Ada Run-Time Kernel: The Implementation*

Pratit Santiprabhob (pratit@cs.fsu.edu)
Chi-Sing Chen (cschen@cs.fsu.edu)
T.P. Baker (baker@cs.fsu.edu)

Department of Computer Science B-173
The Florida State University
Tallahassee, FL 32306-4019

October 13, 1991

Abstract

In this paper we describe prototype implementations of a unified priority based runtime kernel (RTK) for Ada. The kernel has a novel feature, namely the unified treatment of priorities. The same priorities govern interruptibility, resource locking, and processor scheduling, so that priority inversion is minimized. The main goal behind the development of the RTK is to accommodate independently developed runtime system (RTS) components which share machine resources and scheduling policies. The RTK has been implemented on both single processor and multiprocessor architectures. The implementation, as well as problems and limitations encountered during the course of the implementation, is the main topic being discussed here.

Keywords: real-time system, Ada, kernel, run-time system, virtual processor, thread of control, tasking.

1 Introduction

The Ada Run-Time Kernel (RTK) is a runtime system kernel for real-time usage proposed by the ACM Ada Runtime Environment Working Group (ARTEWG). The RTK provides a unifying execution model that can be used to explain the interaction of the standard Ada [7] tasking Run-Time System (RTS) with extended runtime library (XRTL) components. The execution model can also be used as a tool to describe both semantics and interactions of the higher level primitives such as rendezvous, semaphores, etc. Priorities, which govern interruptibility, resource locking, and processor scheduling, are treated uniformly by the RTK. The primary intended users of the RTK are developers of Ada RTS's and RTS extensions. It is envisaged that if the RTK is adopted, a newly developed RTS component can co-exist with a standard Ada RTS without the need of source level integration and testing. This would facilitate and encourage the development of high quality products for the Ada realtime market. The interface to the RTK is given as an Ada package. Details concerning the package specification, as well as other details of the RTK can be found in the paper of Baker and Pazy [4].

The authors have implemented the RTK on both a single processor and multiprocessor machines. The purposes of the implementations are to test the viability and efficiency of the RTK and to gather feedback for the improvement of the kernel's specifications. The prototypes have in fact already contributed to some improvements in the RTK specification, which eventually lead to the current version in [4].

In Section 2, we briefly describe the execution model of the RTK. The general structure of the implementation is described next in Section 3. Major design problems and limitations encountered during the implementation of the prototypes are discussed in Section 4. Details specific to the multiprocessor implementation are given in Section 5. A listing of the RTK package specification of our UNIX[1] implementation is included in Appendix A.

*This work is supported in part by the State of Florida High Technology and Industry Council Applied Research Grants Program. This paper has been submitted to the First Software Engineering Research Forum, Tampa, Florida.

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

2 Execution Model

The execution model of the RTK is based on the concept of *Virtual Processor* (VP). A VP is a construct designed to encapsulate a software component that can execute in parallel with other similar components. In the RTK, besides the VP there are the concepts of *Interrupt* and *Lock*.

Two types of interrupt are supported: system interrupts (hardware interrupts and software traps) and VP interrupts. While a system interrupt interrupts an actual physical processor, a VP interrupt, which is originated by a VP, is a virtual interrupt targeted to one specific VP. Since a hardware interrupt is managed by the underlying hardware interrupt mechanism it may interfere with the execution of the RTK code; since a VP interrupt is managed by the RTK it may not interrupt the RTK. Each interrupt has either a user-defined or a default *Interrupt Handler* (IH) attached to it. An interrupt handler for a system interrupt is global. On the other hand, one for a VP interrupt is local to a VP to which the VP interrupt is targeted.

A lock is an abstract of a non-preemptable shared resource. When a VP wants to use a resource, it seizes an associated lock before it starts using the resource, and releases the lock after it finishes using the resource. While a lock is being held by a VP, the RTK ensures that no other executing VPs or interrupt handlers will seize the lock. In the RTK's unified priority model, priorities are assigned to all VPs, IHs, and locks. These priorities are used in determining interruptibility, locking, and processor scheduling. Every time a VP is created, a lock is initialized, or an IH is attached, a user is responsible for assigning an appropriate priority to the VP, lock, or IH, respectively. Note that VPs can be destroyed, locks can be finalized, and IHs can be detached, at any time by a user.

