handler`. VP interrupt handlers shall execute in normal user mode.

Note that the user may manipulate the interrupt state (e.g. unmask, enable) from within an interrupt handler using implementation specific mechanisms. However, this is not likely to be portable to other systems. Furthermore, doing so may violate the assumptions of the RTK regarding the interrupt state and the priority level in which the program runs. It is therefore the responsibility of the user to ensure that system-wide invariants are not affected by these actions. For example, suppose an application requires an interrupt handler running at priority 7 to enable, temporarily, an interrupt of priority 2 (lower). The designer of the interrupt handler must then ensure that no violation of locks usage will result, since the RTK is not in a state to detect it. The result of violating these invariants is undefined.

While the handler is executing, if the handler interrupted a VP, the interrupted VP shall remain logically "executing" on that physical processor. In particular, if there is more than one physical processor, it shall not be possible for a an interrupted VP to resume execution on another processor before the handler that interrupted it has returned.

Only the following RTK calls shall be permitted from within an interrupt handler:

- `Self`;
- `Is_Valid`;
- `Max_VPs`;
- `Set_Attribute`;
- `Get_Attribute`;
- `Seize_Lock`;
- `Release_Lock`;
- `Self_Is_Holder`;
- `Max_Locks`;
- `Resume_VP`;
- `Suspend_Self`;
- `Interrupt_VP`;
- `Set_Base_Priority`;
- `Base_Priority`;
- `Yield`.

When called from within an interrupt handler these operations shall have the normal effect, except that:

1. Where the description of the operation refers to the "calling VP" it means the interrupted VP; that is, the one that would be specified by the value returned by `Self`. Calling such an operation is an error if the VP ID returned by `Self` is undefined, as it would be for a hardware interrupt handler. The error handling for this situation is specified separately for each operation.

2. Where the description of the operation says that it is a dispatching point for the calling VP, the dispatching point shall be deferred until control returns to the interrupted VP, if any. If the interrupted VP was executing an RTK operation, the dispatching point shall be deferred until the operation completes.

Within the handler procedure, the `Seize_Lock`, `Release_Lock` and `Self_Is_Holder` operations may be used (with care) across processors, or to mask out higher priority interrupts and VP's. Since locks are required to be released in a LIFO order, and since executing a handler is like holding a lock, if a lock is seized in the handler it is an error to fail to release it before the handler returns. The `Resume_VP` and `Interrupt_VP` operations may be used to notify VP's of significant events. The `Suspend_Self` operation may be used to suspend the interrupted VP, if any, and the `Yield` operation may be used to impose round-robin time-slicing within priority ranges.

If a user handler for a VP interrupt or trap attempts to propagate an exception, doing so shall cause that exception to be raised in the interrupted VP. The exception shall be raised at the point where the VP was executing when it was interrupted by the VP interrupt or trap. (Note that the points where VP interrupts may occur are limited by the active priority of the VP, and may be further limited by the RTS and compiler implementation.) When an exception is propagated from a trap handler to the interrupted VP, and the implementation has caused the handler to execute in privileged mode, it shall be the responsibility of the RTK implementation to ensure that the processor is back in normal user mode.

User handlers for hardware interrupts are required to be "transparent" to the RTK. To enable the application to respond quickly to critical events, the RTK implementation should generally allow hardware interrupts during RTK operations wherever this is practical. In turn, user handlers for hardware interrupts must return normally, so that any interrupted RTK operation can complete. It is an error for a user handler for a hardware interrupt to propagate an exception, or call any RTK operation, such as `Yield`, that is defined to apply to the calling VP. The preferred implementation behavior in the face of such errors is to ignore the attempted propagation or erroneous RTK operation and resume whatever processing was interrupted, but this may not always be practical to implement.

Error Handling

A hardware interrupt handler should not return before it has released any locks that it seized. The treatment of this error is specified in Section 4.6.

A hardware interrupt handler should not attempt to propagate an exception. The treatment of this error shall be to return from the interrupt handler, with no effect on the interrupted VP, if optional checks are performed; otherwise, the effect is undefined.

