D Real-Time Systems

The purpose of this Annex is to supplement the core language with features specific to real-time systems. Since Ada is a general-purpose language with a large and diverse audience, not all the capabilities that are required to build applications can be sensibly put in the core language without prohibitively increasing its size and hurting other application domains.

As is the case with Ada 95 in general, this Annex tries to provide a single solution to each recognized problem, even though we acknowledge that several alternatives exist in the market-place. The mechanisms that we have provided can serve as building blocks when more sophisticated solutions are needed. The models that we specify allow for extensions, and accommodate a certain degree of variability. The primary goal was to allow the user to rely on a portable, yet usable, set of capabilities. These capabilities will always be present in an implementation that supports this Annex. Therefore, the default behavior is well-specified, and the user must explicitly request implementation-provided additions. In addition, optionality within this Annex was kept to a minimum.

This annex addresses the following topics

- Priorities of tasks in general and especially the ability to change priorities (they were fixed in Ada 83) either explicitly or as a result of interaction with other tasks and protected objects;
- Scheduling issues including the entry queue discipline;
- Specific measurements on the effect of the abort statement including the formalization of the concept of an abort-deferred region;
- Restrictions on the use of certain aspects of the tasking model that should permit the use of specialized runtime systems;
- Resolution of a number of timing issues in Ada 83, including the introduction of a distinct monotonic clock;
- The addition of explicit synchronous and asynchronous task control protocols.

Note that several features in this Annex are invoked by using configuration pragmas (see [RM95 10.1.5]). This means that the corresponding semantics are defined on a per-partition basis; thus priority scheduling, the queuing policy and time are only specified in the scope of the one active partition containing the unit where the pragma applies. Ada 95 does not address issues concerning multipartition execution beyond the minimum that is in the Distributed Systems Annex. Interactions among partitions are left to implementations or to the providers of partition communication software.

D.1 Task Priorities

In real-time applications, it is necessary to schedule the use of processing resources to meet timing constraints. One approach is to use priority scheduling; this has a well developed theory, and is adequate for a wide range of real-time applications; see [Sha 90a] and [Klein 93].

Ada 83 specified preemptive task scheduling based on static priorities, but left certain aspects implementation dependent. This scheduling model, however, has raised issues in practice. On the one hand, it is viewed as not sufficiently specified for portable real-time system designs. On the other hand, it is viewed as too restrictive to permit the implementation of important real-time programming paradigms.

It is important that a language not limit the application to a particular scheduling model. There are many appropriate scheduling techniques, and more are continually being developed. No one scheduling model is accepted as adequate for all real-time applications.

It is also important to permit Ada to take advantage of the concurrent programming support of commercial real-time operating systems or executives. This is especially so with the growing acceptance of the "open systems" approach to software architecture, and the development of standards for application program interfaces to operating system services, such as POSIX [1003.1 90]. Ada should not impose any requirements on the language implementation that conflict with the scheduling model of an underlying operating system.

For these reasons, the Ada 83 priority scheduling model has been removed from the core of the language. However, this leaves a gap. Some users have found the Ada 83 scheduling model useful and it is clearly essential to continue to support those users. This argues for the inclusion of a priority scheduling model in this Annex, and for it to be compatible with Ada 83.

A second reason for specifying a standard scheduling model in this Annex is economy. Even though a single scheduling model cannot satisfy the requirements of all Ada users, it seems that a large number can be satisfied with priority scheduling, provided that the obvious adjustments to Ada 83 are made. This model thus provides a useful base for vendors and users alike.

The priority model specified in this Annex thus subsumes the Ada 83 model and provides several important improvements: support for dynamic priorities; solutions to the problem of priority inversion; and a unified model of the interactions of task priorities with protected objects and interrupts.

The specification of the priority model is spread over several clauses in [RM95 D.1-5]. Besides readability, the main reason for this organization is to permit the designation of options within the Annex. In particular, while the overall task dispatching model is essential, the standard policies for Task Dispatching, Priority Ceiling Locking, and Entry Queuing may optionally be replaced by other implementation defined alternatives.

D.1.1 Priority Subtypes

The range of possible task priorities is extended so that it can overlap with interrupt priorities as on some hardware architectures. We now have

```
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority
                range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
                range Priority'Last+1 .. Any_Priority'Last;
```

The subtype `Any_Priority` is introduced (rather than simply allowing `Priority` to include interrupt priorities) because existing Ada 83 programs may assume that `Priority'Last` is below interrupt priority. Moreover, since giving a task a priority that blocks interrupts is sufficiently dangerous that it should be very visible in the source code, the subtype `Interrupt_Priority` is introduced. The ranges of `Priority` and `Interrupt_Priority` do not overlap.

A minimum number of levels of priority is specified, in the interest of promoting portability of applications and to ensure that an implementation of this Annex actually supports priority scheduling in a useful form. Research in Rate Monotonic scheduling [Lehoczky 86] has shown that approximately 32 levels of priority is the minimum needed to ensure adequate schedulability in systems with 32 or more tasks. Moreover, it is desirable that where hardware provides support for priority scheduling, it should be possible to use such support. Certain hardware architectures are reported to support only 32 levels of priority, including interrupt priority levels. Therefore the combined number of priority levels is not required to be higher than 32. In order to permit the use of efficient bit-vector operations on 32-bit machines, where one bit may need to be reserved, the actual requirement is reduced to 31 of which one must be an interrupt priority.

As in Ada 83, priority subtypes need not be static, so an implementation that is layered over an operating system can query the underlying operating system at elaboration-time to find out how many priority levels are supported.

D.1.2 Base and Active Priorities

The distinction between base and active priority is introduced in order to explain the effect of priority inheritance. The base priority of a task is the priority the task would have in the absence of priority inheritance of any sort. Priority inheritance is already present in Ada 83, during rendezvous. It is extended here, to bound priority inversion (see D.3.1 for the definition of priority inversion) during protected operations.

In the default scheduling policy, priority inheritance is limited to a few simple forms, in order to permit more efficient implementations. These forms do not cause the active priority of a task to change asynchronously. Inheritance happens only as a direct result of the execution of the affected task, when the task is being resumed, or before the task has ever executed. If inheritance is via protected operations, the priority is raised at the start of the operation and lowered at the end. If inheritance is via rendezvous, the priority is raised at the beginning of rendezvous (either by the accepting task itself, or by the caller before the acceptor is resumed) and then lowered at the end of the rendezvous (by the acceptor). The case of activation is slightly different, since if the active priority of the task is raised, it is raised by the *creator*. However, this change is synchronous for the affected task, since the task has not yet started to execute; the lowering of the priority is done at the end of activation by the action of the activated task.

Priority inheritance via queued entry calls, via abortion, and via a task master waiting for dependents to terminate is intentionally not specified, mainly because the effects are asynchronous with respect to the affected task, which would make implementation significantly more difficult. An additional reason for not specifying inheritance through task masters waiting for dependents is that it would be a one-to-many relation, which would also introduce extra implementation difficulty. Other reasons for not doing inheritance via abortion are stated in D.6.

D.1.3 Base Priority Specification

The initial specification of the base priority of a task is by means of the pragma `Priority`. This is compatible with Ada 83.

The pragma `Interrupt_Priority` is provided for specifying a base priority that may be at an interrupt level. The pragma is different in order to make it very visible in the source code wherever a base priority is being assigned that might have the side-effect of blocking interrupts. The `Interrupt_Priority` pragma is also allowed to specify priorities below interrupt level, so that it is possible to write reusable code modules containing priority specifications, where the actual priority is a parameter.

The rule that the priority expression is evaluated for each task object, at the time of task initialization satisfies the requirement for having task objects of the same type but with different priorities. The expression specifying the priority is evaluated separately for each task. This means that it is possible, for example, to define an array of tasks of different priorities, by specifying the priority as a discriminant of the task, or by a call to a function that steps through the desired sequence of priority values thus

```
task type T is
   pragma Priority(Next_One);    -- call function Next_One
...
```

and similarly for protected objects.

A default base priority is specified, so that the behavior of applications is more predictable across implementations that conform to this Annex. This does not prevent implementations from supporting priority inheritance or other implementation-defined scheduling policies, which relied for legality under Ada 83 on the task priority being unspecified. This is because an implementation need not support this Annex at all but if it does then it may still conform and provide user-selectable task scheduling policies that define additional forms of priority inheritance. Such inheritance may raise the active priority of a task above its base priority, according to any policy the implementation chooses.

The main reason for choosing the default priority of a task to be the base priority of the task that activates it (the base priority of its creator) is that the creator must wait for the new task to complete activation. For the same reason, AI-00288 specifies that during this time the task being activated should inherit the priority of the creator.

The default base priority of the environment task (`System.Default_Priority`) is chosen to be the midpoint of the priority range so that an application has equal freedom to specify tasks with priorities higher and lower than that of the default. It does not seem to always be the case that "normal" tasks (i.e. those that do not have a particular priority requirement), necessarily have the lowest priority in all circumstances.

D.2 Priority Scheduling

The purpose of this section is to define the operational semantics of task priority, and to define a specific default scheduling policy. The definitions introduced here are also used for priority ceiling locking [RM95 D.3] and entry queuing policies [RM95 D.4].

D.2.1 The Task Dispatching Model

Ada 95 provides a framework for a family of possible task dispatching policies, including the default policy which is specified in [RM95 D.2.2] as well as other policies which may be defined by an implementation.

The phrase *task dispatching* is used here to denote the action of choosing a task to execute on a processor at a particular instant, given that one already knows the set of tasks that are eligible for execution on that processor at that instant, and their priorities. This is distinguished from the more general concept of *task scheduling*, which includes determination of the other factors, i.e. which tasks are eligible to execute (in the logical sense), which tasks are allowed to be executed on each processor, and what is the active priority of each task.

The term "processing resource", which was introduced in Ada 83, is developed further. Informally, a processing resource is anything that may be needed for the execution of a task, and whose lack can prevent a task from execution even though the task is eligible for execution according to the rules of the language.

Besides processors, the only specific processing resources that are specified by the Annex are the logical "locks" of protected objects — i.e. the rights to read or update specific protected objects. An important feature of the protected type model (explained more fully in D.3) is that protected objects can be implemented in a way that never requires an executing task to block itself in order to execute a protected subprogram call. As explained in D.3, it is a consequence of the priority-ceiling rules that, if there is only one processor, the highest priority task that is eligible for execution will never attempt to lock a protected object that is held by another task. Thus, based on single-processor systems alone, there would be no need to treat protected objects as processing resources. However, on a multiprocessor system, regardless of how protected types are implemented, a task may be forced to wait for access to a protected object. Thus, access to a protected object must be viewed as a processing resource. Even on a single-processor system, if the implementation chooses not to use priority-ceiling locking, a task may need to wait for access to a protected object. This might be the case, for example, if tasks are implemented using the services of an underlying operating system which does not support economical priority changes. (Note that this potential waiting is not formally considered to be "blocking" by the rules of the language.)

In some systems there may be other processing resources. A likely example is access to a page of virtual memory. This might require a task to wait for a page of real memory to be allocated, and the desired page of virtual memory to be read into it. I/O operations may require access to an I/O device that is in use by another task (or operating system process).

The use of conceptual ready queues in the specification of the task dispatching model is derived from POSIX 1003.4 (Realtime Extension) [1003.4 93] and 1003.4a (Threads Extension) [1003.4a 93] standards.

A separate queue for each processor is specified in the model, in order to allow models of multiprocessor scheduling in which certain tasks may be restricted to execute only on certain processors. If the implementation allows all tasks to run on any processor, then the conceptual ready queues of all processors will be identical. Since this is only a conceptual model, the implementation is free to implement the queues as a single physical queue in shared memory. The model thus accommodates a full range of task-to-processor assignment policies, including the extremes of a single task dispatching queue and a separate queue per processor.

To allow for multiprocessor implementations, it is implementation defined whether a task may hold the processor while waiting for access to a protected object. This allows the implementation to directly use a "spin-lock" mechanism, or to use a (higher-level) suspending lock mechanism such as might be provided by an underlying multiprocessor operating system.

Though it is not specified here, it is desirable for delay queues to be ordered by priority within sets of tasks with the same wake-up time. This can reduce priority inversion when several tasks wake up at once. Ideally, run-time system processing for wake-ups of lower priority tasks should also be postponed, while a high-priority task is executing. This behavior is allowed by the model, but it is not required, since the implementation cost may be high.

Though we hope that the default scheduling policy defined in [RM95 D.2.2] will be adequate for most real-time applications, it is inevitable that there will be a demand for implementation-defined variations. We will consider how several such policies can be accommodated within the framework.

Consider the Earliest-Deadline-First (EDF) scheduling technique. The EDF scheduling algorithm is known to be optimal for systems of independent tasks on a single processor. The EDF priority of a task is the number of ready tasks with later (absolute) deadlines. In general, this value may need to be adjusted for every change in the set of tasks that are eligible for execution. Since there is no mechanism by which a user-defined scheduler can be notified to make such changes, the `Dynamic_Priorities` package (see D.5) is insufficient for a user to implement EDF scheduling. However, an implementation is free to provide EDF scheduling via an implementation-defined mechanism. The implementation could dynamically adjust base priorities to reflect EDF task ordering, in which case the semantics could be defined in terms of the run-time system calling `Set_Priority` to affect the changes. Alternatively, an implementation could model EDF scheduling by means of "priority inheritance", where tasks inherit priority dynamically from some implementation-defined abstraction. For this to work well, the base priorities of all tasks would need to be set to `Any_Priority'First`, since the active priority would need to be lowered dynamically, as well as raised.

Another anticipated application requirement is for time slicing. Implementation-defined time-slicing schemes may conform to this specification by modifying the active or base priority of a task, in a fashion similar to that outlined for EDF scheduling.

D.2.2 The Standard Task Dispatching Policy

The standard dispatching policy can be explicitly requested by writing

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

for the partition. An implementation may provide alternatives but none are required. If no such pragma appears then the policy is implementation defined.

As mentioned above, the purpose of specifying a standard task dispatching policy is to achieve more predictable task scheduling and more portability of applications that use priority scheduling, as compared to the Ada 83 task scheduling model. This leads to a dilemma. On one hand, the ideal is to completely specify which task will be chosen to execute. On the other hand, such specification will prevent (efficient) implementation on certain machines. In particular, there are inherent differences between multiprocessor and single-processor machines, and there may be constraints on task dispatching policies imposed by underlying operating systems. It seems there is no one task dispatching policy that will be acceptable to all users and implementable for all execution environments. Nevertheless, if there is a dispatching policy that will satisfy the needs of a large segment of real-time applications and is implementable on most execution platforms, there are benefits to making it always available.

While implementations are allowed to provide additional dispatching policies, there is no requirement that more than one such policy will be supported in the same active partition. This is based on the assumption that usually it does not make a lot of sense to talk about two independent dispatching policies in the same partition. Interactions must be defined and by doing so the two policies become essentially one. However, the support of two such unrelated policies is not precluded whenever it makes sense for the application and/or the underlying system. In addition, the dispatching policy is unspecified (as opposed to implementation-defined) if the user does not specify the pragma `Task_Dispatching_Policy`. This is because presumably, if the pragma is not provided, the user is not concerned about the dispatching specifics, and in addition, in many cases the actual policy (in the absence of the pragma) can simply be the policy of the underlying OS. This might not be specified, not documented precisely enough, or may even vary from one execution of the program to the next (as would be the case if the policy is controlled from outside the program).

The standard task dispatching policy specified in this Annex can be implemented on both single-processor and multiprocessor machines. It can be implemented by an Ada RTS that runs on a bare machine, and it is also likely to be implementable over some operating systems. In particular, the standard dispatching policy is intended to be compatible with the SCHED_FIFO policy of the Realtime Extension of the POSIX operating system interface standard.

A special feature of the delay statement, whether it appears as a simple statement or in a select statement, is that it always causes the executing task to go to the tail of its ready queue of its active priority. This is true even for delay statements with a zero or negative duration. It means that if there is another task of equal priority competing for the same processor, the task executing the delay will yield to this task. Imposing this rule makes the delay behavior uniform. It is also desired for predictable execution behavior, especially in situations where the delay duration or time is a value computed at run time, and which may have positive, zero, or negative values. As mentioned in UI-0044, causing a task to yield its processor to equal-priority tasks is a side-effect of delay statements in many existing Ada 83 implementations. Some current Ada users rely on this feature to achieve a form of voluntary round-robin scheduling of equal-priority tasks, under application control. Supporting this feature is expected to increase the execution time overhead of zero and negative delays, but the overhead does not seem to be greater than that which would be experienced if the shortest possible nontrivial delay (i.e. one that requires the task to be blocked) were executed.

D.3 Priority Ceiling Locking

Priority-ceiling locking of protected objects serves the following purposes, in order of decreasing importance

- Priority inversion can be bounded.
- A very efficient implementation of locking is permitted.
- Protected subprograms can be called safely from within direct-execution hardware interrupt handlers.
- On a single processor, deadlock is prevented.

Priority ceiling locking is specified by writing

```
pragma Locking_Policy(Ceiling_Locking);
```

in a unit of the partition. Other policies are permitted but not required. As with task dispatching, if no pragma appears for the locking policy, then the policy is implementation defined.

Note that if `FIFO_Within_Priorities` is requested as the task dispatching policy then `Ceiling_Locking` must also be specified.

D.3.1 Bounding Priority Inversion

By specifying that the task executing a protected operation inherits the priority of the protected object, we permit the duration of priority inversion (due to enforcement of mutual exclusion between operations on a protected object) to be bounded. A priority inversion is a deviation from the ideal model of preemptive priority scheduling; that is, a situation where a higher (base) priority task is waiting for a processing resource that is being used by a lower (base) priority task. Priority inversion is undesirable in a priority-based scheduling system, since it represents a failure to honor the intent of the user, as expressed by the task priorities.

Bounding priority inversion is important in schedulability analysis. In particular, if priority inversion can be bounded, Rate Monotonic Analysis can be used to predict whether a set of Ada tasks will be able to meet their deadlines [Sha 90a]. The technique has been successfully applied to several hard real-time systems written in Ada.

The ceiling locking scheme specified in this Annex is similar to the "priority ceiling emulation" in [Sha 90b], and to the "stack resource protocol" described in [Baker 91]. On a single processor, these schemes have the property that, once a task starts to run, it cannot suffer priority inversion until it blocks itself. Thus, the only points at which a task can suffer priority inversion are where the task has been unblocked (e.g. delay or rendezvous) and is waiting to resume execution. At these points, it may need to wait for one task with lower base priority (but a higher inherited priority) to complete the execution of a protected operation.

Among the locking policies that bound priority inversion, the policy specified here is the simplest to implement, and has been shown to be more or less indistinguishable from other policies in effectiveness. Support for this policy is also included in the mutex locking model of the proposed POSIX Threads Extension standard [1003.4a 93].

With priority inheritance through protected object ceilings, the duration of priority inversion encountered by a task `T` that has been unblocked will not be longer than the longest execution time of any one protected operation, over all the protected objects with ceilings higher than or equal to the base priority of `T`. In estimating this bound, the worst case execution time of each operation must be used, including the entry-servicing code. For a protected object with entries, this time bound must include the case where the maximum number of queued entry calls are served. (This number is bounded by the number of tasks that share access to the protected object.)

Checking of priority ceiling violations by the implementation can be helpful to the programmer, even if the implementation is not relying on the accuracy of this information for locking, since it amounts to verifying important assumptions that are made in schedulability analysis.

D.3.2 Efficient Implementation Techniques

Note that the Annex does not require that protected objects be implemented in any specific way. However, it is intended that the model be implementable via an efficient non-suspending mutual exclusion mechanism, based on priorities. Such mechanisms are well understood for static priority systems where the only priority inheritance is through locks, but the inclusion of dynamic base priorities and other forms of priority inheritance complicates the picture.

We will argue the adequacy of the specifications in this Annex to permit an efficient non-suspending mutual exclusion mechanism based on priorities, under certain assumptions. In this discussion it is assumed that priority inheritance occurs only via the mechanisms specified in this Annex, and the only processing resources that can be required by a task are a processor and protected object locks. Here, a lock is an abstraction for having mutually exclusive access to a protected object. The operations on locks are *seize* a lock, and *release* a lock. Locks are not transferable; once seized, a lock is not allowed to be seized by another task until it is released by the last task that seized it. It is assumed that protected objects can be implemented using locks. It is also assumed here that when the base priority of a task is lowered, it yields its processor to any task with active priority equal to the new base priority, in particular to one that is holding a protected object lock with that priority as its ceiling, if such a task exists. The cases of a single processor and a multiprocessor will be considered separately.

Suppose there is only one processor. Assume that the implementation of the seize operation is not able to block the task. We will argue that mutual exclusion is still enforced, by the scheduling policy. In particular, suppose a task, T1, is holding a lock on a protected object, R1. Suppose T2 is another task, and T2 attempts to seize R1 while T1 is holding it. We will show that this leads to a contradiction.

Let C(R) denote the ceiling of a protected object R, Bp(T) denote the base priority of a task T, and Ap(T) denote the active priority of a task T. If Ap(T2) > C(R1), T2 would not be allowed to attempt to lock R1. (This rule is enforced by a run-time check.) Therefore, Ap(T2) £ C(R1).