A VP corresponds to what the Ada reference manual [7] refers to as a *logical processor*. As the name implies, a VP in many ways mimics operations and features of a physical processor. The context of a VP includes: an identification, a hardware context (including the program counter and other registers), a dispatching state, a VP interrupt state and handler information, and two priorities (base and active). The base priority is a priority initially assigned to a VP when it is created; the priority can later be changed, by the `Set_Base_Priority` operation. The active priority is a priority at which a VP is currently executing. It is the maximum of the base priority of the VP, and the priorities of the set of locks currently held by the VP and the IHs currently being executed by the VP. The dispatching, i.e. the allocation of a physical processor to a VP, is done according to the VP's active priority. A VP, once dispatched, is allowed to continue executing until it suspends itself or is preempted by a VP with higher active priority. There are a number of operations on VPs provided by the RTK. The operations are described in detail in [4].

A priority assigned to a lock must be the maximum of the active priorities of all the VPs and IHs that will ever attempt to seize the lock. When a VP (or an IH) has successfully seized a lock the active priority is raised to the lock's priority (if it is not already that high). It is an error for a VP (or an IH) to attempt to seize a lock that has a priority lower than its active priority. If the optional checking of errors is to be performed by the implementation of the RTK, the RTK is required to raise an exception when this error is detected. This locking mechanism, which is essentially the same as that proposed for Ada 9X protected records [1], guarantees within a single processor: mutual exclusion, no priority inversion[2], and consequently no deadlock. An implementation of the RTK on a multiprocessor architecture has to come up with extra mechanisms to guarantee these properties. A scheme implemented for a particular multiprocessor architecture, which is based on a notion of *global lock*, is explained in Section 5.

The unified priority model also implies that all interrupts having priorities lower than the current active priority of a VP (or an IH) are disabled with respect to the execution of the VP (or the IH) during the period in which the active priority is maintained.

Even though the unified priority model of the RTK has somewhat complicated the implementation, it provides a rich semantic model. For example, it solves the problem of reliably using a hardware interrupt to signal an event to a waiting task. This problem is discussed for Ada in [3], and illustrated for POSIX 1003.4a thread primitives by Kleiman in [5].

The problem can be described abstractly as follows. A thread at some point during its execution will check a flag to see if a particular event has occurred. If the event has occurred the thread will proceed with its execution. Otherwise it will go to sleep waiting for the event to occur. Later, when the event occurs, a signal handler is invoked to set the flag and wake up the thread (if it is sleeping). Note that there is a gap be-

[2] A priority inversion occurs when a job with lower priority blocks an execution of another job whose priority is higher. See [6] for more details.

tween the checking of the flag and the time the thread actually goes to sleep. If the event occurs right in that gap, a signal handler will set the flag after the thread has checked it, and will try to wake up the thread before the thread has gone to sleep. Consequently, the thread will not notice that the event has occurred, and miss the event.

Making a thread a VP, and a signal handler an IH, we can use operations provide by the RTK to solve this race condition as shown in Figure 1.

```
Dummy (a VP):
  Seize_Lock(L);
    if Event_Occur = True
    then Event_Occur:= False;
    else Suspend_Self;
    end if;
  Release_Lock(L);

Foo (an IH):
  Seize_Lock(L);
     Event_Occur:= True;
     Resume_VP(Dummy);
  Release_Lock(L);
```

Figure 1: Solving race condition using RTK.