A hardware interrupt handler should not call any RTK operation that is defined to apply to the calling VP. `RTK_Error` should be raised for this error, if optional checks are performed; otherwise, the effect is undefined.

An interrupt handler should not call any operations other than those in the list of permitted operations. `RTK_Error` shall be raised for this error, if optional checks are performed; otherwise, the effect is undefined.

5 Rationale

This section describes some of the alternatives that were considered in the evolution of this interface, and reasons for the choices that were made.

5.1 Ada Package Specification

We have intentionally chosen to specify one monolithic package instead of multiple small ones. This was done to allow for maximum visibility among the RTK components in order to achieve maximum efficiency without worrying too much about information hiding, types conversions, and procedural interfaces (within the RTK).

5.1.1 Arbitrary Limits

Certain arbitrary limits are imposed on implementation-defined ranges. The current limits are that there be at least four values supported for:

1. VP attributes;
2. VP interrupts;
3. suspension ID's

The number four was chosen as a minimum lower bound. The RTK was designed to support RTS extensions. It was generally felt that each RTS extension could probably get by with using only one of each of these. The number four is therefore a lower bound on the number of independent RTS extension components that can be supported.

Some of the ID's are specified as types and some as subtypes. In both cases, the compiler will not allow assigning out-of-range values to objects of those types. We have chosen to require the RTK to raise a `Constraint_Error` in those cases when it detects such an illegal value (as when the user has suppressed checks). The choice of `Constraint_Error` was due to the fact that we wanted it to be the same as the exception the compiler might raise, so a user will not have to be worried about two distinct exceptions for the same kind of error.

5.2 Exceptions and Error Checking

5.2.1 Error Handling

It was not clear what to do when the user of an RTK operation commits an error. The basic problem is that the RTK is intended to be a low-level package. This means:

1. It will ordinarily be used only by packages in the extended run-time library, or by compiler-generated code; we do not expect these packages or the compiler to make errors.
2. The overhead of checking, and providing recovery code, is not tolerable.

Still, just enumerating erroneous usages did not seem adequate, so the exception mechanism was chosen.

Checking for run-time errors was made mostly optional, in order to allow very efficient implementations. However, based on the reasoning that unreliable or inconsistent raising of exceptions is just about useless, or maybe even dangerous, we only allow the two extreme options:

1. Do all optional checks, and raise all exceptions as specified.

2. Don't do any optional checks, and never raise any exceptions.

We have had difficulty deciding exactly which exception to raise, and whether to attempt some form of recovery other than raising an exception (i.e. suppressing the erroneous operation). The present assignment of recovery actions is based on the assumption that it is better to raise an exception immediately than to ignore the error, once an error is detected. Regarding which exceptions to raise, we have taken the point of view that it is not necessary to uniquely identify each cause of error by a separate exception.

5.3 VP ID's

The reason for separating the concept of VP from that of task, and therefore the introduction of a VP ID type distinct from the task ID type of the CIFO and the Ada tasking RTS, is to permit the creation of VP's that are not Ada tasks. The VP is intended to be a "lighter weight" entity than an Ada task, one that does not carry all the semantic restrictions and implementation overhead of Ada tasks.

The `VP_ID` type is not limited, since in order to use the operations on VP's to build the RTS or XRTL packages it will be necessary to store VP ID's in various data structures (e.g. queues).

We have in mind that the `VP_ID` type would be implemented as a combination of pointer (or integer index) with a generation number. (For example, a 64-bit ID might include 16 bits of pointer and 48 bits of generation number.) An invalid (i.e. dangling) VP ID can be detected by mis-match of the generation number encoded within the ID with the generation number recorded within the corresponding VP control block.

For example, suppose we know we can't create more than 800,000 VP's per second and assume that a system will not run continuously for more than 10 years. An implementation could represent a VP ID in 64 bits, as a pair consisting of 16-bit VP control block (VCB) offset and a 48-bit generation-number of the VP relative to that VCB.