T1 must run in order to seize R1, but it cannot be running when T2 attempts to seize R1. So long as T1 is holding R1, it cannot be blocked. (This rule can be enforced statically, or by a run-time check.) Thus T1 must be preempted after it seizes R1 but before T2 attempts to seize R1. When T1 is preempted, it goes to the head of the ready queue for its active priority, where it stays until it runs again. Note that the active priority of T1 cannot be changed until it runs again, according to the reasoning in D.1.2: changes to base priority are deferred while T1 is holding the lock of R1, and T1 cannot inherit higher priority since it is not blocked (and not running) and must already have started activation.

For T2 to attempt to seize R1 while T1 is on the ready queue, T2 must have higher active priority than T1, or have been inserted at the head of T1's queue after T1 was preempted. The latter case can be eliminated: for T2 to be inserted at the head of T1's ready queue, T2 must be preempted after T1; to be preempted after T1, T2 must be running after T1 is preempted; to be running after T1 is preempted, T2 must be at the head of the highest priority non-empty queue; this queue must have higher priority than Ap(T1), since T1 is at the head of its own queue. Thus, in either case, T2 must be executing with higher active priority than Ap(T1), some time after T1 is preempted and while T1 is still on the same priority queue. That is Ap(T1) < Ap(T2).

Since T1 is holding R1, it follows that C(R1) £ Ap(T1) < Ap(T2) at the first point where T2 runs after T1 is preempted, and while T1 is still on the same ready queue. Before T2 attempts to seize R1, the active priority of T2 must drop to a value no greater than C(R1). (This is enforced by a run-time check.) The active priority of T2 cannot drop below Ap(T1), or T1 would preempt. This leaves the possibility that the active priority of T2 drops to exactly Ap(T1). But in this case, the implementation must cause T2 to yield to T1, as part of the operation that changes the base priority of T2 (see [RM95 D.5]). Thus, T2 cannot execute and so cannot attempt to lock R1.

In conclusion, for a single processor, the scheduling policy guarantees that there is no way a task can execute to attempt to seize a lock that is held by another task, and thus, no explicit locking mechanism is required.

On a multiprocessor, it is clear that priorities alone will not be sufficient to enforce mutual exclusion. Some form of interprocessor locking is required. Suppose this is accomplished by means of a busy-wait loop, using an atomic read-modify-write operation such as test-and-set. That is, a processor attempting to seize a protected object lock "spins" until it is able to set some variable in shared memory, which indicates that the protected object is locked. Thus, there is no danger of loss of mutual exclusion. The new problem is deadlock.

A necessary (but not sufficient) condition for deadlock is a cycle of "wait-for" relationships between pairs of tasks. In this case, there are two kinds of wait-for relationships. The obvious kind is where task T is spinning for a lock R held by task T'. The less obvious kind is where T is waiting for a processor that is being held by the spinning task T'.

The priority locking scheme does not prevent a direct deadlock situation of the obvious kind, in which task T1 is spinning waiting for a lock held by task T2, and task T2 is spinning (on another processor) waiting for a lock held by task T1. Fortunately, the user can prevent this kind of a deadlock, by not using nested protected operation calls, or by imposing a fixed ordering on nested protected operation calls.

A more serious problem, if it could occur, would be a deadlock involving a task waiting for a processor that is busy spinning for a lock. For example, suppose task T1 seizes R1, T1 is preempted by T2, and then T2 starts spinning in an attempt to seize R1. This would result in a deadlock if T2 is spinning on the only processor where T1 can execute at this time. This kind of deadlock would be serious, since it would be hidden inside the implementation, where the user could not prevent it.

Fortunately, this kind of deadlock is prevented by the priority ceiling locking scheme. For tasks executing on the same processor, this is obvious. Since T1 inherits the ceiling priority of R1, an exception will be raised if T2 tries to lock R1 while its active priority is high enough to preempt T1. The priority ceiling scheme also prevents such deadlocks in situations involving tasks executing on different processors. For example, suppose task T1 (executing on processor M1) locks R1 and task T2 (executing on M2) locks R2. Suppose task T3 preempts T1 and attempts to lock R2, while T4 preempts T2 and tries to lock R1. For this to happen, either T3 or T4 must fail the priority ceiling check. We will show this for the general case. Suppose there is a cycle of wait-for relationships. If T is waiting for T', we have either:

1 T is spinning for a lock L held by T', so Ap(T) = C(L) £ Ap(T'). (Note that we may have C(L) < Ap(T') if T' performs a nested protected operation with higher ceiling, while it is still holding the lock L.)

2 T is waiting for a processor held by T', which is spinning for some lock L', so Ap(T) < Ap(T') = C(L).

In order for the cycle to happen, both relationships have to hold for at least one pair of tasks, but then we have a contradiction.

D.3.3 Deadlock Prevention

It is a consequence of the priority ceiling locking scheme that an application cannot create a deadlock using protected subprograms on a single processor. This follows directly from the fact that a task executing a protected object operation cannot be preempted by any other task that requires access to that protected object.

Note that this is distinct from the problem of deadlock discussed above, which is within a particular multiprocessor implementation of ceiling locking. In the case of a multiprocessor, the priority ceiling locking does not prevent an application from constructing a deadlock with protected subprograms, but it still can be used to prevent deadlocks that might be caused by the implementation.

D.3.4 Implementing Over an OS

Priority ceiling locking may be very costly (possibly even impossible) where Ada tasks are implemented using the services of certain operating systems or executives. In particular, locking a protected object requires setting the active priority of a task to the protected object ceiling, or making the task entirely non-preemptable in some way, or using specialized operating system primitives. If there is no way to do this at all over a particular OS or executive, [RM95 1.1.3(6)] may be used to relieve the implementation from supporting priority ceiling locking. A more difficult case would be where there is a way to change a task's priority, but this operation is very costly. This might be true, for example, where Ada is implemented over a version of POSIX threads which does not support the priority ceiling emulation option for mutexes.

We considered whether an implementation of this Annex should be allowed to support priority ceiling locking but to only use it on those protected objects for which it is requested explicitly. The rationale is that the cost of priority changes may be too high to tolerate in general, but the user may determine that it is worthwhile in some specific cases. The extra implementation overhead of supporting two kinds of locks would be offset by the gain in efficiency for those cases (perhaps the majority) where ceiling locking is not used. Presumably, an implementation could still use priority ceiling locking with a default priority ceiling when no ceiling is specified, but could also use some other locking protocol in this case.

If this proposal had been accepted, then there would have been a problem with the check for ceiling violations. To reap the maximum benefit in efficiency from not raising the active priority of a task when it locks a protected object, no check for ceiling violations should be required either. This would result in portability problems going from implementations that use a mixture of priority-ceiling and non-priority-ceiling locking (A) to implementations that use priority-ceiling locking for all protected objects (B). For example, suppose PR1 has no specified ceiling, PR2 has a ceiling specified that is somewhere below `Priority'Last`, and all callers of PR1 and PR2 happen to have active priorities below `Priority'Last`. Suppose some operation of PR1 calls some operation of PR2. With implementation (A), this call to PR2 would always be safe, since the active priority of a task is not raised by calling PR1. With implementation (B), the call to PR2 from inside PR1 would be made at the default ceiling priority of PR1, which is `Priority'Last`. This would violate the ceiling of PR2, causing `Program_Error` to be raised.

While this approach could have worked, it did not seem that there was enough user benefit to justify the loss of portability. If the implementation did not support priority-ceiling locking, because the cost of priority changes is prohibitive, but the application designer judged that avoiding priority inversion justifies the overhead of the priority changes, the application might have to adjust the active priority explicitly, by setting the base priority. This would mean calling `Set_Priority` before and after calls to the protected operations where priority inversion is of concern. Naturally, techniques like this are prone to race conditions, especially in the presence of interrupts. Also, it is not clear that the overhead of `Set_Priority` would be any smaller than the direct OS support for priority ceilings.

This Annex provides a prioritized model of mutual-exclusion which is integrated with interrupt disabling when shared data is used between the interrupt handler and normal tasks. There may be situations where this model will be too elaborate and costly. Examples of this may be certain operating systems, or implementations over bare machines which traditionally have disabled preemption and/or interruption for this purpose. This implementation model is allowed by the Annex and is consistent with the other priority rules. In particular, the tasks' priorities still maintain the *granularity* and the range of the type. However, for protected object ceilings, implementations are allowed to round all priority values in the `Priority` range to `Priority'Last`, and those in the `Interrupt_Priority` range to `Interrupt_-Priority'Last`. The net effect of such rounding is that on each call of a protected object with ceiling in the lower range, preemption (or dispatching) is disabled. When a protected object in the interrupt range is involved, all interrupts are disabled. This reduces the number of protected object ceilings to only two values which makes the approach quite similar to the disable preemption/interruption employed by existing kernels. The rest of the priority rules are not affected by this approach.

D.3.5 Implementation and Documentation Requirements

The implementation will require protection of certain processing resources, not visible to the application, from concurrent access. For example, a storage allocation operation generally requires exclusive access to the data structure that is used to keep track of blocks of free storage. Likewise, run-time system operations involved in rendezvous generally require exclusive access to an entry queue. It would be natural to implement such critical sections in the run-time system using protected objects. If this is done, it is important that an application task with high active priority does not unwittingly violate the priority ceiling of one of these run-time system structures.

In order to reduce the likelihood of such problems, the implementation requirement is for the ceilings of such resources to be at least `Priority'Last`. This is intended to make such unwitting ceiling violations impossible unless the application uses interrupt priorities. An application that does use interrupt priorities is responsible for ensuring that tasks avoid operations with low ceilings while they are operating at interrupt priority. The rules against performing potentially blocking operations in protected bodies are expected to help in this respect, by ruling out most of the operations (other than storage allocation) that are likely to require locking run-time system data structures. In addition, the implementation is allowed to limit the RTS operations that are allowed from an interrupt handler.

An application that uses interrupt priority levels will need to know of any implementation uses of resources with lower ceilings, in order to avoid ceiling violations. The implementation is required to provide this information.

D.4 Entry Queuing Policies

The Ada 83 rule that entry calls be served in FIFO order may result in priority inversion, which can cause a loss of schedulable processor utilization. The same issue has been raised regarding the choice between open alternatives of a selective accept statement, which is unspecified by Ada 83. However, for upward compatibility reasons any existing Ada applications that rely on FIFO entry queuing order should continue to work with Ada 95. For this reason, the default entry queuing policy, specified in [RM95 9.5.3] and [RM95 9.7.1] is still FIFO. (This contrasts with the other two policies where, if no pragma is supplied, the policies are implementation defined.)

In addition, the user can override the default FIFO policy with the pragma `Queueing_Policy` thus

```
pragma Queuing_Policy(Priority_Queuing);
```

which stipulates the alternative policy which all implementations supporting this Annex must provide.

An approach that we rejected was for a user to be able to specify different entry service policies for each entry or task. Based on analysis of existing Ada run-time system technology, it appeared that requiring the Ada implementation to support per-entry or per-task selection would impose significant distributed execution-time overhead and would significantly complicate the Ada run-time system. Moreover, the need for mixed policies for entry service has not been demonstrated.

The solution adopted here is that a user can rely on applications that select priority queuing on a partition-wide basis being portable to any implementation that complies with this Annex. It is left to the implementor to decide whether to support finer-grained (i.e. per-task or per-entry) selection of queuing policy, based on customer demand.

It is possible that the choice of entry queuing policy may cause different code to be generated. Thus, the entry queuing policy must be specified no later than the point where each affected entry is declared.

Since certain compilation units (including packages that are intended to be reusable) may depend for correctness on a particular policy, it is important for the compiler or linker to be able to detect inconsistencies in such dependences. This can be guaranteed so long as the choice of policy is fixed at the time the entry is declared, and achieved through the general mechanism of compatible configuration pragmas (see [RM95 10.1.5]).

D.4.1 FIFO Queuing

FIFO queuing is provided for upward compatibility with Ada 83. If the correctness of a particular unit relies on FIFO entry queuing, it may specify this policy explicitly by

```
pragma Queueing_Policy(FIFO_Queuing);
```

This is important when other units that are included in the same partition specify `Priority_Queuing`. If `FIFO_Queuing` was just the default, all units in the partition would have inherited, in this case, the `Priority_Queuing` policy, as opposed to being illegal (due to conflicts) which is the desired behavior. Implementations may support both policies in the same partition, but then the interactions between the policies are implementation-defined.

Nothing is specified about the rules for choosing between open alternatives of a selective accept statement, since there is no consensus among existing Ada compilers or Ada users as to how this choice should be resolved in a FIFO queuing environment. Leaving the rule unspecified provides upward compatibility.

D.4.2 Priority Queuing

Substantial consensus seems to have evolved that priority scheduling requires priority-ordered entry service. Priority-ordered entry service eliminates a source of unnecessary priority inversion and more consistently expedites the execution of higher priority tasks. Therefore, the `Priority_Queuing` policy is specified as a user-selectable option that must be supported by all real-time implementations.

Priority inheritance through queued entry calls was intentionally omitted from the `Priority_Queuing` policy. Several models for priority inheritance through queued calls have been proposed in the literature. However, there is no hard analytical data to support choosing one of these priority inheritance models over another. The basic need for providing access to shared data without unbounded priority inversion is already supported by the inheritance feature of priority-based protected objects. The implementation overhead of more complex forms of priority inheritance is sufficiently high that requiring it is not sensible, if only one standard entry queuing and priority inheritance policy is specified.

The decision to require priority-order selection among open alternatives of selective accept statements, and among open entries of a protected object is based on the desire to avoid unnecessary priority inversion. It is understood that there will be some additional implementation overhead, but this overhead is believed to be justified by the potential gain in schedulability.

Priority ties can occur. If there are several open accept alternatives of a selective accept statement, or several open entries of a protected object, there may be several highest priority calls. For predictable behavior, a tie-breaking rule is needed. Textual order of the select alternatives or entry declarations is specified, on the grounds that this provides the greatest degree of predictability and direct control to the programmer. In addition, it is believed to be easy to implement.

The choice of tie-breaker rules does limit implementation choices. Even though the semantic model for entries is based on there being a separate queue for each entry, the implementation may choose not to provide separate physical queues. For example, when a task reaches a selective accept statement or is exiting a protected object the implementation might do one of the following:

- Do a full scan of all tasks in the system, in priority order, to see whether any of them is trying to call one of the currently open entries. (This might already be very inefficient, due to the rules about the effects of priority changes on queued calls.)

- Search a single priority-ordered queue, which is associated with the accepting task or the protected object, to find the first caller that is calling one of the currently open entries.

With data structures that combine calls to different entries, it would be harder to select the call that corresponds to the lexically-first accept alternative or entry body declaration. The most natural tie-breaker between equal priority calls would be some form of FIFO. On the other hand, if the implementation does maintain a separate queue for each entry, then it may be easier to break ties based on textual order. The present rule takes the point of view that pinning down the choice of tie-breaker rule is important enough to the application that the implementation choice can be so limited.

Reordering of Entry Queues

The decision to specify what effect task priority changes have on queued entry calls is based on the goal of implementation uniformity. The rules distinguish between "synchronous" entry calls and those associated with asynchronous select statements.

Entry calls associated with asynchronous select statements are not reordered when the priority of the task that queued the entry call changes. This decision is largely based on consideration of implementation efficiency and the fact that the task is not waiting for these entry calls. Otherwise, every time the priority of a task changed, its asynchronous calls would be deleted and reinserted in their various entry queues. This would conceivably happen even for temporary changes in active priority associated with starting and completing a protected action in the abortable part.

The priority of an entry call must be determined before the task has locked the protected object, because it is a consequence of the priority ceiling locking mechanism that, at the time the protected object lock is seized, the active priority of the task making the entry call will always be equal to the ceiling of the protected object. If the priority of the call were taken at this time, it would be the same for all callers to the entry, which would defeat the purpose of priority queuing. The desired semantics can be implemented by recording the calling priority as an implicit parameter associated with the queued call, before the protected object is locked.

In an earlier version, asynchronous entry calls were reordered as well, but only upon base priority changes. However, this introduced certain problems. In particular, the task that caused the priority to change would probably have to do the reordering itself, which would imply getting the locks on the various protected objects with asynchronous calls. This would not be possible if the ceiling of the protected object were below the active priority of the task causing the priority change. By contrast, a task waiting on a synchronous entry call can do its own queue reordering, presuming its new priority is not above the ceiling. If it is, it is considered a bounded error, and `Program_Error` might be raised in the waiting task. This is consistent with the behavior which would have occurred if the priority had been raised above the ceiling just before the task originated the entry call, so it was deemed appropriate.

We also considered the idea of requiring that the priority of a task not change while it is on an entry queue. This would eliminate the question of queue reordering, but it has several complicated consequences. Most serious of these seems to be that a task could not lock a protected object while it is on an entry queue and executing the abortable part of an asynchronous select statement. Other limitations would also need to be imposed, including extension of the deferral of base priority changes to cover the case where a task is on an entry queue. This would in turn increase the overhead of entry calls.

More serious is that this limitation would interfere with the use of an entry queue to control user-defined scheduling. It seems plausible to create the equivalent of a ready queue using a protected entry queue, and then use dynamic priority changes coupled with other operations on the protected object to implement a scheduling policy. If dynamic priority changes were not permitted, a hypothetical scheduler would have significantly less flexibility in controlling the order of service of the various tasks on the entry queue.

In contrast to asynchronous calls, a synchronous entry call is reordered upon a priority change in the waiting task. This was deemed important for consistency of the priority model, for example when dynamic priority changes are used to implement mode changes or a user-defined scheduling policy. Moreover, since dynamic priority changes are not expected to be frequent and there are other factors that are already likely to make the `Set_Priority` operation complicated, the extra complexity of checking whether the task is waiting on a (synchronous) entry call does not seem too high.

We considered whether, when a task's priority changes, the new position of its queued call should be based on the new base priority or the new active priority. Since a waiting task could not be inheriting priority from a protected object, the active priority will be the same as the base unless the task is in a rendezvous or activating. (This assumes there are no extra implementation-defined sources of priority inheritance.) In these latter cases, it seems the call should continue to inherit the priority from the activator or entry caller. Therefore, the new priority of the call is specified as the new active priority of the caller after the new base priority is set.

Another semantic detail is whether adjustment of priority causes loss of FIFO position within a priority queue, in the case that the new active priority is the same as the old active priority. For conceptual consistency, `Set_Priority` is specified as having the same round-robin effect on queued entry calls as it does on the task's position in the ready queue(s).

D.4.3 Other Queuing Policies

The possibility of specifying other standard entry queuing policies, including some with priority inheritance, was also considered. The decision not to specify such alternative policies in the Annex was based on a general design goal of avoiding multiple solutions for a single problem. This would be contrary to the intent of the Annex to encourage uniformity among implementations and portability among applications. Moreover, supporting each alternative policy would involve significant implementation cost. Therefore, requiring every implementation of the Real-Time Systems Annex to support several alternative policies would not be sensible. The intent is that there be one policy that all Annex implementations are required to support; this is the `Priority_Queuing`. For applications that require upward compatibility with Ada 83, `FIFO_Queuing` is also specifiable. The basic model defined in this Annex allows experimentation with new policies, and the introduction of new solutions based on market demands. Therefore, implementations are permitted to define alternatives, but portable applications should rely only on the `Priority_Queuing` and `FIFO_Queuing` policies.

D.5 Dynamic Priorities

The ability to vary the priorities of tasks at run-time has been so widely demanded that most Ada implementations provide some form of dynamic priority facility. The package `Ada.Dynamic_-Priorities` provides such a capability in a portable manner. The interactions of priority changes with other aspects of Ada task semantics are also defined. The following subprograms are provided

```
procedure Set_Priority(Priority: Any_Priority;
                       T: Task_ID := Current_Task);

function Get_Priority(T: Task_ID := Current_Task)
                       return Any_Priority;
```

where the priority is the base priority rather than the active priority.

Versions of `Get_Priority` and `Set_Priority` with no explicit task parameter (and so applying implicitly to the calling task) are unnecessary since this capability is provided by the `Current_Task` as a default parameter. Calling such operations might be slightly faster, but they would clutter the interface, and since these operations are not trivial anyway, the benefit did not seem to be worthwhile. (Compilers recognizing this special case can still optimize it by calling a separate entry point in the RTS.)

Calling `Get_Priority` for a terminated task raises `Tasking_Error`. This allows the implementation to reclaim the storage devoted to the task control block upon task termination. Querying the priority of a completed or abnormal task is allowed and has a well-defined meaning since such tasks may still be executing and may still use the CPU, so providing user access to their priorities makes sense.

A function for querying the active priority of a task was intentionally omitted. This is partly because the active priority can be volatile, making the result unreliable. In particular, querying the active priority inside a protected body will not return useful information, since the task will always be executing at the priority ceiling of the protected object. Another reason is that it is likely to be difficult to implement such a function on some systems. Moreover, requiring this value to be available would rule out at least one efficient technique for priority inheritance, in which inheritance relationships are represented only by links from donor to inheritor, and the implementation does not need to explicitly compute the active priority of a task or to store it.

When the base priority of a running task is set, the task is required to go to the tail of the ready queue for its active priority. There are several reasons for this. First, this is what is specified in the SCHED_FIFO policy of [1003.4 93], after which the default task dispatching policy is modelled. Second, this is needed to prevent priority changes from violating the ceiling rules if priority inheritance is used to enforce mutual exclusion. For example, suppose task T1 is executing a protected operation of PR1, and task T2 preempts. Suppose T2 then lowers its own base priority to the ceiling of PR1. T2 is required to go to the tail of the ready queue at this point. This ensures that there is no danger of T2 trying to perform a protected operation on PR1. (Allowing T1 to preempt under these circumstances might also be desirable from the point of view of expediting the release of PR1.)