According to the RTK specification, the effect of `Suspend_Self` is deferred, i.e. a VP is allowed to continue its execution until all locks held by the VP are released, if there are some locks being held by the VP at the time it calls `Suspend_Self`. In addition, `Resume_VP` has no effect if the VP is not suspended. In our code here, once `Dummy` has successfully seized the lock `L` the exeuction of `Foo` is blocked until `L` is released. If the event has not yet occurred at that point `Dummy` prepares itself to go to sleep by calling `Suspend_Self` but this will not actually suspend `Dummy` until `L` is released. Otherwise, `Dummy` will reset the flag `Event_Occur` and proceed on its execution. On the other hand, when an event occurs and `Foo` has successfully seized the lock `L`, `Dummy` is blocked from going to sleep until `L` is released. This ensures that either `Dummy` successfully go to sleep before `Foo` can ever send a signal to it, or the flag `Event_Occur` is set by `Foo` before `Dummy` can prepare itself to sleep. Hence, the race condition has been eliminated.

3 Implementation Specifics

A single processor version of the RTK has been implemented by the authors on two different Motorola 68020 based machines. One has been implemented on the top of SunOS[3] (UNIX) on a Sun 3/60 workstation, while the other has been implemented on an MVME 133a machine without any operating system. A multiprocessor version has also been implemented on multiple MVME 133a machines connected together by a VME bus. Each MVME 133a has its own local memory, that can also be accessed remotely by other machines on the same bus. Figure 2 shows schematic diagrams of the single processor implementations. The multiprocessor implementation is depicted by Figure 3.

For the single processor case, we implemented the RTK on the top of either SunOS or the standard Verdix runtime system. The current version of the multiprocessor implementation, which is called MRTK, is built on the top of the single processor RTK. The MRTK has an interface similar to that of the RTK. An application program should be able to use the interface provided by MRTK to perform RTK operations within as well as across the processors. Between MRTK and RTK, there is a layer called Cross. This Cross layer provides a conceptual link for MRTKs of different processors. More details about the Cross layer are provided in Section 5. According to the current set up, we have one dispatcher per machine, which takes care of only VPs created on that machine.

The code for the implementations is written in the Ada language with machine code insertions for some low-level routines. The code for the UNIX implementation is compiled by the Verdix self-targeted compiler, Version 6.0, while the code for the MVME 133a is complied by the Verdix cross compiler, Version 5.7.

There are two main packages in the single processor implementation. The first one is the `RTK` package, which contains the code for operations specified by the RTK specification, e.g. locking, scheduling, operations on Virtual Processors (VPs), etc. The other one is the `Machine` package which consists of low-level routines that interface with the underlying operating system and/or hardware. `Machine` includes routines for manipulating runtime stacks of VPs and VP interrupt handlers, attaching and detaching system interrupt handlers, and disabling and enabling the system interrupts. For the multiprocessor version, we have two additional packages, the `Cross` package and the `MRTK` package. `Cross` is implemented on the top `RTK`

[3] SunOS is a trademark of Sun Microsystems.

Figure 2: Schematic diagrams of the single processor implementations.

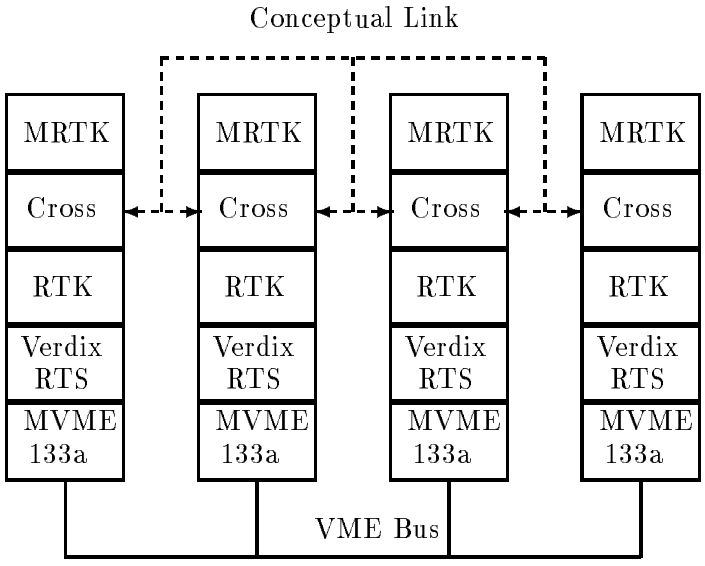

Figure 3: Schematic diagrams of the multiprocessor implementation.

and `Machine`. `Cross` serves two purposes. First, it provides support for the global locking mechanism across the multiple machines. Second, it provides a communication mechanism for MRTKs running on different machines, by which an application program running on one machine can remotely request that an operation such as `Interrupt_VP` or `Resume_VP` be applied to a VP on a different machine. `MRTK`, which is a higher-level interface, is implemented on the top of `Cross` and `RTK`. `MRTK` provides a uniform interface for operations both within and across the processors. The Ada package dependencies of our implementations are shown in Fig 4.