5.4 VP Creation and Destruction

5.4.1 Create_VP

The procedure provided as an entry point for the VP is required to be defined as a library unit or immediately within a package that is a library unit. This is because the RTK does not recognize nested scope structures, or keep track of the lifetimes of execution environments. Allowing nested procedures to be used could cause dangling reference problems. For example, if the VP procedure were declared locally within another procedure and referred to objects declared within that procedure, the VP would run into trouble when the lexically enclosing procedure returned, and the objects local to the procedure were deallocated. This restriction is needed only for extended run-time library packages that may use the RTK. In fact, since Ada allows nested tasks, if the RTK is used as a basis for implementing an Ada tasking system, nested procedure-like constructs generated by the Ada compiler will need to be used. If nested procedures are used, specifying the environment (e.g. via a so-called static link, or display), can be done via the initial state parameter.

5.4.2 Destruction *vs.* Termination

We considered adding a new VP dispatching state, so a VP could be "killed" (e.g. by an unhandled exception) but still have a valid ID, so that when other VP's executed operations on that ID the result would be defined. This is similar in concept to an Ada task being "completed" but not yet "terminated". However, adding such a state did not really solve the problem of sudden VP death, and it made the interface more complicated. It would just postpone the dangling reference problem, until we eventually recover the storage of the VP. It does not completely remove the need for internal checking either, since at least the `Resume_VP` operation needs to check whether the VP is alive or dead, as well as whether the ID is valid.

5.5 User-Definable VP Attributes

We have decided to define the type of the user attributes as System.Address not so much because of their "generic" nature, but rather to allow their usage as pointers to user-defined structures.

5.6 Suspension and Resumption

5.6.1 Suspension ID's

The `Suspension_ID` type was added in order to provide protection against unintended resumption of a VP. That is, we do not want a VP that suspended itself using an operation in some RTS package (e.g. for Ada rendezvous) to be resumed accidentally by the action of some add-on package. This feature is intentionally weaker than queuing constructs, such as POSIX "condition variables" and classical semaphores, in order to reduce the implementation burden. If queuing operations are needed, the user can associate a queue and lock with the suspension ID, and do something like this:

```
procedure Block is
begin
  Seize_Lock(L);
  Enqueue(Self);
  Suspend_Self(Suspension);
  Release_Lock(L);
end;

procedure Un_Block is
  VP: VP_ID;
begin
  Seize_Lock(L);
  Dequeue(VP);
  Resume_VP(VP,Suspension);
  Release_Lock(L);
end;
```

We had a problem with defining the suspension ID of a newly created VP (since it is created in the Suspendable state). Since it is the responsibility of the user to initially resume that VP, we had to decide on the appropriate ID. We could allocate a dedicated ID (such as `Suspension_ID'first`) for this purpose, but that would require each extension component to deal with two ID's. It would also remove one value from the set of available ID's. Instead, we have decided to define the suspension ID of a newly created VP, as being a "special" value, so that a `Resume_VP` with *any* suspension ID will affect that VP and resume it. Thus, each component will deal with its "own" ID only and not worry

about this special one. Since there is no way to suspend with such a special ID, this arrangement does not pose any danger of confusing the ID's.

5.6.2 Only Suspending Self

There was pressure from some ARTEWG members for the capability for one VP to suspend another, asynchronously. This idea was rejected for the RTK, due to difficulty defining the interaction with other features, and concern for the implementation complexity. For example, on a multiprocessor system, in order to ensure that the VP being suspended had actually stopped executing, it would be necessary for the VP executing the `Suspend` operation to wait; this would make the operation into a synchronization operation on the same order of complexity as rendezvous, which would be contrary to the objective of keeping the RTK "low level".

5.6.3 Two-stage Suspension

There was some controversy over the idea of a VP continuing to execute after it had "suspended" itself. Some would have preferred the effect to always be immediate. To still be useful, the `Suspend_Self` operation would then have to release all locks, atomically with the suspension, or there would be risk of dangerous races with the VP that executes the resume operation. Releasing all locks would also have been necessary in order to maintain the essential implementation invariant of the non-queuing model of resource locking, that a VP cannot be suspended while holding locks. One problem with this solution is that it breaks the visible pairing of locking and unlocking operations, and there would be no checking for unintentional release of locks. It would also preclude the use of self-suspension within a VP interrupt handler to suspend the interrupted VP.