Deferral of the Effect of Priority Changes

The effect of `Set_Priority` on a task is deferred while the task is executing a protected operation, for several reasons. One reason is to prevent `Set_Priority` from forcing a task that is executing in a protected object to give up the processor to a task of the same active priority.

Another reason is to permit more efficient implementation of priority inheritance and priority changes. In particular, when entering a protected operation body, or starting a rendezvous, it is permissible to push the old active priority on a stack, from which it is popped when the protected operation is left, or the rendezvous ends. Note that there need be no additional execution time overhead for implementing this deferral, over that already imposed by deferring abortion, in the case that no priority change is attempted during the time the protected operation is executed.

For simplicity of implementation, priority changes are allowed to be deferred until the next abort completion point. This will be primarily useful in the context of target environments that have limited support for preemptive interthread or interprocessor signalling.

Taken from a user's point of view, deferring a change to the base priority of a task during protected operations should make no difference if the change is in the downward direction, since this would not affect the active priority of the task anyway. If the change is in the upward direction, the difference could be noticeable, but no requirement for immediate upward change of base priority during protected operations has been demonstrated. There may be a requirement for a temporary change to the active priority, but this is possible by calling an operation of a protected object with high enough ceiling.

Deferring the effect of changing the base priority also eliminates some semantic questions. One of these is whether the base priority of a task should be allowed to be raised higher than the ceiling priority of a protected object in which the task is currently executing. Allowing this would constitute a retroactive violation of the rule that a task cannot call a protected operation of a protected object while its active priority is higher than the protected object ceiling (the active priority is of course never less than the base priority).

Dynamic Priorities and Ceilings

When ceiling priorities and dynamic changes to priorities are supported in the same environment, some interactions with other language features are unavoidable. The source of these problems is mainly the inherent conflict between the need to arbitrarily and asynchronously change the task base priorities, and the ceiling model where a more disciplined usage of priorities is required. The problems get more serious if the effect of such misuse affects not just the program behavior, but also the correctness of the implementation. At least two interesting cases exist:

1 As part of the `Set_Priority` operation, a protected entry queue may have to be reordered. This happens when the affected task is waiting on a protected entry call (see [RM95 D.4]). If the task calling `Set_Priority` has an active priority higher than the ceiling of the relevant protected object, it will not be able to accomplish this reordering due to a ceiling violation. To circumvent this problem, it can awaken the waiting task which can then itself reorder the queue and continue waiting.

2 A call queued on a protected entry queue may sometimes need to be cancelled. This happens when the task is aborted, its currently executing abortable_part is aborted (and it has some nested calls), or when the abortable_part completes normally and the triggering call needs to be removed from its queue. If the task's base priority was raised after the call was initially queued and remains too high when the call needs to be removed, it might fail to remove the call due to ceiling violations (since such removal involves locking the protected object). This situation is considered a bounded error, and can result in the task's priority being temporarily lowered to accomplish the cancellation of its call.

We considered other alternatives as solutions to the above problems. For the first case, we looked into the possibility of temporarily lowering the priority of the task calling `Set_Priority`. This has the obvious problems of potentially introducing priority inversions, complicating implementations, and presenting a non-intuitive model to the user. We also looked at allowing the reordering to be deferred. This is also undesirable: the deferral may be too long and there may be several priority changes during this time. Resuming the affected task in order to accomplish the reordering was chosen as the suggested implementation model, since a similar mechanism is already required to support abort of a low-priority task by a high-priority task. We also looked at the possibility of limiting the effect of the `Set_Priority` call such that it will raise the priority only to the minimum of the ceilings of protected objects either held by or being queued on by the task. Again, it was not clear that these semantics are desired, and it would certainly add a substantial cost to the implementation.

The second situation introduces a problem that if not addressed might make the implementation of finalization (as part of abortion) impossible. Here, a call is already queued and it must be removed; just raising an exception is not acceptable since this will not solve the problem. We considered various solutions, but ultimately declared the situation a bounded error, and allowed the task when it needs to cancel its call to have its priority temporarily lowered. The temporary priority inversion was not felt to be serious since this is considered an error situation anyway.

Example of Changing Priorities of a Set of Tasks

```
type Task_Number is range 1 .. 4;
type Mode_Type is range 0 .. 2;
Task_Priority: array (Task_Number, Mode_Type) of Priority := ... ;

protected Mode_Control is
   procedure Set(Mode: Mode_Type);
   pragma Priority(System.Priority'Last);
end Mode_Control;

protected High_Priority_Mode_Control is
   procedure Set(Mode: Mode_Type);
   pragma Interrupt_Priority;
end High_Priority_Mode_Control;

use Dynamic_Priorities;
protected body Mode_Control is
   procedure Set(Mode: Mode_Type) is
   begin
      High_Priority_Mode_Control.Set(Mode);
      Set_Priority(Task_Priority(1, Mode),T1);
      Set_Priority(Task_Priority(2, Mode),T2);
   end Set;
end Mode_Control;

protected body High_Priority_Mode_Control is
   procedure Set(Mode: Mode_Type) is
   begin
      Set_Priority(Task_Priority(3, Mode),T3);
      Set_Priority(Task_Priority(4, Mode),T4);
   end Set;
end High_Priority_Mode_Control;
```

The table `Task_Priority` specifies the priorities that the tasks `T1` through `T4` should have, for every mode. Here, in order to avoid blocking every task for a long time, the priority changes are done in stages, at two different active priorities, via two protected objects. The task doing the priority change starts with a call to the lower-priority protected object. This calls the next higher level. The priority adjustments of lower priority tasks can be preempted by the execution of the higher priority tasks.

Metrics

The purpose of the metric for `Set_Priority` is to specify the cost of this operation, compared to other operations, for a case where it should be about as low as can be expected. This metric may be critical for some applications, which need to perform priority changes under time constraints, but the inherent complexity of `Set_Priority` is likely to make it time-consuming.

Of course, complicating factors such as entry queue reordering may make the execution time of `Set_Priority` worse than would be indicated by this metric. The possibility of including more metrics, such as for a situation involving entry-queue reordering, was considered. This idea was rejected on the grounds that it would only be of interest for applications that change the priority of tasks with queued entry calls. Special cases could not be covered uniformly to this level of detail without greatly increasing the number of metrics. Finally, this metric would cover a large part of the RTS code itself, and not just the priority change operation proper, thus it will be influenced by many factors diminishing the value of the specific metric to the user.

D.6 Preemptive Abort

A requirement has been expressed for "immediate" task abortion. There appear to be several motivations for wanting immediate abortion:

1 To stop the task from doing what it is currently doing before it can "contaminate" the application further, possibly with dangerous consequences. A task can contaminate the application by changing the system state or wasting processing resources.

2 To be certain that the task has stopped executing, so that the aborter can proceed without fear of interference from the aborted task (e.g. I/O, rendezvous, writing on shared variables).

3 To be certain that the aborted task does not continue executing indefinitely, as it might if it were (due to error) in an infinite loop without any abort completion points (see [RM95 9.8]).

There are several possible meanings of "immediate" in this context:

- Before the abort statement completes. This is easy to define and implement, but it may take a long or indeterminate time to complete, and so might require blocking the task that executes the abort. It satisfies requirement (2) only.

- Before the affected task(s) are allowed to execute further. This is easy on a single processor, but may be too costly or impossible on a multiprocessor. It satisfies all three requirements.

- As soon as the implementation can do it, and certainly within a bounded time. On a single processor, this would have the same meaning as above, but would allow some delay on a multiprocessor. It satisfies requirements (2) and (3). Whether it satisfies (1) depends on the implementation.

The third meaning of "immediate" seems like the best compromise. This is the basis for the specifications in this section. With respect to what actually has to happen as part of the immediate activity, [RM95 9.8] defines what is included in the completion of an aborted construct. Specifically, [RM95 9.8] requires part of the effect of the abort statement to occur before that statement returns (e.g. marking the affected tasks and their dependents as abnormal). The requirements in the Annex go further and address the completion and finalization of the aborted constructs.

The key requirement here is that the abortion be preemptive, in the sense that abortion should preempt the execution of the aborted task, and if abortion requires the attention of another processor, the abortion request should preempt any activity of the other processor that is not higher in priority than the aborted tasks.

Note that the requirement for bounding the delay in achieving the effect of abortion can be satisfied on a multiprocessor, even if it is not possible for one processor to interrupt another. One technique is to use a periodic timer-generated interrupt on each processor, which causes the processor to check whether the currently executing task has become abnormal.

An alternative was considered to allow the task calling the abort statement to be blocked until all the aborted tasks have completed their finalization, and for those tasks to inherit the blocked task's priority while it is blocked. This would be a change from Ada 83, where it is only necessary to wait for the aborted tasks to become "abnormal", and Ada 83 did not have user-defined finalization. Certainly, one of the reasons for aborting a task may be to release resources that it is holding. The actual release of such resources may be done during task finalization. However, waiting for finalization is not always possible, since a task may abort itself (perhaps indirectly, by aborting some other task on which it indirectly depends). In this case, it is not possible for the task calling for the abortion to wait for all the aborted tasks (including itself) to complete their finalization. Another problem is where the abort statement is inside an accept statement, and the task being aborted is the caller in the rendezvous. In this case, forcing the aborter to wait for the aborted task to complete finalization would result in a deadlock. The problem with self-abortion could be resolved by releasing the aborter to perform finalization, but the problem with rendezvous does not seem to be so easily resolved.

The ability to wait for a collection of tasks to complete finalization is partially satisfied by two other mechanisms. One of these is the rule that requires blocking of a completed task master until its dependent tasks are terminated. If the tasks being aborted are not dependent, another partial solution is to use the delay statement and the `Terminated` attribute to poll the aborted tasks until they have all terminated. However, none of these mechanisms fully accomplishes the objective.

Not allowing abortion to cause blocking has several benefits. In real-time applications, there are situations where the task executing the abort statement does not wish to wait for the aborted task to complete; in this case it could also be said that requiring the task to block is not "immediate" abortion. If the task executing the abort statement were to be blocked, unbounded priority inversion would be possible unless the tasks being aborted inherit the priority of the blocked task. This form of inheritance is undesirable for reasons explained in the next paragraph. A final benefit is that in this way, the treatment of abortion of a task via the abort statement is more similar to the abortion of a sequence of statements via a change to a barrier caused by a protected operation, since executing the body of a protected operation can never involve blocking of the calling task.

Irrespective of the decision not to block the task executing the abort statement, there are other reasons for not requiring that aborted tasks executing finalization code inherit the priority of the task executing the abort. First, this would introduce a new form of one-to-many inheritance, with the associated additional implementation complexity. Second, if the aborted task is a low-priority task, and the aborter has high priority, it might not be appropriate to suspend the aborter while the aborted task finalizes. Third, if the active priority of the aborted task could be raised by abortion, it would be necessary to take into account all abort statements, as well as task dependency hierarchies, in determining protected object ceiling priorities; otherwise, the active priority of a task might violate the ceiling of a protected object during finalization code.

Note finally, that if the user does want to make the finalization of the aborted task happen faster, the only solution is to raise the aborted task's base priority either before or after issuing the abort. Doing it afterwards enables the priority to be set higher than that of the aborting task; if the aborted task is already terminated no harm is done

```
    Abort_Task(Joes_ID);                       -- take that
    Set_Priority(Priority'Last, Joes_ID);      -- die quickly you dog
```

While this approach is not that elegant, it is expected to satisfy such a need.

Documentation Requirements

It is clear that interprocessor communication delays may cause abortion to take longer on some multiprocessor systems. The predictability of such delays is bound to depend on the implementation, and the duration may depend on what other activities are going on in the system at the time. It is important that the designer of an application that uses abortion be able to determine whether this is going to be a problem for a particular application.

Metrics

The execution time of an abort statement is intended only to be a sample of the execution time, in a non-pathological case. Of course the actual execution time will vary, depending on factors such as the number of tasks being aborted, their current states, and their dependence relationships. Providing an upper bound would therefore require specification of more conditions.

The intent of the upper bound on the additional delay for a multiprocessor is primarily to require the implementor to verify that such an upper bound exists. The specific value is less important than the existence of such a value. There must be some upper bound on the delay if abortion is to be useful in a real-time system. An upper bound may not be able to be measured directly, but it should be possible to (over-) estimate a bound by adding the upper bound of the communication delay to the upper bound of the local processing time.

The intent of the metrics for asynchronous transfer of control is to tell whether this capability is implemented efficiently enough to be useful in time-critical applications. Potentially, there is a great gap in performance between an implementation of asynchronous transfer of control that is based on creating a separate thread of control for the abortable part, versus an implementation that uses the context of the same task. The intent is that such a gap can be discovered by the metrics.

D.7 Tasking Restrictions

This section establishes that the Ada standard permits the development of highly optimized implementations for restricted tasking models. It also defines a specific set of restrictions, that both serves as an example of what an implementation can do, and may encourage convergent development and usage.

Builders of hard real-time systems have observed that the full Ada tasking model is more general than they require, and imposes more overhead than they can afford. The existence of very lightweight executives for other tasking models suggests that significant performance improvements are possible for Ada tasking implementations, if some usage restrictions are observed.

Any Ada implementor can define a restricted tasking model and provide a run-time system that is optimized for this model. (In fact many implementations do so currently, but in a non-portable way.) However, Ada 83 has been misinterpreted to give the (incorrect) impression that this is a form of "subsetting", and therefore is not allowed by the language standard. It is not subsetting, as long as the implementor also provides support for the full tasking model, perhaps with different compilation and linking options. Thus, it appears desirable for the Real-Time Annex to endorse this form of optimization.

A restricted tasking model should permit simple and useful multitasking applications to be expressed, but simplify the implementation problem enough so that the size and execution time overhead of the run-time system need be no greater than with traditional real-time executives.

Therefore, the intent behind the model defined here is to satisfy the needs of many of the real-time embedded applications that have rejected the full Ada tasking model for being too heavyweight. These applications include some that consist of a fixed set of cyclic tasks, with periodic and aperiodic timing constraints. This has traditionally been a stronghold of the cyclic executive and rate-monotonic scheduling models. The intended scope of restricted tasking applications also includes some more complex applications, which are event-driven. These applications have traditionally used real-time executives that can dynamically create and schedule extremely lightweight tasks. This kind of system is organized into groups of simple tasks. A task group is created in response to an event, executes for a while, and goes away. There may be precedence relations and delays among tasks within a group, but an individual task never blocks to wait for another task. Each task within a group is very simple: it may be preempted, but otherwise it runs to completion without blocking. The events that can trigger the invocation of task groups include interrupts and actions performed by other tasks. This is a well established model of software architecture, and has been used for more than a decade in guidance and control systems, including radar and sonar tracking systems, process control, railroad signalling and so on. This is also the task model of classical scheduling theories (see [Coffman 73]).

D.7.1 The Chosen Approach To Restrictions

This Annex specifies a set of restrictions which should be such that the potential performance improvement justifies producing one or more special versions of the RTS according to the combinations of restrictions asserted in a particular program.

The `Restrictions` pragma (see [RM95 13.12]), which is a configuration pragma, takes two forms. One such as

```
pragma Restrictions(No_Task_Hierarchy);
```

indicates a simple yes/no restriction, whereas

```
pragma Restrictions(Max_Select_Alternatives => 5);
```

indicates some numerical restriction on the feature concerned.

Compliance

Compliance with this pragma means that all the parameters must be recognized by the implementation, and the associated limitations must be enforced. It is clearly important for the implementation to reject violations of the restrictions. Without such checking, much of the value of the pragma is lost. Checking itself can be of value as well. Where the implementation is for a development host, if the host is used in preliminary testing of an application that is eventually intended for an embedded target, enforcement of the pragma by the development host will help to identify code that will cause problems arising when the time comes to move the application to the final target.

Some of these limitations are to be checked at compile-time (such as for task hierarchies and the presence of finalization). For those that can only be checked at run-time, implementations are allowed to omit the checking code. Programs that violate the corresponding restrictions are considered erroneous. If a run-time check fails, `Storage_Error` should be raised. This exception is chosen because failure of these checks often indicates shortage of the storage allocated for a task — either the run-time stack storage or storage allocated within the task control block.

The permission to omit the run-time checks is given due to the recognition that a check for a given violation may be as complex and costly as the support for the restricted construct. One does not want the checks to be difficult to implement or for the checks to add any overhead to a model that is intended to allow a simple implementation. The resource utilization checks need to be done at run time, and may incur some run-time overhead, but they may be very important during the testing of a system. The decision on whether to omit the checks is therefore left to the implementation based on the particular situation.

The Specific Restrictions

The basic criteria for deciding upon the restrictions were:

- The restriction allows a faster or smaller run-time system.
- The advantage is distributed. If it is a speed improvement, it applies to operations other than those ruled out by the restriction. If it is a size reduction, it is not simply due to deletion of unused run-time system components, such as might be done automatically by a linker.
- Taking advantage of the restriction does not require a major change in compilation strategy for programs that follow the restriction from those that do not.
- A sample of real-time embedded systems developers felt that the restricted feature is not essential for a significant portion of their applications.

In addition, some restrictions have been included because a significant number of users and vendors felt that they were appropriate.

Some of the specific restrictions and the benefits they bring are as follows

- `Max_Tasks` (maximum number of task creations over the lifetime of the partition) plus `Max_Task_Entries` (maximum number of entries per task). These enable fixed RTS and task storage size. The amount of storage required by the run-time system for its own data structures and task workspace should be determinable statically, no later than load time.
- `Max_Task_Entries` plus `Max_Asynchronous_Select_Nesting` (maximum nesting of asynchronous selects). These enable a fixed Task Control Block (TCB) size. If each task can be represented in the RTS by a fixed-size task control block, the complexity of the RTS is reduced. A list of free TCBs may be allocated at load time, speeding up task creation.
- `No_Task_Hierarchy`; all tasks are library tasks. This means no overhead due to code executed to keep track of potential task masters, or to check for unterminated dependent tasks. Likewise, there should be no storage overhead for data structures to keep track of task master nesting. Abortion and task termination should not be complicated by the need to support hierarchies.
- `No_Task_Hierarchy` plus `No_Nested_Finalization` (all controlled objects at library level). These permit the storage for non-library-level collections to be allocated on the stack, and the stack space can be recovered without finalization when the stack frame is popped.
- `No_Dynamic_Priorities`. There are several known scheduling algorithms that do not use the capability to dynamically change the task's base priority. If it is known that the

base priority of a task is static, then it is possible to have much simpler and more efficient queue management and dispatching algorithms in the run-time system.

- `No_Asynchronous_Control`. Even though the semantics of this package are defined in terms of priorities, it is not clear that an implementation approach based on this semantic model is feasible on all possible targets. In general, the ability to asynchronously suspend the execution of another task is considered dangerous from the user's point of view, and may have distributed ramifications on the rest of the run-time system. Since in some applications, this feature will not be used (and may even be disallowed) it makes sense to allow for the corresponding restriction.

- `Max_Protected_Entries`. There are several very efficient algorithms for servicing protected entry queues when the maximum number of entries is statically known and relatively small. The main issue here is the requirement for evaluating barriers whenever the state of the protected objects changes. The language rules in [RM95 9.5.2, 9.5.3] make it possible to evaluate the barrier less often than it would otherwise be needed (provided that the compiler can determine that no "interesting" change has occurred since the last check). It was suggested that the implementation can use a bit-vector (usually of one word length) to represent the true/false state of the barriers, and then check this bit-vector (using only a small number of machine instructions) in appropriate places instead of reevaluating all the barrier expressions. For this approach to work, the number of possible entries should be known a priori. Since the implementation of such a technique often involves the compiler, it might be necessary for this information to be known before any unit is compiled.

- `No_Abort_Statements` plus `No_Asynchronous_Control`. These enable a number of further simplifications to the model.

- `Max_Task_Entries = 0`. Forbidding task entries could reduce the size of the run-time system code. It could also reduce the amount of rendezvous-related information that must be stored in task control blocks. Processing this information during task creation and termination is a source of distributed overhead. The implementation of abortion might also benefit by not having to take into account the special case of tasks engaged in a rendezvous as callers, though the rules requiring deferral during protected operations and finalization operations cast this into doubt. Protected objects provide a lighter-weight mechanism that is more suitable than rendezvous for data and control synchronization in small real-time embedded systems.

- `No_Terminate_Alternatives`. The terminate alternative may add distributed overhead but is less valuable in Ada 95 since a protected object will typically be used rather than a server task.

- `No_Implicit_Heap_Allocation`. We considered adding a restriction forbidding the compiler from using the heap implicitly (i.e. not as a direct result of using an allocator). Such a restriction can improve the deterministic behavior of the memory in the system, by making all memory usage visible and user-invoked. However, this was difficult to mandate as a requirement on the compiler and so we have adopted a slightly different approach. The compiler is allowed to reject (and document) such user constructs that require implicit heap allocation. This way, it can ensure that no such heap requests will be present at run-time, and so the heap usage will be avoided.

Certain other restrictions were considered, but were not included.