In the implementations, we reserve some of the highest priorities for the system interrupts. How many depends on the number of levels of system interrupt that the underlying operating system and/or hardware supports. For UNIX, we support the handling of UNIX signals. A user can provide handlers for all 31 signals except `SIGKILL` (9) and `SIGSTOP` (17) which cannot be caught, blocked, or ignored as defined by the UNIX. All signals have the same priority; therefore we reserve the highest priority for the signals. In the MVME implementation we support the handling of hardware interrupts. We have 256 interrupts ranging from vector number 0 to 255. Each interrupt falls in one of the seven levels. We therefore reserve the seven highest priorities for these interrupts.

4 Design Problems and Limitations

During the course of the implementations we have encountered some design problems and some limitations imposed by the available software and hardware. Some of the problems are situations not explicitly covered by the RTK specification. Some others are requirements that we did not find simple and efficient ways to implement. Among those, we have selected the major ones to discuss below. These problems and limitations occur in both single processor and multiprocessor implementations.

For the Motorola 68020 (the MVME implementation), we have no way of determining the level of a hardware interrupt beforehand. In order to preserve the the actual hardware priority level of the hardware interrupt, we have decided that the base priority of any hardware interrupt handler is to be determined and assigned dynamically for each execution of the handler as the RTK receives the interrupt. Note that a particular interrupt can have different hardware priorities when generated by different devices. Thus, in the MVME implementation, the priority specified by a user when attaching a hardware interrupt handler does not actually have a meaning.

While our implementation supports the full range of priorities for locks, we restrict the range of base priorities for VPs and VP interrupt handlers to be lower than the range of priority reserved for the system interrupts. This has been done solely for efficiency. Otherwise, supporting the full range of base priorities for VPs and VP interrupt handlers would incur more overhead. The overhead is caused by an extra checking, for whether a VP (or a VP interrupt handler) should be executed with system interrupts disabled (and to which level) due solely to its base priority when it is not holding any locks. This extra checking needs to be performed every time a VP (or a VP interrupt handler) is dispatched. On the other hand, in our adopted scheme the checking is confined to the locking operations. Note that system interrupts can still be disabled during the execution of a VP or a VP interrupt handler by simply seizing a lock with high enough priority.

The RTK specification presumes that the RTK either is a layer beneath the Ada RTS or is integrated with the RTS. As such, the RTS start-up is part of the booting of the Ada RTS. For our prototype we had to use a commercial Ada compiler for which we did not have RTS source code. This required that we implement the RTK in a form that looks like a normal Ada compilation, and design the RTK implementation so that it can coexist with the compiler's RTS. Specifically, there must be an Ada procedure compiled and linked as a main program to start up the RTS execution. The problem is how to treat that main procedure. We decided to treat the main procedure as a special VP called `Main_VP`. The data structures for this `Main_VP` are initialized automatically in the body of the `RTK` package. Note, however, that `Main_VP` is not actually a VP, and when this main procedure exits, the whole execution shuts down. We therefore assign the reserved lowest priority to the `Main_VP` to guarantee that all other VPs created and resumed by the `Main_VP` will have a chance to execute before the `Main_VP` exits. A user can obtain an ID of the `Main_VP` by calling `Self`. Using the ID, a user can perform any RTK operation on the `Main_VP`. In the case that the `Main_VP` is destroyed, the whole execution will shut down only when there are no more VPs waiting to be dispatched and no hardware interrupt handlers attached.