5.7 Locks

The Lock type is limited, so that the implementation can allocate storage for a lock anywhere (including on the stack), and keep everything in that storage, without worrying about assignment operations clobbering it. If a user wants references to locks, he can use an array of locks (reference=index) or pointers to locks. The implementor can choose whether to make the declared object actually *be* the lock (i.e. hold the lock state), or make it a number or pointer that identifies some lock data object in the RTS storage. Which way it is done will not be visible to the user.

We provide initialize and finalize operations for lock ID's, so that system storage associated with the lock can be recovered. This leaves a danger of dangling references, but that seems to be an unavoidable penalty for insuring that all storage can be recovered.

The locking operations use `in out` parameter passing. However, for some Ada compilers one may need to use `'address`.

5.7.1 No Lock Breaking

Lock breaking operations have been omitted, on the assumption that it is not safe to break a lock so long as the holder is running. Thus, the only way to break a lock is to use a VP-interrupt on the holder, if it has made this possible by binding a handler that will release the lock.

5.7.2 Unification of Locks and Priorities

The operations provided by the RTK are closely related and tightly intertwined. This is a consequence of inherent interactions at the implementation level, between control over interrupts, locking, and processor scheduling. Two different approaches to handling these interactions were considered:

1. The low level operations can be viewed as semantically independent. A set of design rules can be specified to allow the creation of a safe and reliable system, by avoiding dangerous interactions. The user must enforce the rules. For example, if a lock is shared between an interrupt handler and the rest of the program, the rule might be that the interrupt must be disabled before the program attempts to seize the lock. There would need to be many other such rules.
2. The interactions of the low level operations can be completely specified, in a way that is safe and easily understood, by imposing a single unified priority scheme.

We have chosen to take the second approach, combining the semantics of locks with priorities. This approach allows for a clean unifying model for the scheduling, dispatching, and preemption control in the RTK. Separating the locking and interrupt mechanisms from priorities would require a much more complicated semantic definition (at the RTK interface level) to address the various issues of dynamic priority management and priority inheritance.

5.7.3 No Ceiling Changes

We decided not to provide an operation to change the ceiling priority of a lock for mode-changes. Currently, one must finalize and re-initialize the lock, or use a different one in the new mode. Allowing the lock's ceiling to change while VP's might be using a lock (and while VP priorities might be changing concurrently) was considered too costly to implement correctly. The solution to the need for changing ceilings is to choose the ceiling priority of each lock high enough to accommodate the highest potential priority of each VP that may use the lock.

5.7.4 Relaxing the Ceiling Rule

The present ceiling rule for locks is that the *active* priority of a VP attempting to seize a lock not be higher than the (ceiling) priority of the lock. This property, which is essential to the non-queuing priority-based locking mechanism used between VP' on the same processor, is hard to verify by code inspection, since the active priority of a VP may change. There does not seem to be any good way around the effects of explicit changes to VP's' base priorities, other than defining lock priorities as the maximum anticipated priority of VP's that may use the lock. In contrast, there is a convenient solution to the changes due to inheritance. If we can guarantee that a VP can never use inherited priority to preempt another VP, we can use the base priority in the ceiling rule, instead of the active priority. This makes static checking for violations of the rule much easier.

We considered two ways to ensure that an executing VP can never be preempted on the basis of inherited priority. The direct solution is a more restrictive preemption test: that a VP not be allowed to preempt another VP unless its base priority is higher than the other VP's active priority. An indirect solution is to not allow a VP to migrate to a different processor while it is holding locks. So far, we have held back from these changes, on the grounds that the present rule is more obviously correct, and seems less likely to be invalidatde if we add a more general priority inheritance mechanism.