Specifying a mechanism for configuring the run-time system size limits was also considered. It was left implementation-defined, because the practical mechanism is outside the scope of the language. For example, one method is for the implementor to provide source code of a few run-time system packages, which contain configuration constants. The user could edit these, recompile them, and link them in with the rest of the run-time system and the user's application. Another method is to provide a configuration tool that edits the run-time system object code, inserting constants for the required limits at the required points. This same function might be performed by a linker or a loader.

Max_Storage_At_Blocking

This restriction deserves special mention. If a task is not permitted to be blocked while it is holding more than a fixed amount of stack storage, a much larger number of tasks can be supported, since only one dynamic stack storage area is required for each priority level (essentially, only the TCB of a blocked task needs to be saved). Traditional real time systems are designed to make this possible. Practical ramifications of this requirement include:

- Tasks should not have large local data objects or access collections.
- Tasks should not have entries whose parameters require a large amount of storage.
- Operations that may cause a task to be blocked should not be performed within deeply nested procedure calls or within a block statement that has large local data requirements.

One implementation model is to have a fixed pool of stack spaces that are shared by all tasks, or all the tasks at a priority level. On each processor, not more than one stack space will be needed for each priority level. The stack space for a given level must be configured (by the user) to be large enough to meet the largest stack requirement of any task that executes at that priority level. A task releases its stack area when it blocks, and is allocated a stack area when it wakes up again. Depending on how ready queues are structured, allocation of a stack area might be done at the point where the task wakes up, or as a special case in the dispatcher when it gets ready to do a context switch to a stack-less task. A slight variation of this approach would be to always allocate a small and fixed-size part of the stack to the task, and to allocate the larger part only when the task is ready. In any case, the implementation can go to a linked list of stack spaces, remove one, and link it to the fixed-size part of the stack. This could be kept simple, maybe to the point of just setting the saved stack pointer value and a link to the fixed part of the stack. For example, on a machine with register windows, the implementation could keep one register window stored in the TCB. When allocating a stack area, it would write the new stack pointer (base) into this saved register window. Then, when the task is resumed, the implementation would load registers from the TCB and the task would be running with the new stack.

The intention is that all requirements for non-volatile storage associated with a task be met by the task control block (or by a fixed-size extension of it). For example, this includes storage for the implementation of select statements, entry parameters, local variables, and local access collections. This means that any large non-volatile data used by a task must be declared in library-level packages or passed to the task by means of access values. The size of the task control block and the fixed part of each task's run-time stack is intended to be determinable no later than link time, so that a fixed-size pool of identical task control blocks can be pre-allocated at system initialization time.

D.8 Monotonic Time

The package `Ada.Real_Time` is similar to the package `Calendar`. It has a type `Time`, a function `Clock`, relational operations on time, and arithmetic operations for combining times and durations. In order to explain why such a "duplicate" of `Calendar` is needed, we first review why some real-time applications need facilities beyond those in package `Calendar`.

The inclusion of a standard calendar package and clock function in Ada seems useful. Certainly, the existence of a standard interface to time-keeping services that hides unimportant details of specific execution environments can be an aid to writing portable applications. However, a sample of existing practice in real-time Ada applications reveals that they frequently choose not to use the package `Calendar`. Perhaps the main reason is simply that `Calendar` is typically "political" time and so is not guaranteed to be monotonic since it may be adjusted by the effects of time zones and daylight saving changes.

Another issue is the diversity of time-keeping requirements among real-time applications. It does not seem feasible to meet all these with a single solution. Both the requirements and the hardware (or operating system) support available differ widely from one real-time application to another. At one extreme, a simple real-time controller might be able to use a single 16-bit counter-timer circuit, with very fine granularity but a rather narrow range. At the other extreme, a complex electronic navigation system may require an atomic clock that is precisely synchronized with a global time reference, and may have precision and range requirements that demand 64-bit time representation.

Given this diversity, it is natural to wonder whether Ada 95 should attempt to provide any standard time services at all other than the package `Calendar` which has to be retained for compatibility. To the extent that there are common requirements for such services within certain application domains, they should perhaps be the subject of a language-independent standard; but no such standard exists.

The exisiting delay statement and the delay alternative require the language to provide a clock for two reasons:

- *Coordination of Clock and delay*. If the application uses delay statements to control timing, the application's view of the time should be consistent with that of the implementation.

- *Timer resource sharing*. The implementation needs access to a timer for the implementation of the delay statements. If there is only one such timer (as is the case on some execution platforms), the implementation and application must share it.

Real-time applications clearly need the capability to block task execution until a specified time, and to impose a time limit on how long a task can stay blocked waiting for other operations.

We considered an approach of providing general mechanisms for an application to wait for an event, and to abort blocking operations in response to events. This would have allowed the application to provide its own timer services. The delay statement could then just be a special case of waiting for a time-out event signalled by the user-defined timer, rather than the implementation's default timer. This solution was dropped since the added complexity seemed out of proportion to the benefits.

The inclusion of the `Real_Time` package in this Annex is based on the realization that there was no choice but to provide a real-time clock which applications could use. Specifically, an application that requires time-outs on select statements must use the standard delay statement implementation. If the application needs to know what time it is, based on a time reference that is consistent with the delay, it must use a clock provided by the implementation.

The following general requirements can be identified for a clock that is used in conjunction with the delay statement, to schedule task execution and specify time-outs:

- Monotonically non-decreasing time value, incremented at a steady rate, with bounded discontinuities.

- Fine granularity.

- The ability to be used as the time reference in all forms of delay statement.

- Efficient implementability using clock facilities that are typical of most existing hardware, and real-time operating systems.

- Exact arithmetic on time and duration values, and precise conversion of rational-number durations to time intervals.

- A defined relationship to other time-related features of the language, including the `Calendar` package, `System.Tick`, and the `Standard.Duration` type.

The package `Ada.Real_Time` is intended to provide facilities that satisfy these requirements.

Some real-time applications have other requirements, such as

- Unique time-stamps. With fast processors and multiprocessor architectures, it is possible that for some implementations the clock may be read several times in one tick. Enforcing uniqueness in such an environment would amount to slowing down the clock reading operation.

- Synchronization with external time references. In some situations (such as where the external time reference is non-monotonic, or synchronization cannot be performed frequently enough to avoid large adjustments), synchronization may be incompatible with the requirements for monotonicity and bounded discontinuity of the clock.

These were considered but appeared to conflict with satisfying one or more of the other requirements and so were dropped.

D.8.1 An Ideal Clock

International Atomic Time (TAI), regulated by the Bureau International de l'Heure (BIH) and supported by the various national time references, is currently accepted as the most precise physical time reference. It is monotonic and does not have leap-seconds or seasonal changes. All the other time standards can be defined as a function of TAI. That is, any other standard of political or physical time can be defined as a function C = TAI + D(TAI), where D(TAI) is a piecewise constant function, depending on the current value of TAI. In an ideal world, and an ideal implementation of the language for real-time applications, there would be a clock function that returns the current TAI clock value. Language-defined functions could be provided to convert this time to an arbitrary time zone.

In practice, most Ada execution environments will not have access to an atomic clock. Even if such a clock is available, there may be a need to use other (less accurate) clocks, including the local time as perceived by a human operator or an operating system, or counter-timer circuits that are capable of generating interrupts.

D.8.2 Time Sources

A language implementation is limited by the actual time-keeping resources provided by the hardware, which are possibly filtered through an operating system interface.

In practice, several different kinds of time references are likely to be available to an Ada implementor. These have significantly different characteristics:

- Counter-timer circuit
- Calendar clock circuit
- Externally synchronized clock

A counter-timer circuit is a programmable hardware device which can be viewed as a register counting clock ticks. Such a timer is typically driven by a crystal oscillator, and can be read and reset to a specified value. A typical tick duration might be one microsecond.

A counter-timer can typically be used to generate an interrupt when a specified number of ticks have elapsed. It might then restart automatically at a specified value (a periodic timer) or wait to be reset (a one-shot timer).

Counter-timer circuits are comparatively inexpensive, and are easily added to a microprocessor-based design. Thus, in a specific hardware configuration of an embedded application, there may be several counter-timer circuits. However, these are not likely to be known and available to the implementation. The standard configuration of most processors typically has only a small number of counter-timer circuits (possibly one) that can be relied upon to always be available for use by the application and the language implementation. In small embedded real-time applications, these counter-timer circuits may be the only time reference available. The strengths of counter-timer circuits include:

- Small clock-tick, typically one microsecond or smaller;
- Monotonicity, subject to periodic wrap-around to zero;
- Very regular ticks;
- Ability to generate interrupts;
- Very regular periodic interrupts, in periodic mode.

Some limitations of counter-timer circuits include: jitter up to one clock-tick, variation in interval from one timer to another and with temperature, and a limited range before wrap-around.

A calendar-clock circuit is a programmable hardware device that is very similar to a counter-timer-circuit. The main differences are:

- The visible update rate (granularity) of the clock may be coarser (e.g. once per second). However, the underlying oscillator rate, and hence the accuracy, is likely to be just as high as the counter-timer.
- The range of times representable by the clock is large, perhaps 100 years.
- The clock may be programmable to automatically take into account leap years and seasonal political time changes, such as daylight savings time.
- Time values, instead of being a simple count of ticks, may be represented in terms of second, minute, hour, day, month, and year.

Various forms of externally synchronized time references may be available in a specific application. In a system requiring very precise global positioning there might be a local atomic clock, periodically synchronized with the TAI via a communications link. In a network, there might be a broadcast periodic "heartbeat", or a message-based averaging algorithm for keeping the local clocks of various network nodes synchronized within some tolerance. Generally, the frequency of external synchronization is limited, and if it relies on communications with external systems there may be times when the local system is cut off from its source of external synchronization. Typically, local clock or timer circuits are used to keep time between external synchronization points, so that a finer granularity of time is available locally.

In general, synchronization conflicts with regularity and fine granularity. That is, if the granularity of the clock is fine enough, synchronization will cause discernible irregularities in the rate of progress of the clock. Clock synchronization may require the local clock to jump forward or backward. Of these two, backward jumps are especially troublesome, since they can induce apparent ordering inversion if the clock happens to be used to determine the times of events immediately before and after a jump. However, an error in the measurement of an interval due to a forward jump can also be serious.

A good synchronization method can reduce the impact of clock adjustments by several techniques. Backward jumps may be avoided by arranging to skip some ticks of the local time reference until the desired adjustment is reached. Discontinuities due to forward jumps and skipped ticks may be smoothed by splitting a large adjustment into several smaller adjustments, separated by intervals. Better, the size of adjustments may be kept small by frequent synchronization. Still, these techniques are limited. In less than ideal circumstances, one must anticipate that a synchronized clock may be available but not be able to deliver as fine a granularity, or as regular a rate of progress, as unsynchronized time references that may be available locally.

Where Ada is implemented over an operating system, and so does not have direct access to time-keeping hardware circuits, it may be necessary to rely on the time-keeping services of the operating system. The operating system ultimately must rely on hardware devices similar to those described above, and it may or may not attempt to synchronize with other clocks; therefore, operating system time sources are subject to the same diversity of characteristics discussed above. In addition, they are subject to other limitations, including:

- Inability to access the full accuracy of the hardware, due to timer-programming decisions made by the OS implementor.

- Loss of accuracy as viewed by the user, due to system-call overhead.

- Delay between expiration of a wake-up time and notification being delivered to a waiting process, due to system overhead.

- Discontinuities, due to setting of the clock by a human operator or an unrelated application program.

- Disparate views, due to "environment variables" specifying different time zones for different processes within a system.

While these factors may affect the suitability of a particular operating system for a real-time application, they must be accepted as inherent limitations from the point of view of the Ada language. One is forced to assume that the time services provided by the OS have sufficient accuracy and low enough overhead to meet the needs of Ada applications on that system.

For the purposes of this discussion, whatever time sources are provided by an operating system are presumed to have characteristics similar to one of the three basic types of clocks mentioned above.

A Single-Clock Model

In a real-time application, there may be requirements that cannot be satisfied by any single time source that is available. As explained above, the actual time-keeping resources available in a specific environment may have significant limitations, and the choice of time references may require that one thing be sacrificed for another. For example, fine granularity may mean sacrificing range or synchronization, greater range may mean sacrificing granularity, and synchronization may mean sacrificing the regularity or fine granularity, all at the cost of higher overhead. It follows that if all of these properties are important for a given application, a combination of different time references must be used.

In some cases, it may be possible to provide a virtual interface that creates the illusion of a single time reference, using multiple time references in the implementation. For example, this is the case when a local timer is used to interpolate between points of synchronization with a remote clock. However, preserving this illusion is not always possible, or practical. In the extreme, there may be a fundamental conflict, as between steady tick rate and synchronization with an external reference. An implementation of a single-clock interface may be useless if it ends up exhibiting the same time anomalies such as sudden jumps, insufficient granularity, or insufficient accuracy. In this case, the promise of portability becomes a dangerous illusion.

The Ada 83 `Calendar` package attempts to provide a single-clock interface. In order to ensure that it can be implemented in most execution environments, very little is specified about `Calendar.Clock` and, as mentioned, the predominant practice among existing implementations is to treat `Calendar.Clock` as political time. The values are likely not to be monotonic, and the resolution may be rather coarse. In effect, `Calendar.Clock` cannot be relied upon for measurement of "physical time" in real-time applications.

A Two-Clock Model

For the Real-Time Annex, we considered adding requirements to `Calendar.Clock` so that it would satisfy real-time requirements. For example, it could be required to be monotonic and have at least one millisecond precision. This idea was rejected. One reason is that the requirement for monotonicity might conflict with existing practice and other (non-real-time) requirements for a standard clock that returns local political time. A second reason is that requiring fine precision for `Calendar.Clock` might prevent an implementation from using hardware calendar-clock circuits. Thus `Calendar.Clock` is essentially as in Ada 83.

In contrast, `Real_Time.Clock` is used for computations of physical parameters based on time, and scheduling of task execution to satisfy real-time constraints. The implementation must ensure that the value of the clock progresses monotonically, and that irregularities are strictly bounded. After the system starts, the clock is not allowed to be reset by an operator, the underlying operating system, or the run-time environment.

Of course, there is no requirement for an implementation to have multiple clocks internally. The implementation may simply provide two package interfaces to a single underlying (monotonic) clock. The capability of supporting clock adjustments and seasonal time changes for `Calendar.Clock` is not mandated by the language, so the values of the two clocks could be the same. Moreover, where the application requires `Calendar.Clock` to do things that are incompatible with the specification of `Real_Time.Clock`, such as seasonal time changes and clock adjustments, the effect may be accomplished by computing a transformation of the value of `Real_Time.Clock`. It is in fact recommended that both `Calendar.Clock` and `Real_Time.Clock` be transformations of the same underlying timebase.

The suggestion was made that a way might be provided for the application to modify the rate of the clock, so that the application could do clock synchronization, and do it in a way that would not compromise monotonicity. However, such a requirement would be applicable to only a small subset of applications, and the cost of providing the capability would be unwelcome for applications not needing it. In fact, for most existing general purpose processors, such a facility is not provided in the hardware, and providing it in software would introduce significant overhead in the clock driver. Alternatively, this capability, as well as the capability to do other low-level clock functions, is better provided by expecting the implementation to export the interface to its low-level clock driver in these systems, allowing it to be replaced by applications with special clock requirements.

D.8.3 Clock Accuracy Requirements

The average clock tick given by the constant `Real_Time.Tick` is specified as not being larger than one millisecond. This number is conservative in the direction of not imposing extreme requirements on implementors, and seems adequate for the task scheduling needs of many real-time applications. Finer clock resolution is recommended.

D.8.4 Relationship to Delays

The requirement that `Real_Time.Clock` be consistent with the effect of delay statements may be problematic for some implementations, but the conceptual consistency is seen as outweighing the implementation difficulty. One problem is that the range of times measurable directly by the available counter-timer circuit may be very narrow. In this case, the clock may need to be implemented in two parts. The low-order part may be decremented by every tick of the hardware timer, and the high-order part may be incremented by an interrupt handler that is driven by underflow of the timer. Another possible problem is that a separate timer circuit may be used for delays. It is desirable to utilize one timer to implement the real-time clock, using the simple treatment of timer underflow explained above, and to program another timer to generate an interrupt at the next point a delay is due to expire. However, in this case, since the delay timer is used only to express offsets from the clock, any difference between the two timers may not be noticeable.

D.8.5 Representation of Duration, Time_Span, and Real_Time.Time

The `Time_Span` type is introduced to allow more precise representation of durations. A new type is introduced because the need for upward compatibility rules out changes to the range requirement for `Standard.Duration`.

Requirements and Representation

Lack of sufficient precision is one of the issues with the `Calendar` package and delay statements in Ada 83. The `Duration` type is required to be able to represent a full day, in the positive or negative direction. The hardware arithmetic on many machines today is limited to 32 bits. If `Duration` values are represented with 32 bits, `then Duration'Small` cannot be smaller than `2.0**(-14)` seconds. This is coarser than the resolution of timer circuits. If the resolution of the timer is not exactly equal to an integer multiple (or divisor) of `Duration'Small`, additional precision can be lost in conversion. For example, suppose the clock is implemented using a timer with microsecond resolution, and the difference of two clock values is 100 microseconds. If `Duration'Small is 2.0**(-14)`, the nearest `Duration` value to 100 microseconds is `2*Duration'Small`, or about 122 microseconds. Conversion to `Duration` in this example has introduced an error of 22 percent!

The required minimum range and precision of `Time_Span` represent a compromise, given the assumption that the value should be representable in 32 bits. Originally, we required that `Time_Span_Unit` be equivalent to at most one microsecond and the range, equivalent to at least

`-2.0 .. 2.0` seconds. These requirements would still allow for a nanosecond representation in 32 bits (for example, the real-time extensions to POSIX specify nanosecond precision for timers). On the other hand, it would allow a range of up to an hour (with one microsecond precision). However, reviewers have commented that a portable range of `-2.0 .. 2.0` is too small to be useful. We have changed the requirements so that a minimum resolution of twenty microseconds, and a minimum range of − one hour are mandated. This compromise still allows for "spare" bits in each direction, so that implementations, using 32 bits, can still have some flexibility in responding to stricter application needs without violating the range or precision requirements. Of course, this freedom sacrifices portability for users who require a greater range or finer precision than these minimum specifications. It is expected that in many implementations, the representation of `Time_Span` will use 64 bits (just as for the type `Time`). Since this type is private, such an implementation approach will not require 64-bit arithmetic in general.

Since these requirements are based on a 32-bit machine architecture, for machines with a smaller word size, we have provided an escape clause in the form of an implementation permission. For example, some machines have only 24-bit words with no easy way to manipulate double-words. If we want to maintain the model of one word for `Time_Span` and two for `Time`, we must relax the range/accuracy requirements. On the other hand, a 16-bit machine such as the 1750A, which has built-in double-word operations, can easily use one double-word for `Time_Span` and two double-words for `Time`, and thus meet the requirements.

The possibility was also considered of having `Time_Span` as a visible integer type, which could be a count of ticks. This is appealing, since clocks count time in ticks, and arithmetic on real numbers tends to introduce unnecessary loss of accuracy. Under the Ada 83 rules, the multiplication and division operations on fixed point types require much greater precision than for integer types of the same size. Moreover, real-time systems often involve computations in which time is viewed as cyclic. Such computations are naturally expressed in terms of the integer division and rem operations, rather than fixed point operations. This idea was discarded because there was a potential for confusion arising from different units for logically similar types. For example, the assignment statements in

```
T: Time_Span;
...
T := T + 3        -- add 3 ticks to T
...
T := T + 3.0;     -- add three seconds duration to T
```

would have a vastly different meaning and yet both be allowed because both relevant overloadings of the `"+"` operator would exist.

The concept of a `Time_Unit` is introduced to ensure that the choice of representations for `Time` and `Time_Span` do not cause loss of information in time arithmetic. That is, the value obtained by subtracting two `Time` values should be exactly representable as a `Time_Span`, and adding a `Time_Span` value to a `Time` value should yield an exact `Time` value. This is the origin of the requirement that `Time_Span_Unit` be equal to `Time_Unit`.

Fixed point Issues

An alternative considered was to replace both `Time` and `Time_Span` by a single (64-bit) fixed point type. This would have simplified the interface and allowed a full range of user needs to be met. However, we concluded that supporting fixed point arithmetic on 64 bits would have been an unreasonable requirement to impose on all real-time implementations. Moreover, users who do not require extreme range or precision would have suffered from the overhead of arithmetic operations on objects of such a type. Finally, the requirements for accuracy and determinism on these types would have disturbed the general model of fixed point types in the core too much. Some of the needed changes would have been in direct conflict to the changes needed to support decimal types. Also, they would have been upward incompatible and too much of an implementation burden. Below, we provide more details about this alternative and related issues.

D.8.6 Arithmetic and Relational Operators

In Ada 83 [RM83 9.6(5)], nothing is specified about the semantics of arithmetic and relational operators on times and durations except that the operations "have the conventional meaning". One of the objectives of this Annex is to give a more precise specification. Several approaches were considered. One of these is to specify a representation for `Time`, and then define the effects of the operations in terms of the representation. Possibilities considered included: a two-part record, analogous to the POSIX "timespec" type (a two-part record, consisting of a signed integer count of seconds and an unsigned integer count of nanoseconds); a very long integer type; and a very long fixed point type. This approach was rejected on the grounds that it would not allow the natural implementation for a wide enough variety of machines and operating systems. On the assumption that `Time` must be a private type, the possibility of providing an axiomatic specification of time arithmetic was considered. This approach was rejected on the grounds that it is inconsistent with the style of the rest of the Ada language definition. The present approach draws on analogy to the definition of arithmetic on integer types. In addition, for the conversion functions, rounding is specified (away from zero) to ensure deterministic results.