Since VP interrupt handlers share the same runtime stack with the VP to which they belong, a VP interrupt

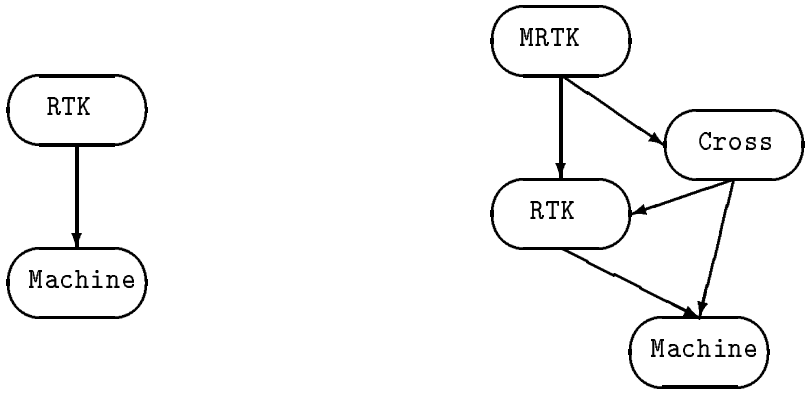

Figure 4: Ada package dependencies.

handler may block the execution of a preempted VP whose priority is raised (by the `Set_Base_Priority` operation) to be higher than that of the VP interrupt handler. This blocking occurs when the handler (whose priority was originally higher than that of the VP) is currently either executing or preempted by something else. In this case an activation record of the handler will be sitting above that of the VP on the stack, preventing the VP from resuming its execution. This can lead to a system deadlock or the runtime stack being corrupted. To solve the problem, we employ temporary priority inheritance for all VP interrupt handlers that are blocking the VP. The handlers will then assume the VP's new priority, and have a chance to execute to the end one by one in the LIFO order of the stack space they occupied. Eventually, the blocking runtime stack space will all be released, and the VP can resume its normal execution. We unavoidably introduce temporary priority inversion here in order to prevent the deadlock.

The `Set_Base_Priority` operation, which allows a user to dynamically change a base priority of an existing VP, is expensive, but unavoidable if such dynamic priority changing is essential to the application. We have decided to make the operation totally uninterruptible; therefore we risk missing some important system interrupts. However, if we let the operation be interruptible and there is an interrupt whose handler operates on the VP whose base priority is being changed, the result of the interleaved execution will be indeterminate.

The last major problem we encountered was with the Ada exception propagation mechanism. As stated in the RTK specification, an exception raised in an interrupt handler should be propagated back to the interrupted VP. To do this, we need to catch the exception in the handler and reraise it in the interrupted VP without having the exception propagated through the RTK code. Since we have no access to the source code of the Verdix compiler's runtime system, we have not found a way to provide the mechanism.

5 Multiprocessor Case

5.1 Overview

As stated before, we implemented the multiprocessor version of the RTK on top of the single processor MVME version. VPs, interrupt handlers and ordinary locks are maintained locally by the RTK running on each processor. Main objectives of the current multiprocessor implementation are to provide an interprocessor locking mechanism that respects VP priorities, and an interprocessor communication mechanism by which an operation such as `Interrupt_VP` can be performed remotely on a VP that resides on a different processor. Both of the mechanisms are implemented in an Ada package called `Cross`. On top of `Cross`, we have another Ada package named `MRTK`, which provides a uniform interface that hides the distribution. The two additional packages and their dependencies are depicted in Figure 4.

In order to implement an interprocessor locking mechanism, we create a set of locks called *global* locks that

all VPs on all processors are allowed to seize. These global locks are used to protect resources shared by all processors. When a VP wants to access such a resource it has to seize the associated global lock. While the interprocessor communication mechanism allows a VP to request an operation be applied to another VP that resides on a different processor, a VP is precluded from seizing a non-global lock on a different processor. This allows us to implement non-global locks in a more efficient manner.

5.2 Hardware and Limitations

Multiple MVME133a machines are connected through a VME bus. Each MVME133a machine has on-board *local* memory. The machines can access their own local memory without recourse to the VME bus, and they can remotely access others' local memory via the VME bus. There is also *shared* memory, which can be accessed by all of machines on the bus.