5.7.5 LIFO Lock-Releasing

LIFO order of unlocking is required in order to allow more efficient implementation of locks, through the use of a stack structure to save and restore active priorities, and to prevent long-duration blocking through "chaining" of overlapping critical sections.

5.8 Dynamic Priorities

5.8.1 No Active Priority Function

The `Active_Priority` function was intentionally deleted, since it would preclude an efficient implementation of priority ceilings and priority inheritance, in which the dispatcher uses base priority and recorded inheritance relationships directly, rather than explicitly computing active priorities.

5.8.2 Changing Priority

The active priority of the VP calling `Set_Base_Priority` is raised, temporarily, to the maximum of its own previous active priority and the new priority specified by the parameter. It intentionally does not include the current active priority of the affected VP in this maximum, for implementation reasons. Including this priority would add complexity to this primitive. Unlike the new priority, which is immediately available via the parameter, the active priority of the VP's could only be obtained by visiting the affected VP. This would probably involve locking other data structures, worrying about order of locking, and other details. Since the calling VP can still raise its own priority explicitly, the feature of raising the calling VP's priority as part of the operation is only a convenience. Note that a VP who wants to lower the priority of a higher-priority VP can still raise its own beforehand.

Earlier, we included a version of `Set_Base_Priority` that applied changes to a list of VP's, as a single operation. This was deleted, because it gave an illusion that this would be an atomic transaction, whereas we believe that implementing it as such would require global locking on a scale and for a duration that would seriously hurt response time. An application needing to change many priorities at once can assure atomicity with less locking overhead, since it can enforce the restriction that all such changes be done by a single VP.

5.9 Interrupts

At first, we tried to treat hardware interrupts and traps exactly the same as VP interrupts. After more analysis, we decided that there are significant enough differences in the underlying mechanisms to justify some distinctions in the interface.

We have not included the lower level operations on interrupts which are in the ExTRA and CIFO proposals (e.g. masking and unmasking, enabling and disabling, clearing, acknowledging). To include these operations seems counter to the effort we have made to combine priority and interrupt masking into a single lock concept. Moreover, almost any use of such low-level operations on interrupts without inside-knowledge of the RTK implementation would risk breaking it. Another reason for leaving these primitives out is that they are very system specific. We would be forced to either leave out operations important to certain machines or include operations that could not be implemented. They are not required for the completeness of the model and services that we provide here.

We assume that these capabilities will be available, if needed, in some implementation defined form (e.g. via machine-code inserts) or via some other packages in CIFO and ExTRA.

5.9.1 Interrupt Priorities

We have taken the viewpoint that the priority of an interrupt for preemption is the same as the execution priority of the handler. There has been disagreement within the group whether this restriction is needed. It seems to make sense that the handler's execution priority might be higher than the hardware preemption priority; in the extreme, a handler for a low priority interrupt may need to run with ALL interrupts disabled. Of course, one can certainly raise the priority of the handler as high as one wants after it starts to execute, but on some architectures this can be done more efficiently as part of the initial transfer to the handler.

The RTK does not have to impose any specific order on the priorities of interrupts, but they must be prioritized. This limits the ability of the RTK to support applications with non-linear interrupt behavior. For example, suppose an application wants to allow interrupt 3 to be active when interrupt 5 is active, but wishes interrupts 1-5 to all be blocked when interrupt 6 is active. Currently, this can only be done using implementation-specific extensions, and at the risk of breaking the RTK. If a better solution can be found, it should be considered.

5.9.2 Return and Dispatch

Presently, we assume that return from every user interrupt handler is a dispatching point. That is, if the handler performs any operation (such as `Resume_VP` or `Suspend_Self`) that changes the dispatching state of a VP, return from the handler must be through the dispatcher. This may be considered too much overhead. If so, a "return and dispatch" operation would need to be added. This would be called by interrupt handlers that want to exit through the dispatcher; other interrupt handlers would then exit without a dispatching point. We have left this out of the document, for simplicity. It seems not to be too much overhead to let the RTK set a flag in operations that change the dispatching state of a VP, and call the dispatcher during the return-from-interrupt only when this flag is set.