Another possibility considered was of specifying that the time values are unsigned integers. As such, there is no overflow or underflow, and arithmetic is modular. One unfortunate aspect of using modular arithmetic for time is that the relational operations must be used with great care. For example, on a 12-hour clock it is not possible to say whether eleven o'clock is earlier or later than one o'clock, without further information. Because of this potential for confusion, the idea of arithmetic on time values being modular was dropped. This means that the `Time` type cannot be practically represented in 32 bits.

If `Time` is going to take 64 bits, there is no problem representing a large range. A 32-bit signed count of seconds can represent a range of about 136 years. The requirement for a range of 50 years has been chosen because it is well within this range, and appears more than adequate to handle the continuous running time of most real-time systems.

The operations `Nanoseconds`, `Microseconds`, and `Milliseconds` construct values of the type `Time_Span`. We considered having constants for one nanosecond, one microsecond, etc. However, the possibility that such real time values might not be representable accurately as `Time_Span` values, when using the constants to convert multiples of these values, leads to the danger of introducing cumulative errors. For example, if one wants to have a value of `Time_Span` equal to five milliseconds, calling `Milliseconds(5)` will return a more accurate result than doing `5*One_Millisecond`, where `One_Millisecond` is a constant of `Time_Span` representing one millisecond. Using `Milliseconds`, one can convert up to almost 25 days worth of milliseconds (assuming a 32-bit implementation of `Integer`). This range seems large enough for this purpose, so a function that takes seconds as a parameter is not provided.

D.8.7 Other Issues

In order to allow direct mapping of `Real_Time.Time` onto the most natural time reference that is available for a particular implementation, it is not required that there be any fixed correspondence between time values and specific real-time intervals. For example, `Real_Time.Time` could be a simple count of ticks since the clock was started. Given a fixed size representation of time values, this gives the widest possible range of values in the direction of interest, which is forward from the time of system start-up. It is also easy to implement, since there is no requirement for synchronization to obtain the initial clock value.

In a model with this degree of implementation freedom, it is difficult to specify meaningful counterparts of `Calendar.Split` and `Calendar.Time_Of`. In this context, `Split` and

Time_Of are likely to be used as a communication means to the outside world (since both Time and Time_Span are private). Examples include constructing a (local) time value from information read from a network, and logging event times in a readable format. Two possible approaches were considered.

One approach was to provide functions for conversion between Real_Time.Time and Calendar.Time. The Split and Time_Of operations on Calendar.Time could then be used. The other approach was to define Time_Of and Split as operations that would convert a Time value into a seconds value and Duration part, or construct a Time value from these values. The seconds value would then be interpreted as an extended duration since clock start-up. Both of these approaches could be implemented, within some degree of tolerance for error, if the implementation reads both Real_Time.Clock and Calendar.Clock at the time of system start-up to establish a common reference point.

The second approach, with a slight variation, was chosen for two reasons. First, it does not seem appropriate to require applications to include the package Calendar, just for this I/O purpose, if it is not needed otherwise (as is often the case). Second, as was discussed above, the package Calendar allows for certain implementation-defined behavior; it is not clear that the operations of this package will always be capable of serving as a transparent filter, one that provides the appropriate range and accuracy needed by the Real_Time.Time type representation.

Accordingly, an integer type, Seconds_Count, is introduced. It represents the elapsed time from the epoch (the origin point) in seconds. (Since the epoch of the time is not specified by this Annex, the meaning of the Seconds parameter has to be interpreted based on implementation and application conventions.) A seconds representation was chosen based on range considerations. Even a 32 bit representation is enough to hold 50 years. Seconds_Count is a signed integer since the Annex does not specify that Time_First equals the epoch. In fact, it is legal to have the epoch defined somewhere in the future, and have Time values as negative offsets from that point. Hence, Seconds_Count should be able to hold both positive and negative values.

For the fraction part, we had originally chosen the type Duration (as opposed to Time_Span). This was done in light of the fact that the primary purpose of the Split and Time_Of operation is communication with the outside world. A visible and printable type is much more convenient in this case. However, some reviewers commented that by doing so we introduce the possibility of an error "at the source", and that Time_Span should be used instead of Duration as the parameter for these operations. Since there exist other conversion routines that return a Duration value, and since the suggestion seemed to provide more flexibility, it was accepted.

Metrics

The intent of the upper bounds on clock ticks and clock jumps is to quantify the maximum fine-grain clock variation that can be expected.

The upper bound on clock drift rate is intended to provide an estimate of the long-term accuracy of the clock.

The upper bound on the execution time of a call to the Clock function is intended to expose implementations where reading the clock is extremely time-consuming. This might be the case, for example, where the clock function involves an operating system call, which involves context switches in and out from the operating system kernel.

The reason for the metric on time arithmetic is to expose extremely inefficient time representations. For example, this is likely to expose the difference between an implementation based on a record containing years, months, days, etc. and an implementation based on a 64-bit count of clock ticks.

Not all of these metrics are intended to be testable by pure Ada benchmark programs, such as the PIWG performance tests. That measurement technique is inherently limited, especially by the accuracy and precision of the software clock. Instead, it is intended that an external timing instrument, such as a logic analyzer, plus some knowledge of the implementation, may be needed to obtain the values of some metrics. In particular, this applies to measurements of the accuracy of the clock itself. Benchmark programs that rely on the internal clock for a time reference are inherently incapable of measuring the behavior of the clock itself. Moreover, for fine measurements such programs must settle for average execution times, since they must perform many iterations before they can accumulate enough execution time that is measurable on the internal clock. Thus, benchmarks are intrinsically incapable of deriving worst-case bounds for short execution times.

D.9 Delay Accuracy

Real-time applications require that a task of sufficiently high priority be able to delay itself for a period of time with the assurance that it will resume execution immediately when the delay expires — i.e. that the duration of the interval between the start of the delay and the time the task resumes execution must be equal to the requested duration, within a predictable tolerance.

[RM95 9.6] only requires that execution of the task that executes the delay be blocked for at least the duration specified. It is not in general possible to require an upper bound on the duration of the execution of any statement, due to possible interleaved operations of other tasks on the same processor. However, it is both possible and necessary to have an upper bound on the duration of the interval between the start of a delay and the time the expiration of the delay is detected. It is also possible to guarantee that if the task whose delay has expired has higher priority than all the other tasks it will resume execution as soon as the expiration of the delay is detected.

This section of the Annex tightens the core requirements on the implementation of delays, and requires documentation of implementation characteristics. These tighter semantics also apply to uses of delay statements within select statements. These tighter semantics will permit better prediction of application timing behavior.

Coordination with Real_Time.Clock

An important reason for a language to provide a standard clock is to present a view of time that is coordinated with the implementation of the delay statement. Without such coordination, the utility of both delays and the clock is significantly diminished.

The measurement of delays relative to a time reference that may be reset or adjusted (i.e. the time-of-day/calendar clock) is unacceptable, due to possible anomalies. In general, it may be necessary to adjust the calendar clock, for such things as leap-seconds or time zones. Maintaining a relationship between the actual delay duration and the time, relative to such a non-continuous clock, would make delays useless for most hard real-time applications, and would impose extra complexity on the delay implementation.

The specific requirements in this section for coordination with `Real_Time.Clock` are minimal, since a delay statement is only required to delay a task for "at least" a specified time. However, taken together, the metrics on delay accuracy and clock accuracy permit a user to determine more precisely how well coordinated delays are with `Real_Time.Clock`.

We also considered specifying a relationship between the clock resolution and the delay resolution. It is not reasonable to require that the delay resolution be at least as fine as that of the clock itself. The internal resolution can have very fine granularity if it is implemented via a hardware timer, much finer than the overhead of setting up a delay or reading the clock. If a periodic timer-generated interrupt is used to check for delay expirations, the interval between interrupts must be kept long enough to get useful work done; this limits delay granularity. If delay expirations are implemented via a programmed-interval timer, delay accuracy is limited by the overhead of receiving an interrupt and reprogramming the timer. It is possible to achieve finer granularity (without blocking) via execution of a timed sequence of "idle" instructions. This may provide delay resolution below the level of granularity achievable by a timer, provided the task is able to execute without preemption. Otherwise, if the task is preempted, it may delay longer than desired. To remain accurate in the face of preemption, the task could loop, reading the clock and comparing the clock value against the desired wake-up time; in this case, the precision is limited by the time it takes to execute an iteration of this loop. Of course, such busy-waiting techniques would not be sensible where delays are used within select statements, if the task is waiting for a rendezvous with a task that must execute on the same processor. It is not reasonable to require that the clock resolution be at least as fine as the delay resolution, either, since this could rule out the high-resolution delay implementation techniques described above.

Uniform Behavior Near Zero

A problem with timed entry calls was pointed out by the Third International Workshop on Real-Time Ada Issues [Baker 89]. Suppose the requested delay duration is a variable, and consider the effect of the timed entry call as the requested duration approaches zero from above. For large positive requests, an attempt will be made to perform a rendezvous. For small enough positive requests, an implementation is permitted to not make any attempt to rendezvous, on the presumption that simply determining whether a rendezvous is possible will take longer than the requested delay. The effect is that for small positive requests there will certainly be no rendezvous, and the total execution time of the timed entry call will be short. Then, as the requested delay approaches zero, the semantics change abruptly, back to what they would be for large positive requests (this is because of the conditional entry call semantics as specified in [RM95 9.7.2, 9.7.3]). The implementation must check whether a rendezvous is possible. This may take a long time. There is again a possibility of rendezvous, and the execution time of the timed call will be longer than it is for requests with small positive delays. An implementation that conforms to this Annex should avoid this anomalous behavior for small positive values, by always attempting to make a rendezvous (even if the requested duration is very short).

Similar issues come up with timed entry calls using the absolute form of the delay statement, and for delay alternatives in selective accept and asynchronous select statements. However, for asynchronous select statements, the required behavior is modelled after the case where an entry call replaces the delay statement. In this situation, if the entry call can proceed immediately, the abortable part never starts. Similarly, when the delay amount is zero, the alternative is chosen, and the abortable part does not start.

An Alternative Proposal

The Third International Workshop on Real-Time Ada Issues proposed a more detailed accuracy model for delays [Baker 89]. One possibility that we considered was to incorporate this approach into the implementation requirements. This proposal has not been adopted, because it is expressed in terms of events in the implementation that are not directly visible to the user, and it was believed to be too complex.

Documentation Requirements

The implementation is required to document the thresholds that it uses to determine whether a delay statement will result in the blocking of the task.

Metrics

The specifications given here are intended to allow enough flexibility that they can be applied to a variety of different implementation techniques.

The upper bound on the execution time of a relative delay statement with zero duration, and the execution time of an absolute delay whose wake-up time has already arrived, are intended to give the user an approximate idea of the minimum execution time overhead of the statement, excluding blocking.

The upper bounds on the lateness of delay statements are intended to give the user an idea of the accuracy with which delays are implemented. As with other upper bounds, its mere existence is actually more important than the precise value of the bound.

It is understood that these metrics will not expose the full implementation behavior. For example, if busy-wait delays are used for short durations, the granularity there may be much finer than further up the scale. The present metric ignores this issue. Likewise, if the hardware timer has limited range, a timer-task might be used for delays outside this range. Thus, there might be another shift in granularity farther out. The metrics chosen here do not require the implementor to expose such details. However, the implementor is free to provide more detailed information, by expressing the bound as a function of the requested delay.

D.10 Synchronous Task Control

During the 9X revision, the term *suspension* was replaced with *blocking* since it was considered to better describe the actual state (i.e. waiting for something to happen — being blocked as opposed to just being suspended). We recognize that traditionally suspend and resume were the common terms used when discussing these primitives. In this and the following section, we use the term *blocked* when referring to the "official" Ada state, and the term *suspended* when referring to the generic capability.

An important goal for Ada 95 was to allow protected objects for a simple suspend/resume mechanism that in turn could be used to build higher-level synchronization primitives. Here, by suspend, we mean the ability to block only the calling task itself, not another (for the latter, see the next section). Even for such a simple mechanism, some guarantees have to be made. This is commonly known as the *two-stage suspend* problem. (Strictly speaking, this name refers more to the solution, rather to the problem itself.) The problem that needs to be solved can be briefly described as follows. A task may want to block itself, after it has checked some data structure, and found that a particular system state is not present yet. The data structure is used by other tasks as well. One of these tasks will eventually set the data structure to the appropriate state and will resume the blocked task. Therefore, this data structure must be protected from concurrent access, i.e. a lock is needed. This in turn leads to the requirement that a task will be able to atomically release the lock and block itself. If it first releases the lock, the state might change just before it is about to be blocked (for example, the desired state may now be present, but there will be no way to detect it). On the other hand, if the task is blocked while still holding the lock, another task will not be able to access the shared data structure, and to record the new state — a deadlock.

If the state that is being waited upon can be easily expressed using a protected entry barrier expression, then such functionality already exists in the language. However, this is not always the case. When user-defined schedulers or servers are being implemented, it is often much more natural to separate the blocked state (and the corresponding operations) from the actual reason the task is waiting (it might be waiting for multiple events).

There are several approaches to solve this problem. They all depend on the kinds of primitives the underlying system provides.

1 Allow the calling task to suspend while it still holds the lock. When the task is suspended, it loses the lock, and has to reacquire it when it wakes up.

2 Register a *pending suspension* request while still holding the lock. After the task releases the lock (presumably soon after registering the request), it will be suspended.

3 Atomically clear a bit in the user space marking the suspension intention, and then suspend waiting for the bit to be set. A task wishing to resume the suspended task does so by atomically setting the bit. If the protocol of manipulating this bit is well-coordinated, such a technique can work safely and efficiently.

It is beyond the scope of this discussion to analyze the trade-offs of the approaches described above. For Ada 95, we have chosen the third approach mainly for its simplicity and the fact that it does not require any changes to the semantics of protected types and does not have complex interactions with other existing semantics. Here, the two-stage suspend means that first the task announces its intention to suspend itself, and then it actually does so. Between these two operations, the task is logically suspended as viewed by other tasks in the system, and so they may reliably resume it even before the actual suspension is done. For example, it would be wrong for the suspending task to clear the bit again without first checking its state to ensure that no other task has resumed it in the meantime. Failing to do so will effectively result in losing the resume operation.

Originally, we proposed to express the needed functionality as visible operations of a language-defined protected type. The abstraction presented by a simple protected type with `Set_True` and `Set_False` operations and a `Suspend_Until_True` entry, in addition to one boolean flag, seemed appropriate. Having this type known to the implementation would ensure that optimization was straightforward.

We rejected this idea for two reasons: a procedural interface enables the implementation to choose the most efficient technique by isolating this feature from the general support of the protected types. Second, by not having a visible protected entry for the `Suspend_Until_True` operation, the user is not able to use it in a select statement. While this may be considered as a loss of functionality, it has not been demonstrated that such functionality (timed, conditional, and asynchronous waits) is needed with such a low-level primitive. Not having to support the various kinds of select statements allows a much simpler, and hence, more efficient implementation.

The chosen solution is thus to make the suspension object of a private type with the operations described above (that is, `Set_True`, `Set_False`, and `Suspend_Until_True`). In addition, we provide a function `Current_State` to query the state of the object. This function should be used with care since the state may change asynchronously, and in particular, immediately after the function returns. We considered providing additional operations, to atomically change the state of the object and to return its previous state. We did not provide these operations since they really do not belong to this abstraction and we could not find a practical use for them; they were unreliable and they required an extra locking mechanism inside the implementation. This locking would be required when `Set_False` and `Set_True` (both with a return parameter) were called at the same time.

A suspension object can be viewed as a private binary semaphore in that it can be assumed to belong to one task only. This assumption is not enforced by the language, but a check is provided that only one task may wait on such an object at any point in time, `Program_Error` being raised if it fails. This rule makes it unnecessary to maintain a queue — a major saving in run-time cost.

A suspension object (or a pointer to it) can be passed to other components, thus indirectly maintaining the identity of the task that needs to be resumed when a certain state becomes true.

A typical example of the use of suspension objects is as follows

```
-- Assume that the protected object state
-- contains just a simple (protected) indication of the state;
-- the rest is elsewhere.

use Ada.Synchronous_Task_Control;
type Token is access all Suspension_Object;
protected State is
   procedure Check(T : in Token; Result : out Boolean);
   procedure Set(D : in Some_Data_Structure);
private
```

```
      Data : Some_State_Structure;
      Waiting_Task : Token;
   end State;

   protected body State is
      procedure Check(T : in Token; Result : out Boolean) is
      begin
         if Data = Desired_State then
            Result := True;
         else
            -- Mark intention to suspend
            Set_False(T.all);
            Waiting_Task := T;
            Result := False;
         end if;
      end Check;

      procedure Set(D : in Some_Data_Structure) is
      begin
         Data := D;
         if Data = Desired_State then
            if Waiting_Task /= null then
               -- Resume waiting task
               Set_True(Waiting_Task.all);
            end if;
         end if;
      end Set;
   end State;

   -- Task wishing to suspend
   task body T1 is
      SO : aliased Suspension_Object;
      In_Desired_State : Boolean;
   begin
      State.Check(SO'Unchecked_Access, In_Desired_State);
      if In_Desired_State then
         process-data
      else
         Suspend_Until_True(SO);      -- suspend
      end if;
      ...
   end T1;

   -- Another task detects that the waiting task needs
   -- to be resumed
   task body T2 is
      Data : Some_Data_Structure;
   begin
      State.Set(Data);
   end T2;
```

When `Check` is called by `T1`, the state is checked. If it is possible to continue, `T1` does so and processes the data. Otherwise, `T1` suspends itself until the object becomes true. When `T2` updates a new state, it checks to see if the updated state is a desired one. If it is, and a task is waiting, it is resumed (`Set_True`). The new state is saved, so when `T1` checks again, it will not have to be suspended. The important thing to remember is that it makes no difference whether the `Set_True` is called before or after the `Suspend_Until_True`. Since the semantics of suspension objects are defined to be persistent in the sense that there is a bit to keep the state, the suspending task will always notice the resume request.

D.11 Asynchronous Task Control

An important facility for some real-time aplications is a very low-level, simple, and efficient capability to suspend the execution of another task (and resume it later).

The core part of Ada 95 intentionally omitted this capability because of the well-known problems with integrating such a feature into a multi-tasking environment. The asynchronous transfer of control is the feature that comes closest to this requirement, but it is not the full answer; it requires cooperation of the "to-be-suspended" task, and does not allow the continuation of the affected task from exactly the same point where it was interrupted. There are very good reasons for these limitations in the general case. Suspending a task asynchronously at an arbitrary point is likely to leave the system state in an inconsistent state. This state would then become visible to the remaining tasks in the system. In addition, the interaction of such suspension with the other blocking primitives of the language is quite problematic (particularly, when priority scheduling is in effect).

In practice, two choices exist. One is to define the complete semantic model of such a feature and how it interacts with the rest of the language. Such a model would then require additions to the core and was believed to be very complex to understand and implement, especially for those users that do not need this capability. The other option is to leave all these interactions as implementation-defined. This is obviously undesirable since many of the benefits of standardizing such a capability would be lost. In addition, using such features is likely to move the program into the "erroneous zone", since the semantic model of tasking would not apply. Finally, and probably due to the above, experience with such primitives has proven in the past to be quite error-prone.

However, for a certain class of applications, such a capability is considered essential. These applications can be characterized as small, time-critical, and often safety-critical. They usually do not use the full power of the language, especially its tasking model. For certification reasons, as much as possible of the code needs to be visible in the actual program as opposed to be "hidden" inside the run-time system support supplied by the vendor. So even though this capability by itself may be considered unsafe, using it on top of a very simple run-time system, and applying strict guidelines, can make a system easier to certify. A final argument in favor of adding such a capability is that within certain application domains, this paradigm is well-understood, has been heavily used in the past, and is known to be implementable efficiently. Note that the issue of feature interaction with the rest of the tasking primitives is less of a concern here, since most of these primitives are not likely to be used by such an application.

Existing capabilities in the language and annexes allow a task to block itself until a specified state becomes true. This is not the same as a capability to asynchronously suspend another task. Because of this difference, the problems mentioned above, and issues concerning the best way to define such a feature in Ada, the straightforward approach of just defining a `Suspend_Other` primitive was rejected. Such an approach would necessitate introducing another task state ("suspended"), in addition to the existing states, and defining all the necessary interactions.

Instead, the approach taken by this Annex is based on the observation that a "suspend-other" capability is quite similar to the capability to lower a task's priority to a value that is so low as to effectively prevent the task from being dispatched. (In fact, using dynamic priorities is a known workaround to this problem, but it does not scale well to multiprocessors.)