It is clear that both of mechanisms we want to implement need to use the VME bus. The architecture of the hardware available to us imposes the following limitations:

- *slow bus speed*
  A processor that wants to access data remotely through the VME bus must obtain bus mastership before it can execute a VME bus cycle. Therefore the access time for remote memory is much longer than for local memory.
- *fixed bus priority*
  The VME bus arbitrates between processors according to a fixed priority scheme, based on the physical position of the machines. We will call this fixed bus priority of each processor its *privilege*, to distinguish it from the active priority of the processor, which is that of the VP or interrupt handler that happens to be executing.
- *broadcast interrupt*
  From our understanding of the hardware available to us, the only way for one processor to interrupt another processor is to issue a VME bus interrupt, that is broadcast to all the processors on the bus.

5.3 Interprocessor Locking

Since an interprocessor locking mechanism must access globally shared data structures through the VME bus, the accesses are expensive and must be kept as infrequent as possible. There are three problems, namely bus contention, enforcing VP priorities, and avoiding race conditions, to be addressed in the design of an interprocessor locking mechanism that respects VP priorities.

To solve the first problem, the bus contention, it is necessary to avoid busy-waiting on globally shared variables. This is done by having a processor that fails in an attempt to grab a shared variable (using an atomic instruction such as test-and-set or compare-and-swap) busy-wait on a local variable instead of the shared variable. A processor that is busy-waiting on a local variable will be awakened by the processor that releases the shared variable via a remote write operation into the local variable on which it is busy-waiting. This relieves the contention for the bus and allows processor with lower privileges to have a chance to access global variables through the VME bus without starvation.

Since we have decided to make the processor waiting for an already-seized global lock busy-wait on its local variable, it is easy for us to enforce the VP priorities in granting of the global lock. When the global lock is to be released, we only need to scan through a list of processors waiting for the lock, and grant the lock to the processor that is executing the highest priority VP.

Now, there can be a race condition during the scanning of the list of waiting processors. Note that this scanning must be done before we decide to either free the global lock, if we find that no one is waiting for the lock, or to grant it to the highest priority processor waiting for it. The race condition occurs when a new processor tries to seize the global lock after the scanning has already started but before it has finished. The newcomer could be missed. The newcomer should already be, or will be, busy-waiting on its local variable. So, if the newcomer has the highest priority, it will be a priority inversion. On the other hand, if the scanning does not find anyone else waiting for the lock and decides to release the lock, the newcomer will continue spinning on its local variable until someone else seizes the lock and later releases it. At worst, this could lead to deadlock. To prevent these undesired situations from occurring, we have a flag which is reset at the beginning of each scan. Every time a processor tries to seize the global lock, it sets the flag. Therefore, all we need to do is to check at the end of each scan to see whether the flag has been set by any newcomers. If it has been set we need to rescan the list of waiting processors.

5.4 Interprocessor Communication

A remote procedure call model is used for the interprocessor communication. Each VP has a block of space in shared memory, called a *communication block*, which stores the information related to the services the VP requests from another processor. This space is declared and allocated in the package `Cross`. Whenever a VP wants to request some RTK operation be performed on another processor it writes the request into its communication block and enqueues the request on a per-processor queue in shared memory. The queue is ordered according to the priority of the VP making the request. After the request has been enqueued, the requesting processor will busy-wait for the response. We provide two ways that a request can be detected by the target processor. Normally each processor will periodically poll its own queue for requests. In case of a more urgent request, the index of the target processor will be written into a shared space and an interrupt will be broadcasted over the VME bus. When the interrupt is received, every processor will check the shared space. Only the processor which is targeted will serve the request. Meanwhile, the other processors will resume their work. The broadcasting of an interrupt should only be used for an extremely urgent request, since it will interrupt all the other processors on the VME bus. Either way, when the target processor gets the request and has acted upon it, a response is sent to the processor from which the request originated. Only when the response is received is the requesting processor freed to continue with its own work.