5.9.3 Interrupts not Queued

The choice not to queue repeated occurrences of interrupts of the same ID is based on the premise that the RTK provides direct linkage to the underlying hardware interrupt mechanism. Hardware typically is capable of latching and holding only one occurrence of an interrupt, and further interrupts are then ignored. Many devices will not interrupt again until the CPU acknowledges the previous interrupt. Moreover, if an attempt were made to queue interrupts in software, some limit would need to be imposed, due to memory requirements. For the latter reason, and for uniformity, the rule that is applied to VP interrupts is the same as the one that is applied to hardware interrupts and traps.

5.9.4 Default Actions for VP-Interrupts

We originally preferred treating the arrival of an unforeseen VP interrupt as a fatal error, which would kill the VP and release all its locks. The big problem with this was how to suppress it, when an application wants to ignore unwelcome VP interrupts. A VP would have to execute a binding operation to replace the default handler, but it might receive an interrupt before it can execute this

handler. Therefore, we chose to provide a default handler, with priority so low that the interrupt handler would never execute. This has the effect of preserving any interrupt that comes in while this default handler binding is in force.

5.9.5 Multiple VP Interrupts

The RTK is required to support several distinct VP interrupt ID's, so that different RTS subsystems (e.g. tasking implementation, and CIFO extensions) do not need to share the use of a single VP interrupt. This does require documentation of VP interrupt usage, and tailoring of any "later" addition to use an interrupt that is not yet in use, but does make such additions possible.

5.10 Excluded Topics

5.10.1 Assigning VP's to Processors

There needs to be some way to limit the set of physical processors a given VP is allowed to execute on. For now, the document takes the point of view that this would be the kind of addition that is specific to a hardware architecture or implementation. We anticipate that implementations or other standards will provide these capabilities. We believe that they can be added to the current interface set without affecting its semantic consistency.

For example, one might add a package like this:

```
with RTK;
package Processor_Management is
   type Processor_ID is ...;
   procedure Enable_for_Processor(VP: VP_ID; P: Processor_ID);
   procedure Disable_for_Processor(VP: VP_ID; P: Processor_ID);
end Processor_Management;
```

5.10.2 General Priority Inheritance

There is no support for priority inheritance between VP's. We do believe inheritance to be important, and even mandatory for certain applications, and we hope that users who build blocking constructs such as semaphores using the RTK consider some form of priority inheritance. Unfortunately the cost of implementing inheritance must be paid in the most time critical parts of the RTK. We do not wish to penalize those systems which do not require inheritance by mandating its inclusion.

If priority inheritance is required, several implementation choices are available. Inheritance can be implemented by simply changing VP Base Priorities as needed. This approach can be expensive, as it can introduce extra dispatching points into the system, and tricky book-keeping as well. These problems are aggravated by the need for many-to-one inheritance in Ada tasking, as during task activation or while a completed master task is waiting for its children to terminate.

An alternative approach, might use a package as follows:

```
with RTK;
package RTK_Inheritance is
   procedure Create_Relationship(Grantor, Receiver: RTK.VP_ID);
   procedure Destroy_Relationship(Grantor, Receiver: RTK.VP_ID);
end RTK_Inheritance;
```

This package allows the RTK to build a graph to describe the inheritance relationships between VP's. Considering each VP as a node in a directed graph, `Create_Relationship` adds an edge between `Grantor` and `Receiver` and `Destroy_Relationship` deletes it. Such a graph can allow the RTK to perform inheritance by traversing the graph during dispatching, by following wait-for links from the VP with the top base priority until an unblocked VP is found. Extra dispatching points are avoided, and many-to-one inheritance can be supported.