The package `Asynchronous_Task_Control` introduces a conceptual *idle task* for each processor in the system, in addition to a priority level which is so low as to be below any other task in the system including the idle task. This level is also conceptual; it need not actually exist as a separate level in the ready queue. The `Hold` procedure is defined in terms of sources of priority inheritance. The general model of priority inheritance as defined in [RM95 D.3] states that the task's own base priority is always a source of priority inheritance. However, when the task is being held, its own base priority is no longer such a source, and instead the value of the special priority level becomes such a source. For reasons similar to those discussed in D.10, we do not want to stop the task's execution while it is inside a protected action. With this approach, a held task will still inherit the ceiling priority of the protected object in which it is executing, and will therefore continue to run until it leaves the protected action. When the task does not inherit any other priority, its active priority becomes lower than the conceptual task; therefore it does not run. The `Continue` operation simply changes the inheritance sources back to the default.

The benefit of this approach is that nothing else has to be said with respect to interactions with other tasking constructs. All the rules are ramifications of the above definitions and the general priority rules. (For a more detailed analysis of the various cases, see the notes in [RM95 D.11].) In this way, no additional mechanism is needed in the run-time system, and the feature can be implemented efficiently while still presenting a consistent and safe interface to the user. For implementation flexibility, nothing in this section really requires the use of dynamic priorities inside the implementation; priorities are used just to describe the semantic model. A straightforward implementation approach that uses traditional states is therefore possible.

D.12 Other Optimization and Determinism Rules

This section of the Annex describes various requirements for improving the response and determinism in a real-time system.

The maximum duration that interrupts are blocked by the implementation (in supporting the language features) must be bounded and documented. Clearly, this value is very important to the application for schedulability analysis. In addition, a real-time application often needs to interact with an external device at a certain frequency. If the implementation-induced interrupt blocking time is too long, such a device interface is not feasible.

Another requirement addresses the problem of the storage associated with terminated tasks. In a real-time system, tasks are often allocated using a library-level access type, and their storage is sometimes released only upon exit from the access type's scope. In this case, this will mean not until the partition as a whole terminates, which is clearly too late. Ada 83 did not require `Unchecked_Deallocation` of tasks to actually release the task's storage, and this is the motivation for the new requirement.

When a protected object does not have entries, it acts similarly to a simple lock (mutex) abstraction with no need for any overhead associated with checking barriers and servicing queues. It is expected that such protected objects will be used heavily by concurrent applications to achieve simple mutual exclusion. It is therefore important that implementations will recognize such cases, and avoid any unnecessary run-time costs. In general, performance can be neither legislated nor validated; the purpose of the requirement is to direct the attention of implementors to this important case. The corresponding metrics are provided for the purpose of exposing the degree to which such an optimization is carried out in a given implementation.

D.13 Requirements Summary

The requirements

R5.1-A(1) — Elapsed Time Measurement

R5.1-B(1) — Precise Periodic Execution

are met by the introduction of `Real_Time.Time` and the precise requirements on the delay statement.

The requirement

R5.2-A(1) — Alternative Scheduling Algorithms is generally addressed by the various pragmas such as `Task_Dispatching_Policy`, `Locking_Policy` and `Queueing_Policy` plus the facilities for priority control. The packages for synchronous and asynchronous task control provide facilities for special techniques.

E Distributed Systems

The Ada 95 model for programming distributed systems specifies a partition as the unit of distribution. A partition comprises an aggregation of library units that executes in a distributed target execution environment. Typically, each partition corresponds to a single execution site, and all its constituent units occupy the same logical address space. The principal interface between partitions is one or more package specifications. The semantic model specifies rules for partition composition, elaboration, execution, and interpartition communication. Support for the configuration of partitions to the target execution environment and its associated communication connectivity is not explicitly specified in the model.

The rationale for this model derives from the Ada 9X Requirements for Distributed Processing (see R8.1-A(1) and R8.2-A(1)); namely, that the language shall facilitate the distribution and dynamic reconfiguration of Ada applications across a homogeneous distributed architecture. These requirements are satisfied by a blend of implementor- and user-provided (or third-party) capabilities.

In addition, the following properties are considered essential to specifying a model for distributed program execution:

- The differences between developing a distributed versus a nondistributed system should be minimal. In particular, the same paradigms, rules for type safety, and interface consistency for a nondistributed system should apply to a distributed system. Furthermore, it must be possible to partition an Ada library for varying distributed configurations without recompilation.

- The implementation should be straightforward. In particular, the run-time system of each partition should be autonomous. In this way, robust type-safe distributed systems can be implemented using off-the-shelf Ada compilers that support the model, rather than depending upon custom adaptations of a compiler to a specific distributed environment.

- The partitioning should be separated from the details of the communications network architecture supporting the distributed system. Similarly, inter-partition communication should avoid specifying protocols more appropriately provided by an application or by standard layers of the network communications software.

- The model should facilitate programming fault-tolerant applications to the extent that an active partition failure should not cause the distributed program to fail. In particular, it should be possible to replace the services provided by a failed partition with those of a replacement partition.

- The model should be compatible with other standards that support open distributed applications.

The requirements and properties are satisfied in the Annex by specifying a simple, consistent, and systematic approach towards composing distributed systems based upon the partition concept. Partitions are specified before runtime, usually during or after the linking step. Programming the cooperation among partitions is achieved by library units defined to allow access to data and subprograms in different partitions. These library units are identified at compile-time by categorization pragmas. In this way, strong typing and unit consistency is maintained across a distributed system. Finally, separation of implementor and user responsibility is allowed by specifying a common interface to a partition communication subsystem (PCS) that performs message passing among partitions. The PCS is internally responsible for all routing decisions, low-level message protocols, etc. By separating the responsibilities, an implementation need not be aware of the specific network connectivity supporting the distributed system, while the communication subsystem need not be aware of the types of data being exchanged.

E.1 The Partition Model

An Ada 83 program corresponds to an Ada 95 active partition (see below); an Ada 95 program is defined in [RM95 10.2] as a set of one or more partitions. The description in the Core is kept purposefully non-specific to allow many different approaches to partitioning a distributed program, either statically or dynamically. In the Annex, certain minimal capabilities are specified to enhance portability of distributed systems across implementations that conform to these specifications.

This Annex develops the partitioning concept for distributed systems in terms of *active* and *passive* partitions. The library units comprising an active partition reside and execute upon the same processing node. In contrast, library units comprising a passive partition reside at a storage node that is accessible to the processing nodes of different active partitions that reference them. Library units comprising a passive partition are restricted to ensure that no remote access (such as for data) is possible and that no thread of control is needed (since no processing capabilities are available and no tasking runtime system exists in such a partition). Thus, a passive partition provides a straightforward abstraction for representing an address space that is shared among different processing nodes (execution sites).

It is implementation-defined (and must therefore be documented) whether or not more than one partition may be associated with one processing or storage node. The characteristics of these nodes are target dependent and are outside the scope of the Annex.

Similar to an Ada 83 program, each active partition is associated with an environment task that elaborates the library units comprising the partition. This environment task calls the main subprogram, if present, for execution and then awaits the termination of all tasks that depend upon the library units of the partition. Therefore, there is no substantive difference between an active partition and an Ada 83 program.

A partition is identified as either active or passive by the post-compilation (link-time) aggregation of library units. Post-compilation tools provide the necessary functionality for composing partitions, linking the library units of a partition, and for resolving the identities of other partitions. A passive partition may include only shared passive and pure library units.

By naming a shared passive library unit (which resides in a passive partition) in a context clause, the referencing unit gains access to data and code that may be shared with other partitions. Different active partitions (executing on separate nodes) may thus share protected data or call subprograms declared in such shared passive library units. An active partition can obtain mutually exclusive access to data in a shared partition package if the data is encapsulated in a protected object or is specified as atomic.

An active partition may call subprograms in other active partitions. Calls to subprograms in a different active partition are allowed only if the called subprogram is declared in a library unit with a `Remote_Call_Interface` pragma. Each active partition calling the subprogram must name the corresponding remote call interface (RCI) library unit in its context clause. So we might have

```
package A is                        -- in one active partition
   pragma Remote_Call_Interface(A);
   procedure P( ... );
   ...
end A;

--------------
```

```
with A;
package B is                           -- in another active partition
   ...
      A.P( ... );                      -- a remote call
   ...
end B;
```

When an active partition calls such a subprogram, the call is termed a remote procedure call (RPC). Stubs are inserted in the calling code and in the called code to perform the remote communication; these are termed the calling stub and receiving stub respectively. In addition, an asynchronous procedure call capability is provided to allow the caller and the remote subprogram to execute independently once the call has been sent to the remote partition.

The categorization of library units establishes potential interfaces through which the partitions of a distributed system may cooperate. In a distributed system where no remote subprogram calls or shared library units are required, e.g., all inter-partition data is exchanged through other communication facilities, library unit categorization is unnecessary. In such a case the multipartition program is similar to the multiprogramming approach allowed by Ada 83 (using a set of quite distinct programs).

The library unit categorization and link-time identification of partitions provides a flexible and straightforward approach for partitioning the library units of an Ada program library. Library units may be aggregated to form partitions exploiting the target execution environment for the distributed system, with the single stipulation that any given shared passive or RCI library unit may be assigned to only one partition. Different distributed configurations of the same target execution environment may then be supported by a single version of an Ada library. (A change to the configuration does not require recompilation of library units.) Library units are elaborated and executed within the context of the environment task associated with the active partition, and until they communicate with another partition, their execution proceeds independently (since all the library units in a passive partition must be preelaborated, the environment task in such a partition is purely conceptual).

The runtime system of each active partition is independent of all other runtime systems in a multi-partition program. This is achieved by first disallowing tasks and protected types with entries in the visible parts of the interface library units, and second, by declaring the library units `Calendar` and `Real_Time`, as well as the subtype `Priority`, as local to each active partition. In consequence, tasks (and hence entry queues) are not visible across partitions. This allows each active partition to manage its own tasking subsystem independently, avoiding such complexities as remote rendezvous, distributed time management, and distributed activation and termination management. (Protected objects without entries are allowed in passive partitions, since access to their data requires only a simple mutual-exclusion, a capability assumed to be present for a passive partition.)

Mechanisms to specify the allocation of partitions to the target execution environment are not included in the Annex; similarly, the dynamic creation and replication of partitions is not explicitly specified. These capabilities are deemed beyond the scope of the requirements. However, because partition replication is essential towards programming fault-tolerant applications, remote calls may be directed to different partitions using one of the two forms of dynamic binding, by dereferencing an access-to-subprogram object or access-to-class-wide tagged object. Thus, implementations that support the replication of partitions can allow a failed partition to be replaced transparently to other partitions.

In summary, this approach allows for flexible, link-time partitioning, with type-safety ensured at compile-time. The model separates categorization and partitioning from configuration and communication thus promoting compiler/linker independence from the target execution environment. The objective is to maintain the properties of a single Ada program for distributed execution with minimal additional semantic and implementation complexity. Fundamental to this objective is the ability to dynamically call remote subprograms.

E.2 Categorization of Library Units

Several library unit categorization pragmas exist. They are

```
pragma Shared_Passive( ... );
pragma Remote_Types( ... );
pragma Remote_Call_Interface( ... );
```

where in each case the optional parameter simply names the library unit. These pragmas identify library units used to access the types, data, and subprograms of other partitions. In other words, the library units that are associated with categorization pragmas provide the visible interface to the partitions to which they are assigned. These pragmas place specific restrictions upon the declarations that may appear in the visible part of the associated library units and the other library units that they may name in their context clauses. In addition, such library units are preelaborated.

The pragma `Pure`, which is defined in the core since it also relates to preelaboration, is also important for distribution and has the most severe restrictions.

The various categories form a hierarchy, in the order given above with `Pure` at the top. Each can only "with" units in its own or higher categories (although the bodies of the last two are not restricted). Thus a package marked as `Shared_Passive` can only with packages marked as `Shared_Passive` or `Pure`.

Restricting the kinds of declarations that may be present in such library units simplifies the semantic model and reduces the need for additional checking when the library unit is named in the context clause of another library unit. For example, by disallowing task declarations (and protected types with entries), we avoid the interaction among the run-time systems of different partitions that is required to support entry calls across partitions.

Pure library units [RM95 10.2.1] may be named in the context clauses of other interface library units. For example, a pure library unit may contain type declarations that are used in the formal parameter specifications of subprograms in RCI library units. To achieve remote dispatching, a library unit specified with pragma `Pure` must declare the corresponding dispatching operations. Such a library unit is replicated in all referencing partitions. The properties of a pure library unit allow it to be replicated consistently in any partition that references it, since it has no variable state that may alter its behavior.

When no categorization pragma is associated with a library unit, such a unit is considered normal; it may be included in multiple active partitions with no restrictions on its visible part. Unlike a pure library unit, replication of such a unit in different partitions does not necessarily maintain a consistent state. The state of the unit in each partition is independent.

E.2.1 Shared Passive Library Units

The rules for a shared passive library unit ensure that calling any of its subprograms from another partition cannot result in an implicit remote call, either directly or indirectly. Moreover, the restrictions eliminate the need for a run-time system (e.g., to support scheduling or real-time clocks) to be associated with a passive partition. Thus a passive partition corresponds to a logical address space that is common to all partitions that reference its constituent library units.

As mentioned earlier, a shared passive unit must be preelaborable and can only depend on pure and other shared passive units. There are also restrictions on access type declarations which ensure that it is not possible to create an access value referring back to an active partition.

E.2.2 Remote Types Library Units

Originally, this Annex provided only the `Shared_Passive` and `Remote_Call_Interface` pragmas (in addition to the core pragma `Pure`). However, this omitted an important functionality.

Often one needs to be able to pass access values among partitions. Usually, such access values have no meaning outside their original partition (since their designated object is still in that partition). Hence we generally disallow access types for remote subprograms' formal parameters. However, there are cases in which the access type has a user-defined meaning (such as a handle to a system-wide resource) that can be "understood" in other partitions as well. Since the implementation is not aware of such a meaning, the user must supply specific `Read` and `Write` attributes to allow the meaningful transfer of the information embodied in such access values. In addition, such a library unit often needs to be able to maintain specific (per-partition) state, to support such conversions. This is the main reason for introducing the `Remote_Types` categorization pragma. The restrictions enforced by this pragma are quite similar to those enforced by pragma `Pure`; a separate copy of a remote types package is placed in every partition that references it. Since a remote types library unit may be withed by a remote call interface, the types declared in the former may be used as formals of remote subprograms.

E.2.3 Remote Call Interface Library Units

For RCI library units the restrictions ensure that no remote accesses need be supported, other than remote procedure calls. These calls may be

- direct, through static binding,
- indirect, through a remote access to subprogram type,
- dispatching, through a remote access to class wide type.

Furthermore, the types of all formal parameters may be converted to and from a message stream type using the `Write` and `Read` attributes respectively [RM95 13.13.2]. This message stream type is the primary interface to the partition communication subsystem.

Child library units of an RCI library unit must be assigned to the same partition as the RCI library unit. As a consequence, visible child library units of an RCI library unit have the same restrictions as RCI library units. That is, the private part and the body of a child library unit have visibility to the private part of the parent. Thus a child library unit, unless included in the same partition as its parent, may make an unsupported remote access to its parent's private part. By constraining a child to the same partition, its visible part must be as restricted as the root RCI library unit.

The support for remote call interface library units is optional in the Annex, since RPC is not always the appropriate communication paradigm for a particular application. The other capabilities introduced by this Annex might still be useful in such a case.

Pragma All_Calls_Remote

For some applications, it is necessary that the partition communication subsystem get control on each remote procedure call. There are several motivations for such a requirement, including support for debugging (for example, isolating problems to either the PCS or to the generated code) and the need in some circumstances to have the PCS perform application-specific processing (e.g. supporting broadcasts) on each remote call. For such techniques to be feasible, users need to be assured that remote calls are never "optimized away". This can be assured by inserting

```
pragma All_Calls_Remote;
```

in the unit concerned.

Note that opportunities for such optimizations arise often, for example when the calling library unit and the called RCI library unit are assigned to the same active partition. In such cases, the linker can transform the remote call to a local call, thereby bypassing the stubs and the PCS. (In fact, such an optimization is extremely important in general, to allow the design of library units independent of their eventual location.) Similar optimization is possible (although probably not as straightforward) when multiple active partitions are configured on the same processing node.

When a call on a subprogram declared in the visible part of an RCI library unit (usually a remote call) is generated from either the body of that library unit or from one of its children, it is always guaranteed to be local (regardless of the specific configuration). This is because the Annex rules ensure that all corresponding units end up in the same partition. For this reason, the `All_Calls_Remote` pragma does not apply to such calls, and they remain local. Doing otherwise would constitute a change of the program semantics (forcing a local call to be remote), would introduce implementation difficulties in treating otherwise normal procedure calls as special, and would introduce semantic difficulties in ensuring that such a local-turned-remote call did not violate the privacy rules that guarantee that remote references are not possible.

E.3 Consistency of a Distributed System

Consistency is based on the concept of a version of a compilation unit. The exact meaning of version is necessarily implementation-defined, and might correspond to a compilation time stamp, or a closure over the source text revision stamps of all of the semantic dependences of the unit.

E.4 Remote Subprogram Calls

RCI library units allow communication among partitions of a distributed system based upon extending the well-known remote procedure call (RPC) paradigm. This is consistent with the ISO RPC Committee Draft that presents a proposed RPC Interaction Model (see subsequent section) and Communication Model for cooperating applications in a distributed system. Calls to remote partitions may be bound either statically or dynamically.

The task executing a synchronous remote call suspends until the call is completed. Remote calls are executed with at-most-once semantics (i.e., the called subprogram is executed at most once; if a successful response is returned, the called subprogram is executed exactly once). If an exception is raised in executing the body of a remote subprogram, the exception is propagated to the calling partition.

Unless the pragma `Asynchronous` (see below) is associated with a procedure (for a direct call) or an access type (for an indirect call), the semantics of a call to a remote subprogram are nearly identical to the semantics of the same subprogram called locally. This allows users to develop a distributed program where a subprogram call and the called subprogram may be in the same or different partitions. The location of the subprogram body, determined when the program is partitioned, only affects performance.

The exception `System.RPC.Communication_Error` may be raised by the PCS (the package `System.RPC` is the interface to the PCS). This exception allows the caller to provide a handler in response to the failure of a remote call as opposed to the result of executing the body of the remote subprogram; for example, if the partition containing the remote subprogram has become inaccessible or has terminated. This exception may be raised for both synchronous and asynchronous remote calls. For asynchronous calls, the exception is raised no later than when control would be returned normally to the caller; any failure after that point is invisible to the caller.

E.4.1 Pragma Asynchronous

An asynchronous form of interaction among partitions is provided by associating the pragma `Asynchronous` with a procedure accessible through an RCI library unit. Thus using the previous example we might write

```
package A is
   pragma Remote_Call_Interface(A);
   procedure P( ... );
   pragma Asynchronous(P);
   ...
end A;
```

When this pragma is present, a procedure call may return without awaiting the completion of the remote subprogram (the task in the calling partition is not suspended waiting the completion of the procedure). This extends the utility of the remote procedure call paradigm to exploit the underlying asynchronism that may be available through the PCS. As a consequence, synchronous and asynchronous interactions among partitions are maintained at a consistent level of abstraction; an agent task is not required to await the completion of a remote call when asynchronism is desired. Asynchronous procedure calls are necessarily restricted to procedures with all parameters of mode in (and of course a function cannot be asynchronous).

Unhandled exceptions raised while executing an asynchronous remote procedure are not propagated to the calling partition but simply lost. When the call and called procedure are in the same partition, the normal synchronous call semantics apply.

The use of asynchronous procedure calls, when combined with the capability to dynamically bind calls using remote access values, allows the programming of efficient communication paradigms. For example, an asynchronous procedure call may pass a remote access value designating a procedure (in the sending partition) to be called upon completion. In this way, the results of the asynchronous call may be returned in some application-specific way.

E.5 Post-Compilation Partitioning

Aggregating library units into partitions of a distributed system is done after the units have been compiled. This post-compilation approach entails rules for constructing active and passive partitions. These rules ensure that a distributed system is semantically consistent with a nondistributed system comprising the same library units. Moreover, the required implementation is within the capability of current post-compilation tools. Therefore, in order to allow the use of existing tools and to avoid constraining future tools, the Annex omits specifying a particular method for constructing partitions.

Each RCI library unit may only be assigned to a single active partition. Similarly, each shared passive library unit may only be assigned to a single passive partition. Following the assignment of a library unit to a partition, a value for the attribute `Partition_ID` is available that identifies the partition after it is elaborated. (This attribute corresponds to values of the type `Partition_ID` declared in `System.RPC`; see E.7. This library unit provides the interface to the PCS; however, it is not required that this unit be visible to the partition using the attribute and hence the attribute returns *universal_integer*.)

In order to construct a partition, all RCI and shared passive library units must be explicitly assigned to a partition. Consequently, when a partition is elaborated, the `Partition_ID` attribute for each RCI or shared passive library unit referenced by this partition has a known value. The construction is completed by including in a partition all the other units that are needed for execution.

An exception is that a shared passive library unit is included in one partition only. Similarly, the body of an RCI library unit is in one partition only; however the specification of an RCI library unit is included in each referencing partition (with the code for the body replaced by the calling stubs).

A library unit that is neither an RCI nor shared passive library unit may be included in more than one partition. Unlike a nondistributed system, a normal library unit does not have a consistent state across all partitions. For example, the package `Calendar` does not synchronize the value returned by the `Clock` function among all partitions that include the package.