6 Conclusion

This paper describes the design of prototype implementations of the RTK on both single processor and multiprocessor architectures. Problems encountered during the course of the implementations are also discussed. The current version of multiprocessor implementation is built on the top of the single processor RTK.

The research is still going on. We are currently working on improvements to our prototypes. The full evaluation of the implemented RTK, including performance testing, is also underway, for both single processor and multiprocessor cases. The evaluation could result in further recommendations for both the implementations and the specification of the RTK. We also hope to have a chance to implement the RTK on different architectures, as well as to use an Ada compiler that allows us to install the RTK under its RTS, in the future.

References

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

[2] T.P. Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes", *Proceedings of the IEEE Real-Time Systems Symposium*, (December 1990).

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

[4] T.P. Baker and Offer Pazy, " A unified priority-based Kernel for Ada", submitted to ACM SIGAda *Ada Letters*.

[5] S. Kleiman, "Synchronization in asynchronous signal handling environment", IEEE POSIX working paper P1003.4-N0299, IEEE (May 23, 1991).

[6] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization", Technical Report, CMU-CS-87-181, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 (November 13, 1987).

[7] U.S. Department of Defense, *Military standard Ada programming language*, ANSI/MIL-STD-1815A, Ada Joint Program Office (January 1983).

A RTK package specification

```
with System;
with Util;
package RTK is

  -- Exceptions and Error Checking
  Optional_Checks_Performed: constant Boolean:= False;
  RTK_Error:     exception;
  Locking_Error: exception;

  -- VP ID's
  Max_VP_IDs: constant:= 1_000_000;
  type VP_ID is private;
  Null_VP: constant VP_ID;
  function Is_Valid(VP: VP_ID) return Boolean;
  function Self return VP_ID;

  -- Creation and Destruction
  type Init_State is
    record
      Stack_Addr: System.Address;
      Stack_Size: Natural;
    end record;
  subtype VP_Priority is Integer range 0..99-1;
  -- the highest priority is reserved for UNIX signals.
  Max_VPs: constant:= 20;
  procedure Create_VP
    (Priority: VP_Priority;
     Initial_State: Init_State;
     Entry_Point: System.Address;
     VP: out VP_ID);
  procedure Destroy_VP(VP: in out VP_ID);

  -- User-definable VP Attributes
  Null_Attribute: System.Address
                := Util.Null_Procedure'address;
  type Attribute_ID is range 1..10;
  procedure Set_Attribute
    (VP: VP_ID;
     Attribute: Attribute_ID;
     Value: System.Address);
  function Get_Attribute
    (VP: VP_ID;
     Attribute: Attribute_ID) return System.Address;

  -- Suspension and Resumption
  type Suspension_ID is range 1..10;
  procedure Suspend_Self(Suspension: Suspension_ID);
  procedure Resume_VP(
     VP: VP_ID;
     Suspension: Suspension_ID);

  -- Locks
  type Lock_ID is limited private;
  subtype Lock_Priority is Integer range 0..99;
  Max_Locks: constant:= 20;
  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;

   -- 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;

   -- Interrupts
   type Interrupt_ID is range 1..35;
   -- for UNIX signals number 1 to 31, and four VP_Interrupts.
   subtype VP_Interrupt_ID is Interrupt_ID range 32..35;
   subtype Interrupt_Priority is Integer range 0..99;
   type Interrupt_Info is range 0..0; -- not in use.
   type Handler_Info is range 0..0; -- not in use.
   function Current_Priority
     (Interrupt: Interrupt_ID) return Interrupt_Priority;
   procedure Interrupt_VP
     (VP: VP_ID;
      VP_Interrupt: VP_Interrupt_ID;
      Info: Interrupt_Info:= 0);
   procedure Attach_Interrupt_Handler
     (Interrupt: Interrupt_ID;
      Priority: Interrupt_Priority;
      Handler_Address: System.Address;
      Info: Handler_Info:= 0);
   procedure Detach_Interrupt_Handler
     (Interrupt: Interrupt_ID);

private
   ...

end RTK;
```