For example, here is an implementation of blocking semaphores using this package, which is due to Bruce Jones [9], of the ARTEWG:

```
with RTK;
package Semaphores is
   type Semaphore is private;
   procedure Acquire(S: in out Semaphore);
   procedure Release(S: in out Semaphore);
private
   type Semaphore is
      record
         RTK_Lock: RTK.Lock;
         Q: VP_Q_Type;
         Is_Locked: Boolean:= FALSE;
         Owner: RTK.VP_ID;
      end record;
end Semaphores;

with RTK;
with RTK_Inheritance;
package body Semaphores is
   ...declarations...
   procedure Acquire(S: in out Semaphore) is
   begin
      RTK.Seize_Lock(S.Lock);
      if not S.Is_Locked then
         S.Is_Locked:= True;
         S.Owner:= RTK.Self;
      else
         Enqueue(S.Q);
         RTK_Inheritance.Create_Relationship(RTK.Self, S.Owner);
         RTK.Suspend_Self(RTK.Self, Suspend_ID);
      end if;
      RTK.Release_Lock(S.Lock);
   end Acquire;

   procedure Release(S: in out Semaphore) is
      Waiter: RTK.VP_ID;
      New_Q: VP_ID_Q_Type;
   begin
      RTK.Seize_Lock(S.Lock);
      if (Is_Empty(S.Q)) then
         S.Is_Locked:= FALSE;
         S.Owner:= RTK.Null_VP;
      else
         S.Owner:= Dequeue(S.Q);
         RTK_Inheritance.Destroy_Relationship(S.Owner, RTK.Self);
         while (not Is_Empty(S.Q)) loop
            Waiter:= Dequeue(S.Q);
            Enqueue(New_Q, Waiter);
            RTK_Inheritance.Destroy_Relationship(Waiter, RTK.Self);
```

```
            RTK_Inheritance.Create_Relationship(Waiter, S.Owner);
         end loop;
         S.Q:= New_Q;
         RTK.Resume(New_Owner, Suspend_ID);
      end if;
      RTK.Release_Lock(S.Lock);
   end Release;

end Semaphores;
```

Another detail that must be decided, if support is provided for general priority inheritance, is whether to provide one-to-many inheritance, such as from a task to all of the children for whose activation it is waiting, or from a completing master to all the dependents for whose termination it is waiting. This must be weighed carefully, because it would complicate the implementation of the entire RTK, and is certain to add to the runtime overhead.

5.10.3 No Timer Support

Operations involving time, such as a clock function, time-delay, or time-out facility, are intentionally not included in the RTK. Logically, they would be at about the same level as a queued entry facility, which is not included either; that is, one level above the RTK. We believe a time-out facility could be built using a timer interrupt handler, a lock to protect a time-ordered queue, and the `Resume_VP` operation.

5.10.4 No Memory Management

Support for memory management (e.g. virtual-to-physical address mapping) and dynamic storage allocation has intentionally been left out. This is partly because of a desire to keep the RTK simple. Memory management was also left out because the function is very dependent on the specific machine architecture, and on the way in which the hardware is used by the system. It was also felt that memory management can be added by the user, or the developer of the compiler or RTS, without affecting the RTK interface defined here. For example, an address look-aside-buffer trap could be handled by a trap handler that does not need to interact directly with the RTK implementation.

The memory allocation function was also believed to be implementable independently from the RTK. In fact, it appears that the locking features of the RTK could be used to construct a safe memory allocator.

6 Conclusion and Future Work

The RTK has already served a useful role by providing a framework for discussions that led to working out some of the gaps and conflicts in CIFO 3.0. This version of the RTK is being submitted for publication in order to provide background for CIFO 3.0, and because it may be useful to people working on Ada runtime systems. The authors are interested in feedback from Ada users and implementors. Any such feedback will be taken into account when the ARTEWG takes up the RTK again.

The relationship of the RTK and Ada9X still needs to be addressed. The locking semantics of the RTK was designed anticipating some of the revisions that have been proposed to the Ada tasking model for Ada 9X, particularly Protected Records[2]. When the Ada 9X standard has firmed up, it

will be interesting to see how much modification will be required to the RTK to support an Ada 9X RTS implementation.