A type declaration within the visible part of a library unit elaborated in multiple partitions yields matching types. For pure, RCI, and shared passive library units, this follows either from the rule requiring library unit preelaboration (RCI and shared passive) or the restrictions on their declarations. For normal library units, since non-tagged types are not visible across partitions, this matching is of little significance. However, a special check is performed when passing a parameter of a class-wide type to make sure that the tag identifies a specific type declared in a pure or shared passive library unit, or the visible part of a remote types or RCI library unit. Type extensions declared elsewhere (in the body of a remote types or RCI library unit, or anywhere in a normal library unit) might have a different structure in different partitions, because of dependence on partition-specific information. This check prevents passing parameters of such a type extension, to avoid erroneous execution due to a mismatch in representation between the sending and the receiving partition. An attempt to pass such a parameter to a remote subprogram will raise `Program_Error` at run-time. For example, consider the following declarations:

```
package Pure_Pkg is
   pragma Pure;
   type Root_Type is tagged ...
   ...
end Pure_Pkg;

with Pure_Pkg;
package RCI_Pkg is
   pragma Remote_Call_Interface;
   -- Class-wide operation
   procedure Cw_OP(Cw : in Pure_Pkg.Root_Type'Class);
end RCI_Pkg;

with Pure_Pkg;
package Normal_Pkg is
   ...
   type Specific_Type is new Pure_Pkg.Root_Type with
      record
         Vector : Vector_Type(1 .. Dynamic_Value);
      end record;
end Normal_Pkg;

with RCI_Pkg;
package body Normal_Pkg is
   Value : Specific_Type;
begin
   -- The following call will result in Program_Error
   -- when the subprogram body is executed remotely.
   RCI_Pkg.Cw_OP(Cw => Value);
end Normal_Pkg;
```

In the above example, if `Normal_Pkg` is included in a partition that is not assigned `RCI_Pkg`, then a call to `Cw_OP` will result in a remote call. When this call is executed in the remote partition, `Program_Error` is raised.

The following library units are a simple example of a distributed system that illustrate the post-compilation partitioning approach. In this particular example, the system uses mailboxes to exchange data among its partitions. Each partition determines the mailbox of its cooperating partitions by calling a subprogram specified in an RCI library unit.

The mailboxes for each partition are represented as protected types. Objects of the protected types are allocated in a shared passive library unit. RCI library units (instantiations of `Gen_Mbx_Pkg`) are included in active partitions, and they with the shared passive package `Ptr_Mbx_Pkg`. When an allocator for `Ptr_Safe_Mbox` is executed (on behalf of a library unit in another partition), the protected object is allocated in the passive partition, making it accessible to other partitions. Consequently, no remote access is required to use mailbox data. However, to access a mailbox of another partition, a remote subprogram call is required initially.

```
package Mbx_Pkg is
   pragma Pure;
   type Msg_Type is ...
   type Msg_Array is array (Positive range <>) of Msg_Type;
   type Key_Type is new Integer;

   protected type Safe_Mbx(Lock : Key_Type;
                           Size : Positive) is
      procedure Post(Note : in Msg_Type);
      -- Post a note in the mailbox
      procedure Read(Lock : in Key_Type;
                     Note : out Msg_Type);
      -- Read a note from the mailbox if caller has key
   private
      Key : Key_Type := Lock;
      Mbx : Msg_Array(1 .. Size);
   end Safe_Mbx;
end Mbx_Pkg;

with Mbx_Pkg;
package Ptr_Mbx_Pkg is
   pragma Shared_Passive;
   type Ptr_Safe_Mbx is access Mbx_Pkg.Safe_Mbx;
   -- All mailboxes are allocated in a passive partition and
   -- therefore remote access is not required.
end Ptr_Mbx_Pkg;

with Mbx_Pkg;
with Ptr_Mbx_Pkg;
generic
   Mbx_Size : Positive;
   Ptn_Lock : Mbx_Pkg.Key_Type;
package Gen_Mbx_Pkg is
   -- This package creates a mailbox and makes the
   -- access value designating it available through
   -- a remote subprogram call.
   pragma Remote_Call_Interface;
   function Use_Mbx return Ptr_Mbx_Pkg.Ptr_Safe_Mbx;
end Gen_Mbx_Pkg;

with Mbx_Pkg;
package body Gen_Mbx_Pkg is
   New_Mbx : Ptr_Mbx_Pkg.Ptr_Safe_Mbx :=
                  new Mbx_Pkg.Safe_Mbx(Ptn_Lock, Mbx_Size);
   -- A mailbox is created in the passive partition.
   -- The key to read from the mailbox is the elaborating
   -- partition's identity.
   function Use_Mbx return Ptr_Mbx_Pkg.Ptr_Safe_Mbx is
```

```
      -- The access value designating the created mailbox is
      -- made available to the calling unit.
      begin
         return New_Mbx;
      end Use_Mbx;
   end Gen_Mbx_Pkg;

   with Ptr_Mbx_Pkg, Gen_Mbx_Pkg;
   package RCI_1 is
      -- This package is the interface to a set of library units that
      -- is conveniently identified by the library unit Closure_1.
      pragma Remote_Call_Interface;
      package Use_Mbx_Pkg is new Gen_Mbx_Pkg(1_000, RCI_1'Partition_ID);
      function Use_Mbx return Ptr_Mbx_Pkg.Ptr_Safe_Mbx
         renames Use_Mbx_Pkg.Use_Mbx;
      -- All partitions include this remote subprogram in
      -- their interface.
      ...
   end RCI_1;

   with Ptr_Mbx_Pkg, Gen_Mbx_Pkg;
   package RCI_2 is
      -- This package is the interface to a set of library units that
      -- is conveniently identified by the library unit Closure_2.
      pragma Remote_Call_Interface;
      ...
   end RCI_2;

   with Ptr_Mbx_Pkg, Gen_Mbx_Pkg;
   package RCI_3 is
      -- This package is the interface to a set of library units that
      -- is conveniently identified by the library unit Closure_3.
      pragma Remote_Call_Interface;
      ...
   end RCI_3;

   with Closure_1;
      -- Names library units that execute locally.
   with RCI_2, RCI_3;
     -- Names RCI packages for interfacing to other
     -- partitions executing at different sites.
   package body RCI_1 is
      My_Mbx : Ptr_Mbx_Pkg.Ptr_Safe_Mbx := Use_Mbx_Pkg.Use_Mbx;
      Mbx_2,
      Mbx_3 : Ptr_Mbx_Pkg.Ptr_Safe_Mbx;
      ...
      -- Obtain access values to other partition mailboxes.
      -- For example
      Mbx_2 := RCI_2.Use_Mbx;
      Mbx_3 := RCI_3.Use_Mbx;
      ...
      My_Mbx.Read(RCI_1'Partition_ID, Next_Note);
      Mbx_2.Post(Next_Note);
      Mbx_3.Post(Next_Note);
      -- Read note in local mailbox and pass to other mailboxes.
      ...
   end RCI_1;
```

```
with Closure_2;
   -- Names library units that execute locally.
with RCI_1, RCI_3;
  -- Names RCI packages for interfacing to other
  -- partitions executing at different sites.
package body RCI_2 is
   ...
   -- Obtain access values to other partition mailboxes.
   -- For example
   Mbx_1 := RCI_1.Use_Mbx;
   Mbx_3 := RCI_3.Use_Mbx;
   ...
end RCI_2;

with Closure_3;
   -- Names library units that execute locally.
with RCI_1, RCI_2;
   -- Names RCI packages for interfacing to other
   -- partitions executing at different sites.
package body RCI_3 is
   ...
   -- Obtain access values to other partition mailboxes.
   -- For example
   Mbx_1 := RCI_1.Use_Mbx;
   Mbx_2 := RCI_2.Use_Mbx;
   ...
end RCI_3;
```

The following post-compilation partitioning support is implementation defined; the syntax is for illustration only. Several possible combinations for partitioning are presented. In each combination, the first partition specified is a passive partition where the mailboxes are allocated. This partition is accessible to other partitions by simply calling the protected operations of the mailbox.

The minimally distributed partitioning comprises two partitions; one passive partition and one active partition. All RCI library units in the application are assigned to a single active partition. There would be no remote calls executed as a result of this partitioning.

```
Partition(Ptn => 0, Assign => (Ptr_Mbx_Pkg))           -- passive
Partition(Ptn => 1, Assign => (RCI_1, RCI_2, RCI_3))   -- active
```

A more distributed version comprises three partitions. The RCI library units in the application are assigned to two active partitions.

```
Partition(Ptn => 0, Assign => (Ptr_Mbx_Pkg))           -- passive
Partition(Ptn => 1, Assign => (RCI_1))                 -- active
Partition(Ptn => 2, Assign => (RCI_2, RCI_3))          -- active
```

A fully distributed version comprises four partitions. The RCI library units in the application are assigned to three active partitions.

```
Partition(Ptn => 0, Assign => (Ptr_Mbx_Pkg))           -- passive
Partition(Ptn => 1, Assign => (RCI_1))                 -- active
Partition(Ptn => 2, Assign => (RCI_2))                 -- active
Partition(Ptn => 3, Assign => (RCI_3))                 -- active
```

Note that there is no need to mention the pure unit `Mbx_Pkg` because it can be replicated as necessary. Moreover, generic units do not have to be mentioned since it is only their instances that really exist.

E.5.1 Dynamically Bound Remote Subprogram Calls

In Ada 95, the core language supports dynamically bound subprogram calls. For example, a program may dereference an access-to-subprogram object and call the designated subprogram, or it may dispatch by dereferencing an access-to-class-wide type controlling operand. These two forms of dynamic binding are also allowed in distributed systems to support the programming of fault-tolerant applications and changes in communication topology. For example, through dynamically bound calls, a distributed program may reference subprograms in replicated partitions to safeguard against the failure of active partitions. In the event of a failure in a called active partition, the caller can simply redirect the call to a subprogram backup partition.

An advantage of these two forms of dynamic binding is that they relax the requirement for library units in the the calling partition to semantically depend on the library units containing the actual remote subprograms. Partitions need only name an RCI or Remote-Types library unit that includes the declaration of an appropriate general access type; objects of such types may contain remote access values.

A remote access value designating a subprogram allows naming a subprogram indirectly. The remote access value is restricted to designating subprograms declared in RCI library units. This ensures that the appropriate stubs for the designated subprograms exist in a receiving (server) partition. In order to pass remote access values designating subprograms among partitions, subprograms declared in RCI library units may specify formal parameters of access-to-subprogram types.

The remote access-to-class-wide type provides an alternative dynamic binding capability that facilitates encapsulating both data and operations. The remotely callable subprograms are specified as the primitive operations of a tagged limited type declared in a pure library unit. In an RCI or Remote-Types library unit, a general access type designating the class-wide type is declared; this declaration allows the corresponding primitive operations to be remote dispatching operations when overridden. Similar to the binding using access-to-subprogram types, library units in the calling partition need only include the RCI or Remote-Types library unit (that declares the access-to-class-wide type) in their context clause in order to dispatch to subprograms in library units included in other active partitions.

By restricting dereferencing of such remote access values to occur as part of a dispatching operation, there is no need to deal with remote addresses elsewhere. The existing model for dispatching operations corresponds quite closely to the dispatching model proposed for the linker-provided RPC-receiver procedure suggested in [RM95 E.4].

These dynamic binding capabilities are enhanced further when combined with a name server partition. Typically, the name server partition provides a central repository of remote access values. When a remote access value is made available to a client partition, the value can be dereferenced to execute a remote subprogram call. This avoids a link-time dependence on the requested service and achieves the dynamic binding typical of a client/server paradigm.

The following library units illustrate the use of access-to-class-wide types to implement a simple distributed system. The system comprises multiple client partitions, which are instantiations of `Client_Ptn`, a mailbox server partition named `Mbx_Server_Ptn`, and two partitions to access local and wide-area network mailboxes named `Lan_Mbx_Ptn` and `Wan_Mbx_Ptn` respectively. A client partition may communicate with other partitions in the distributed system through a mailbox that it is assigned by the mailbox server. It may post a message to its mailbox for delivery to another partition (based on the address in the message), or wait for a message to be delivered to its mailbox. A client may be connected either to the LAN or the WAN, but this is transparent to the application.

```
package Mbx_Pkg is
   pragma Pure;
   type Mail_Type is ...
   type Mbx_Type is abstract tagged limited private;
   procedure  Post(Mail : in Mail_Type;
                   Mbx  : access Mbx_Type) is abstract;
   procedure  Wait(Mail : out Mail_Type;
                   Mbx  : access Mbx_Type) is abstract;
private
   type Mbx_Type is abstract tagged limited null record;
end Mbx_Pkg;

with Mbx_Pkg; use Mbx_Pkg;
package Mbx_Server_Ptn is
   pragma Remote_Call_Interface;
   type Ptr_Mbx_Type is access all Mbx_Type'Class;
   function Rmt_Mbx return Ptr_Mbx_Type;
end Mbx_Server_Ptn;

with Mbx_Server_Ptn;
package Lan_Mbx_Ptn is
   pragma Remote_Call_Interface;
   function New_Mbx return Mbx_Server_Ptn.Ptr_Mbx_Type;
end Lan_Mbx_Ptn;

with Mbx_Server_Ptn;
package Wan_Mbx_Ptn is
   pragma Remote_Call_Interface;
   function New_Mbx return Mbx_Server_Ptn.Ptr_Mbx_Type;
end Wan_Mbx_Ptn;

with Mbx_Pkg;
package body Lan_Mbx_Ptn is
   type Lan_Mbx_Type is new Mbx_Pkg.Mbx_Type with ...;

   procedure Post(Mail : in Mail_Type;
                  Mbx  : access Lan_Mbx_Type);
   procedure Wait(Mail : out Mail_Type;
                  Mbx  : access Lan_Mbx_Type);
   ...

   function New_Mbx return Ptr_Mbx_Type is
   begin
      return new Lan_Mbx_Type;
   end New_Mbx;
end Lan_Mbx_Ptn;

with Mbx_Pkg;
package body Wan_Mbx_Ptn is ...

with Lan_Mbx_Ptn, Wan_Mbx_Ptn;
package body Mbx_Server_Ptn is
   function Rmt_Mbx return Ptr_Mbx_Type is
   begin
      if ... then
         return Lan_Mbx_Ptn.New_Mbx;
      elsif ... then
         return Wan_Mbx_Ptn.New_Mbx;
      else
```

```
         return null;
      end if;
   end Rmt_Mbx;
end Mbx_Server_Ptn;

   -- The client partitions do not need to with the specific
   -- LAN/WAN mailbox interface packages.
with Mbx_Pkg, Mbx_Server_Ptn, ...
use Mbx_Pkg, Mbx_Server_Ptn, ...
procedure Use_Mbx is
   Some_Mail : Mail_Type;
   This_Mbx  : Ptr_Mbx_Type := Rmt_Mbx;
   -- Get a mailbox pointer for this partition
begin
   ...
   Post(Some_Mail, This_Mbx);
   -- Dereferencing controlling operand This_Mbx
   -- causes remote call as part of Post's dispatching
   ...
   Wait(Some_Mail, This_Mbx);
   ...
end Use_Mbx;

generic
   ...
package Client_Ptn is
   pragma Remote_Call_Interface;
end Client_Ptn;

with Use_Mbx;
package body Client_Ptn is
begin
   ...
end Client_Ptn;

package Client_Ptn_1 is new Client_Ptn ...
...
package Client_Ptn_N is new Client_Ptn ...

   -- Post-compilation partitioning
Partition(Ptn => 0, Assign => (Mbx_Server_Ptn))
Partition(Ptn => 1, Assign => (Lan_Mbx_Ptn))
Partition(Ptn => 2, Assign => (Wan_Mbx_Ptn))
Partition(Ptn => 3, Assign => (Client_Ptn_1))
...
Partition(Ptn => N+2, Assign => (Client_Ptn_N))
```

In this next example, there is one controlling partition, and some number of worker partitions, in a pipeline configuration. The controller sends a job out to a worker partition, and the worker chooses either to perform the job, or if too busy, to pass it on to another worker partition. The results are returned back through the same chain of workers through which the original job was passed. Here is a diagram for the flow of messages:

```
             Job        Job        Job        Job
Controller ---->   W1 --->   W2 --->   W3 --->   W4 ...
           <---       <---       <---       <---
           Result     Result     Result     Result
```

The elaboration of each worker entails registering that worker with the controller and determining which other worker (if any) the job will be handed to when it is too busy to handle the job itself. When it receives a job from some other worker, it also receives a "return" address to which it should return results. The workers are defined as instances of a generic RCI library unit.

The first solution uses (remote) access-to-subprogram types to provide the dynamic binding between partitions. Two access-to-subprogram types are declared in the RCI library unit (`Controller`) that designate the procedures to perform the work and return the results. In addition, this library unit declares two procedures; one to register and dispense workers for the pipeline and one to receive the final results. An instantiation of a generic RCI library unit (`Worker`) declares the actual subprograms for each worker. The locations of these procedures are made available as remote access values; elaboration of `Worker` registers the `Receive_Work` procedure with the `Controller`.

```
package Controller is
   pragma Remote_Call_Interface;
   type Job_Type is ...;
   -- Representation of job to be done
   type Result_Type is ...;
   -- Representation of results
   type Return_Address is access procedure (Rslt : Result_Type);
   -- Return address for sending back results
   type Worker_Ptr is access
                procedure(Job : Job_Type; Ret : Return_Address);
   -- Pointer to next worker in chain
   procedure Register_Worker(Ptr  : Worker_Ptr;
                             Next : out Worker_Ptr);
   -- This procedure is called during elaboration
   -- to register a worker.  Upon return, Next contains
   -- a pointer to the next worker in the chain.
   procedure Give_Results(Rslt : Result_Type);
   -- This is the controller procedure which ultimately
   -- receives the result from a worker.
end Controller;

with Controller; use Controller;
generic
   -- Instantiated once for each worker
package Worker is
   pragma Remote_Call_Interface;
   procedure Do_Job(Job : Job_Type;
                    Ret : Return_Address);
   -- This procedure receives work from the controller or
   -- some other worker in the chain
   procedure Pass_Back_Results(Rslt : Result_Type);
   -- This procedure passes results back to the worker in the
   -- chain from which the most recent job was received.
end Worker;

package body Worker is
   Next_Worker : Worker_Ptr;
   -- Pointer to next worker in chain, if any
   Previous_Worker : Return_Address;
   -- Pointer to worker/controller who sent a job most recently

   procedure Do_Job(Job : Job_Type;
                    Ret : Return_Address) is
   -- This procedure receives work from the controller or
   -- some other worker in the chain
```

```
    begin
       Previous_Worker := Ret;
       -- Record return address for returning results
       if This_Worker_Too_Busy
           and then Next_Worker /= null then
          -- Forward job to next worker, if any, if
          -- this worker is too busy
          Next_Worker(Job, Pass_Back_Results'Access);
          -- Include this worker's pass-back-results procedure
          -- as the return address
       else
          declare
             Rslt : Result_Type;  -- The results to be produced
          begin
             Do The Work(Job, Rslt);
             Previous_Worker(Rslt);
          end;
       end if;
    end Do_Job;

    procedure Pass_Back_Results(Rslt : Result_Type) is
       -- This procedure passes results back to the worker in the
       -- chain from which the most recent job was received.
    begin
       -- Just pass the results on...
       Previous_Worker(Rslt);
    end Pass_Back_Results;
 begin
    -- Register this worker with the controller
    -- and obtain pointer to next worker in chain, if any
    Controller.Register_Worker(Do_Job'Access, Next_Worker);
 end Worker;

    -- Create multiple worker packages
 package W1_RCI is new Worker;
 ...
 package W9_RCI is new Worker;

    -- Post-Compilation Partitioning
    -- Create multiple worker partitions
 Partition(Ptn => 1, Assign => (W1_RCI))
 ...
 Partition(Ptn => 9, Assign => (W9_RCI))
    -- create controller partition
 Partition(Ptn => 0, Assign => (Controller))
```

The second solution uses (remote) access-to-class-wide types to provide the dynamic binding between partitions. A root tagged type is declared in a pure package `Common`. Two derivatives are created, one to represent the controller (`Controller_Type`), and one to represent a worker (`Real_Worker`). One object of `Controller_Type` is created, which will be designated by the return address sent to the first worker with a job. An object for each worker of the `Real_Worker` type is created, via a generic instantiation of the `One_Worker` generic. All of the data associated with a single worker is encapsulated in the `Real_Worker` type. The dispatching operations `Do_Job` and `Pass_Back_Results` use the pointer to the `Real_Worker` (the formal parameter `W`) to gain access to this worker-specific data.