An effort should be made to see if various priority inheritance protocols can be implemented directly using the RTK services. In particular it should be studied whether our notion of base and active priority is sufficient to describe the rules and semantics of a system under priority inheritance. We do not believe that the RTK needs to support the actual policies of PI, but instead it may be enough to provide a small number of primitives, such as for keeping track of ownership of resources and creating dependency graphs, to facilitate the implementation of more elaborate mechanisms.

As is the case with every interface specification, it is very beneficial to develop prototype implementations. This serves to check some of the assumptions behind the interface, and the various trade-offs made. Of particular interest is the question of efficiency; this can only be checked by an implementation. Also, in order to verify the claims that the proposed set of primitives is sufficient to build higher-level components, like the CIFO, some examples of implementations of such components using the RTK would be beneficial. Candidates include: task restart; asynchronous exceptions; asynchronous transfer of control; time slicing; suspending another VP; blocking semaphores; priority inheritance policies (including one-to-many inheritance); preemption control.

A family of prototype implementations of the RTK has been produced by students at the Florida State University for the Motorola 68020 processor[12]. One version runs on a Sun workstation, using the Sun UNIX[1] operating system, and another runs on a "bare" 68020 board. These implementations have been tested enough to know that they work, and have helped iron out some details in the interface definition. A multiprocessor implementation has also be built, as a layer over the single processor implementation. This runs on four MVME133a computers connected by a VME-bus.

More prototyping is needed. In particular, the implementations mentioned above were tested using the non-tasking support of a vendor-supplied Ada runtime systems, by simply ignoring the tasking support of that RTS. This is a limitation. To verify the principle of using the RTK to add-on XRTL components, it will be necessary to test the RTK *under* an Ada RTS. This will require construction of a full Ada RTS over the RTK, which is compatible with the interface of an existing Ada compiler.

7 Acknowledgements

We are indebted to the ARTEWG principals for reviewing an earlier draft of the RTK specification, and especially the members of the RTK "Tiger Team": Scott Carter, Bruce Jones, and Tom Quiggle. We also thank Pratit Santiprabhob and Chi-Sing Chen for their comments and criticisms of earlier drafts, and their efforts producing an implementation of the RTK.

References

[1] U.S. Department of Defense, *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, Ada Joint Program Office (January 1983).

[2] Ada 9X Project Office, "Draft Mapping Document", Ada 9X Project Report, Office of the Under Secretary of Defense for Acquisition (February 1991).

[3] ARTEWG (i.e. Ada Run Time Environment Working Group of ACM SIGAda), "A Catalog of Interface Features and Options for the Ada Run Time Environment", Release 2.0, ACM SIGAda (1988).

[1] Unix is a trademark of AT&T.

[4] Ada Run Time Environment Working Group(ACM SIGAda), "A Framework for Describing Ada Runtime Environments", ACM SIGAda (15 October 1987).

[5] T.P. Baker, "Comment on: 'Signaling from within Interrupt Handlers"", *Ada Letters XI,1* (January/February 1991) 17-18.

[6] M.J. Carolus, "ExTRA-CIFO Comparison". Aerospatiale Report 468.026/90 01 Raf-Di-Pro (1990).

[7] ExTRA Working Group, Draft Package Specifications (1989).

[8] J.B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks", Proceedings of the Second International Workshop on Real Time Ada Issues, Moreton-Hampstead, Devon, England (June 1988); published in *Ada Letters* VIII, 7 (Fall 1988).

[9] B. Jones, personal communication.

[10] IEEE TCOS POSIX Working Group, *Threads Extension for Portable Operating Systems*, Draft 5, IEEE (December 1990).

[11] S. Kleiman, "Synchronization in Asynchronous Signal Handling Environments", IEEE POSIX working paper P1003.4-N0299 (May 1991).

[12] Pratit Santiprabhob, Chi-Sing Chen, and T.P. Baker, "Ada Rum-Time Kernel: the implementation", in *1st Software Engineering Research Forum*, Tampa, FL (8-9 November 1991) 89-98.

[13] Chi-Sing Chen, "An Implementation of RTK on a Multiprocessor Machine", MS project report, Dept. of Computer Science, Florida State University, Tallahassee, FL (January 1992).