The access type `Worker_Ptr` is used to designate a worker or a controller, and can be passed between partitions because it is a remote access type. Normal access types cannot be passed between partitions, since they generally contain partition-relative addresses.

```
package Common is
   -- This pure package defines the root tagged type
   -- used to represent a worker (and a controller)
   pragma Pure;
   type Job_Type is ...;
   -- Representation of Job to be done
   type Result_Type is ...;
   -- Representation of results
   type Worker_Type is abstract tagged limited private;
   -- Representation of a worker, or the controller
   procedure Do_Job(W : access Worker_Type;
                    Job : Job_Type;
                    Ret : access Worker_Type'Class) is abstract;
   -- Dispatching operation to do a job
   -- Ret may point to the controller
   procedure Pass_Back_Results(W : access Worker_Type;
                               Rslt : Result_Type) is abstract;
   -- Dispatching operation to pass back results
private
   ...
end Common;

with Common; use Common;
package Controller is
   pragma Remote_Call_Interface;
   type Worker_Ptr is access all Common.Worker_Type'Class;
   -- Remote access to a worker
   procedure Register_Worker(Ptr  : Worker_Ptr;
                             Next : out Worker_Ptr);
   -- This procedure is called during elaboration
   -- to register a worker.  Upon return, Next contains
   -- a pointer to the next worker in the chain.
end Controller;

package body Controller is
   First_Worker : Worker_Ptr := null;
   -- Current first worker in chain
   type Controller_Type is new Common.Worker_Type;
   -- A controller is a special kind of worker,
   -- it can receive results, but is never given a job
   The_Controller : Controller_Type;
   -- The tagged object representing the controller
   Controller_Is_Not_A_Worker : exception;
   procedure Do_Job(W : access Controller_Type;
                    Job : Job_Type;
                    Ret : access Worker_Type'Class) is
   -- Dispatching operation to do a job
   begin
      raise Controller_Is_Not_A_Worker;
      -- Controller never works (lazy pig)
   end Do_Job;

   procedure Pass_Back_Results(W : access Controller_Type;
                               Rslt : Result_Type) is
      -- Dispatching operation to receive final results
```

```
      begin
         Do Something With Result(Rslt);
      end Pass_Back_Results;

      procedure Register_Worker(Ptr : Worker_Ptr;
                                Next : out Worker_Ptr) is
         -- This procedure is called during elaboration
         -- to register a worker.  It receives back
         -- a pointer to the next worker in the chain.
      begin
         -- Link this worker into front of chain gang
         Next := First_Worker;
         First_Worker := Ptr;
      end Register_Worker;
   begin
      -- Once all workers have registered, Controller initiates
      -- the pipeline by dispatching on Do_Job with First_Worker
      -- as the controlling operand; Controller then awaits the
      -- results to be returned (this mechanism is not specified).
   end Controller;

   with Common; use Common;
   with Controller; use Controller;
   package Worker_Pkg is
      -- This package defines the Real_Worker type
      -- whose dispatching operations do all the
      -- "real" work of the system.
      -- Note: This package has no global data;
      -- All data is encapsulated in the Real_Worker type.
      type Real_Worker is new Common.Worker_Type with
         record
            Next : Worker_Ptr;
            -- Pointer to next worker in chain, if any
            Previous : Worker_Ptr;
            -- Pointer to worker/controller who sent
            -- us a job most recently
            ... -- other data associated with a worker
         end record;

      procedure Do_Job(W : access Real_Worker;
                       Job : Job_Type;
                       Ret : access Worker_Type'Class);
      -- Dispatching operation to do a job
      procedure Pass_Back_Results(W : access Real_Worker;
                                  Rslt : Result_Type);
      -- Dispatching operation to pass back results
   end Worker_Pkg;

   package body Worker_Pkg is
      procedure Do_Job(W : access Real_Worker;
                       Job : Job_Type;
                       Ret : access Worker_Type'Class) is
         -- Dispatching operation to do a job.
         -- This procedure receives work from the controller or
         -- some other worker in the chain.
      begin
         W.Previous := Worker_Ptr(Ret);
         -- Record return address for returning results
```

```
      if W.This_Worker_Too_Busy
            and then W.Next /= null then
         -- Forward job to next worker, if any, if
         -- this worker is too busy
         Common.Do_Job(W.Next, Job, W);
         -- now dispatch to appropriate Do_Job,
         -- include a pointer to this worker
         -- as the return address.
      else
         declare
            Rslt : Result_Type;  -- The results to be produced
         begin
            Do The Work(Job, Rslt);
            Common.Pass_Back_Results(W.Previous, Rslt);
            -- dispatch to pass back results
            -- to another worker or to the controller
         end;
      end if;
   end Do_Job;

   procedure Pass_Back_Results(W : access Real_Worker;
                               Rslt : Result_Type) is
      -- Dispatching operation to pass back results
      -- This procedure passes results back to the worker in the
      -- chain from which the most recent job was received.
   begin
      -- Pass the results to previous worker
      Common.Pass_Back_Results(W.Previous, Rslt);
   end Pass_Back_Results;
end Worker_Pkg;

generic
   -- Instantiated once for each worker
package One_Worker is
   pragma Remote_Call_Interface;
end One_Worker;

with Worker_Pkg;
with Controller;
package body One_Worker is
   The_Worker : Worker_Pkg.Real_Worker; -- The actual worker
begin
   -- Register this worker "object"
   Controller.Register_Worker(The_Worker'Access, The_Worker.Next);
end One_Worker;

-- Create multiple worker packages
package W1_RCI is new One_Worker;
...
package W9_RCI is new One_Worker;

-- Post-Compilation Partitioning
-- Create multiple worker partitions
Partition(Ptn => 1, Assign => (W1_RCI))
...
Partition(Ptn => 9, Assign => (W9_RCI))
-- create controller partition
Partition(Ptn => 0, Assign => (Controller))
```

E.6 Configuring a Distributed System

In the previous examples, post-partitioning has been illustrated in terms of the library units that comprise a partition. The configuration of partitions to nodes has been omitted since this is beyond the scope of the Annex. For example, whether partitions may share the same node is implementation defined. The capability for a passive partition to share a node with multiple active partitions would allow a distributed system to be configured into a standard, multiprogramming system, but this may not be practical for all environments.

The mapping of partitions to the target environment must be consistent with the call and data references to RCI and shared passive library units, respectively. This requires only that the target environment support the necessary communication connectivity among the nodes; it does not guarantee that active partitions are elaborated in a particular order required by the calls and references. To allow partitions to elaborate independently, a remote subprogram call is held until the receiving partition has completed its elaboration. If cyclic elaboration dependencies result in a deadlock as a result of remote subprogram calls, the exception `Program_Error` may be raised in one or all partitions upon detection of the deadlock.

The predefined exception `Communication_Error` (declared in package `System.RPC`) is provided to allow calling partitions to implement a means for continuing execution whenever a receiving partition becomes inaccessible. For example, when the receiving partition fails to elaborate, this exception is raised in all partitions that have outstanding remote calls to this partition.

To maintain interface consistency within a distributed system, the same version of an RCI or a shared passive library unit specification must be used in all elaborations of partitions that reference the same library unit. The consistency check cannot happen before the configuration step. (The detection of unit inconsistency, achievable when linking a single Ada program, cannot be guaranteed at that time for the case of a distributed system.) It is implementation defined how this check is accomplished; `Program_Error` may be raised but in any event the partions concerned become inaccessible to one another (and thus later probably resulting in `Communication_Error`); see [RM95 E.3].

In addition to the partition termination rules, an implementation could provide the capability for one partition to explicitly terminate (abort) another partition; the value of the attribute `Partition_ID` may be used to identify the partition to be aborted. If a partition is aborted while executing a subprogram called by another partition, `Communication_Error` will be raised in the calling partition since the receiving partition is no longer accessible.

E.7 Partition Communication Subsystem

The partition communication subsystem (PCS) is notionally compatible with the proposed Communications Model specified by the in-progress recommendations of the ISO RPC Committee Draft. The Annex requires that, as a minimum capability, the PCS must implement the standard package RPC to service remote subprogram calls. Standardizing the interface between the generated stubs for remote subprogram calls and the message-passing layer of the target communications software (the RPC package) facilitates a balanced approach for separating the implementation responsibilities of supporting distributed systems across different target environments.

The remote procedure call (RPC) paradigm was selected as the specified communication facility rather than message passing or remote entry call because of the following advantages of RPC:

- The RPC paradigm is widely implemented for interprocess communication between processes in different computers across a network. Several standards have been initiated by organizations such as ISO and OSF. Furthermore, emerging distributed operating system kernels promote support for RPC. Such considerations require that a language for programming distributed systems provide RPC as a linguistic abstraction. Finally, the need for RPC support is identified in U.S. Government initiatives towards developing open systems.

- A tenet of the revision is to maintain the type safety properties of the existing standard. Type-safe interfaces among partitions are a consequence of using the RPC paradigm. RPC is a compatible extension of the standard which, unless included in the Annex, would be difficult to support (by user-defined facilities) since detailed information on the compiler implementation is required.

- The RPC paradigm allows programs to be written with minimal regard for whether the program is targeted for distributed or nondistributed execution. Except in the instance of asynchronous procedure calls, the execution site implies no change in semantics from that of a local subprogram call. This is necessary for partitioning library units into various distributed configurations in a seamless or transparent manner. Furthermore, the use of RPC maintains concurrency/parallelism as orthogonal to distribution. This orthogonality reduces the complexity of the run-time system and allows remote references to be controlled through straightforward restrictions.

- The asynchronous form of RPC relaxes the normal synchronous semantics of RPC. This facilitates programming efficient application-specific communication paradigms where at-most-once semantics are not required.

Ada 95 includes important enhancements that allow dynamic subprogram calls using access-to-subprogram types and tagged types. To restrict these enhancements to nondistributed programs is likely to promote criticism similar to the absence of dynamic calls in Ada 83. In addition, the capability to support remote dispatching is an important discriminator between Ada 95 and other competing languages.

Package System.RPC

This package specifies the standard interface necessary to implement stubs at both the calling and receiving partitions for a remote subprogram call. The interface specifies both the actual operations and the semantic conditions under which they are to be used by the stubs. It is also adaptable to different target environments. Additional non-standard interfaces may be specified by the PCS. For example, a simple message passing capability may be specified for exchanging objects of some message type using child library units.

(Note that the normal user does have to use this package but only the implementer of the communication system.)

The package specifies the primitive operations for `Root_Stream_Type` to marshall and unmarshall message data by using the attributes `Read` and `Write` within the stubs. This allows an implementation to define the format of messages to be compatible with whatever message-passing capability between partitions is available from the target communication software.

The routing of parameters to a remote subprogram is supported by the `Partition_ID` type that identifies the partition, plus implementation-specific identifiers passed in the stream itself to identify the particular RCI library unit and remote subprogram. A value of type `Partition_ID` identifies the partition to which a library unit is assigned by the post-compilation partitioning.

The procedures `RPC` and `APC` support the generation of stubs for the synchronous and asynchronous forms of remote subprogram call. Each procedure specifies the partition that is the target of the call and the appropriate message data to be delivered. For the synchronous form, the result data to be received upon the completion the call is specified. As a consequence, the task originating the remote subprogram call is suspended to await the receipt of this data. In contrast, the asynchronous form does not suspend the originating task to await the receipt of the result data.

To facilitate the routing of remote calls in the receiving partition, the procedure `Establish_RPC_Receiver` is specified to establish the interface for receiving a message and for dispatching to the appropriate subprogram. The interface is standardized through the parameters specified for an access-to-subprogram type that designates an implementation-provided RPC-receiver procedure. In this way, post-compilation support can link the necessary units and data to the RPC-receiver procedure. Once the RPC-receiver procedure has been elaborated, it may be called by the PCS.

A single exception `Communication_Error` is specified to report error conditions detected by the PCS. Detailed information on the precise condition may be provided through the exception occurrence. These conditions are necessarily implementation-defined, and therefore, inappropriate for inclusion in the specification as distinct exception names.

E.7.1 The Interaction Between the Stubs and the PCS

The execution environment of the PCS is defined as an Ada environment. This is done in order to provide Ada semantics to serving partitions. In the calling and receiving partitions, the canonical implementation relies upon the Ada concurrency model to service stubs. For example, in the calling partition, cancellation of a synchronous remote call, when the calling task has been aborted, requires that the PCS interrupt the `Do_RPC` operation to execute the cancellation. In the receiving partition, the stub for a remote subprogram is assumed to be called by the RPC-receiver procedure executing in a task created by the PCS.

E.8 Requirements Summary

The facilities of the Distributed Systems annex relate to the requirements in 8.1 (Distribution of Ada Applications) and 8.2 (Dynamic Reconfiguration of Distributed Systems).

More specifically, the requirement

R8.1-A(1) — Facilitating Software Distribution

is met by the concept of partitions and the categorization of library units and the requirement

R8.2-A(1) — Dynamic Reconfiguration

is addressed by the use of remote access to subprogram and access to class wide types for dynamic calls across partitions.

F Information Systems

One of the major goals of Ada 95 is to provide the necessary language facilities for the development of large-scale information systems that previously have been produced in COBOL and 4GLs. To a large extent, core language enhancements such as child units and object-oriented programming, and the new support for distribution, serve to meet this goal. However, there are also specific requirements at the computational level and for report-oriented output that must be addressed in order to ensure applicability to financial and related Information Systems applications. The major needs are

- Exact, decimal arithmetic for quantities up to at least 18 digits of precision;
- The ability to produce human-readable formats for such values, with control over the form and placement of currency symbol, sign, digits separator, and radix mark;
- The ability to interface with data produced by, or programs written in, other languages (in particular C and COBOL).

This chapter describes the facilities and gives the reasons for the major decisions taken in Ada 95 to satisfy these requirements.

F.1 Decimal Computation

A numeric model highly appropriate for information systems, especially for financial applications, is that supplied by the COBOL language. In COBOL the programmer defines numeric items via a "picture" in terms of a specified number of decimal digits and the placement of the decimal point. The arithmetic verbs provide exact arithmetic, with control over truncation versus rounding on a per-computation basis. For example:

```
05 FRACTION           PIC S9V99    VALUE .25.
05 ALPHA              PIC S9999V9  VALUE 103.
05 BETA               PIC S9999V9.
```

`FRACTION` has values in the range `-9.99` through `9.99`, and each of `ALPHA` and `BETA` is in the range `-9999.9` through `9999.9`.

```
  MULTIPLY ALPHA BY FRACTION GIVING BETA ROUNDED.
* Now BETA = 25.8, the rounded value of 25.75
  ADD ALPHA TO BETA.
* Now BETA = 128.8
  DIVIDE BETA BY 10.
* Now BETA = 12.8, since truncation is the default
```

It is also possible to express the above calculation more succinctly in COBOL:

```
COMPUTE BETA = (ALPHA * FRACTION + ALPHA) / 10.
```

However, the effect of rounding versus truncation is now implementation dependent, so the result may be either 12.8 or 12.9.

F.1.1 Decimal Arithmetic through Discriminated Type

In addressing the exact computational requirements, we examined several alternatives. One was to rely on a private discriminated type, with discriminants reflecting scale and precision. (The terminology here is the same as in SQL [ISO 92]: precision is the total number of decimal digits; scale is the number of digits after the decimal point.) For example

```
package Computation is
     subtype Scale_Range is
       Integer range implementation-defined .. implementation-defined;
     subtype Precision_Range is
       Positive range 1 .. implementation-defined;
     type Decimal(Precision : Precision_Range;
                  Scale     : Scale_Range) is private;
     ... -- Subprograms for arithmetic
end Computation;
```

Such an approach would have the benefit of separability from the core features, but its numerous drawbacks led to its rejection:

- Literals are unavailable for private types, hence the programmer would need to perform explicit conversions either from `String` or from a specific real type. Such a style would be both inefficient and aesthetically displeasing. In an early version of the Ada 9X mapping there was a capability to obtain numeric literals for private types, but this was removed as a simplification.

- Non-trivial optimizations are needed to avoid time and space overhead. In COBOL precision and scale are known at compile time, so the compiler can generate efficient code. The discriminated type approach lets the programmer defer specifying the precision and scale until run time, but the generality comes at a price.

- A problem often cited with the COBOL model is the lack of typing. For example, if by mistake a COBOL programmer adds a unitless fraction to a salary, this error will not be detected by the compiler. To obtain compile-time protection from such an error in Ada, the programmer would need to derive from type `Decimal`, for example to declare the types `Fraction` and `Salary`. However, derivation provides more operations than make sense and hence other kinds of errors could still arise (for example, multiplying a `Salary` by a `Salary` to obtain a `Salary`). At the same time it yields less than what is needed; for example, it would be useful to be able to divide two `Salary` values and obtain a `Fraction`, but this would not be provided automatically. Although both of these problems could be solved by the programmer providing some additional explicit declarations, programmers might be tempted to forego the type derivations (and the resulting safety) and to simply declare all their data of type `Decimal`.

- Specifying just precision and scale allows more values than might be sensible. For example, if we want a fraction value to be in the range 0.00 .. 1.00, we need to specify `Decimal(Precision=>3, Scale=>2)`, but this allows all values in the range -9.99 .. 9.99.

Another major problem with the discriminated type approach is the error-prone effects of having arithmetic operators take parameters of type `Decimal` and deliver a result of type `Decimal`.

Division in particular is troublesome; languages that attempt to address the issue lead inevitably to anomalies. For example, the well-known curiosity in PL/I is that the operation 10+1/3 overflows, since the language rules attempt to maximize the precision of the quotient. Moreover, the rules for precision and scale of an arithmetic result would clash with the need for discriminant identity on assignment. For example, consider the simple fragment:

```
declare
   Salary : Decimal(Precision => 9, Scale => 2);
            -- Values in -99_999_99.99 .. 99_999_999.99
   Fraction : Decimal(Precision => 2, Scale => 2);
            -- Values in -0.99 .. 0.99
begin
   ...
   Salary := Salary * Fraction;
   ...
end;
```

The intuitive rule for "*" would be to define the precision of the result as the sum of the precisions of the operands, and similarly for the scale. Thus `Salary*Fraction` would have precision 11 and scale 4, sufficient to hold any product. But then the rules for discriminant matching would cause `Constraint_Error` to be raised by the assignment to `Salary`.

A possible solution would be to introduce special rules for discriminant matching in such cases, but this adds complexity. An alternative would be to omit the operator forms for the arithmetic subprograms and instead to provide a procedural interface with an out parameter, thereby making the result precision and scale known at the point of call. For example:

```
procedure Multiply
   (Left, Right : in Decimal;
    Result      : out Decimal;
    Rounding    : in Boolean := False);
```

Although such an approach has been successfully applied in the Ada 83 Decimal Arithmetic and Representation components [Brosgol 92], the other drawbacks led us to seek alternative solutions for Ada 95.

F.1.2 Decimal Arithmetic and Ada 83 Numeric Types

The Ada 83 numeric types give us a choice among integer, floating point, and fixed point. In some sense integer arithmetic provides the most appropriate computational model, since it matches the requirements for exact results. For example, one might consider using an integer type Pennies to represent monetary values. However, this would be impractical for several reasons: the absence of real literals is a hardship, keeping track of implicit scaling is awkward, and many compilers do not support the 64-bit integer arithmetic that would be needed for 18 digits of accuracy.

Floating point is unacceptable because of the inherent inexactness of representing decimal quantities. Consider the following program fragment, where `X` is a floating point variable:

```
X := 0.0;
for I in 1 .. 10 loop
   X := X + 0.10;
end loop;
```

After execution of the loop using typical floating point hardware, `X` will not equal `1.00`. Moreover, 64-bit floating point does not have enough mantissa bits to represent 18 decimal digits.

At first glance, fixed point seems no better. The apparent motivations behind the fixed point facility in Ada were to deal with scaled data coming in from sensors in real-time applications, and to provide a substitute for floating point in target environments lacking floating point hardware. Indeed, the inherent bias toward powers of 2 for the attribute `Small` in the Ada 83 fixed point model seems at odds with the needs of decimal computation.

However, fixed point provides a closer fit than might be expected [Dewar 90b]. The Ada 83 unification of floating point and fixed point under the category of "approximate" computation is more artificial than real, since the model-number inaccuracy that is appropriate in the floating point case because of differences in target hardware is not applicable at all to fixed point. The fixed point arithmetic operations `"+"`, `"-"`, `"*"`, `"/"` are exact, and through a `Small` representation clause the programmer can specify decimal scaling. Thus consider a COBOL declaration

```
05   SALARY PICTURE S9(6)V9(2) USAGE COMPUTATIONAL.
```

which defines `SALARY` as a signed binary data item comprising 8 decimal digits, of which 2 are after the assumed decimal point. This can be simulated in Ada:

```
type Dollars is delta 0.01 range -999_999.99 .. 999_999.99;
for Dollars'Small use 0.01;

Salary : Dollars;
```

The programmer-specified `Small` not only provides the required decimal scaling, it also prevents the implementation from supplying extra fractional digits. This is important in financial applications: if the programmer requests 2 fractional digits, it would be incorrect for a compiler to provide 3.

The fixed point approach immediately avoids several of the problems with discriminated types: we get numeric literals, compile-time known scales and precisions, strong typing, and the ability to specify logical ranges. Moreover, the rules for the arithmetic operators are fitting. The `"+"` and `"-"` operators require identical operand types and deliver a result of the same type, which is an intuitively correct restriction. Adding or subtracting quantities with different scales is not a frequent situation; when it arises, it is reasonable to require an explicit conversion to indicate the rescaling. Automatic availability of mixed-type `"*"` and `"/"` also makes sense.

There are, however, several problems with adopting the Ada 83 fixed point model unchanged for decimal arithmetic.

- The Ada fixed point model leads to occasional surprises, even in the presence of a `Small` representation clause. For example, one or both endpoints supplied in the definition of a fixed point type may be absent from the implemented range for the type.

- The Ada 83 fixed point rules require conversions of real literals (and named numbers of type *universal_real*) that appear as factors in multiplication or division. Without the programmer providing an explicit declaration of an applicable `"*"` operator, it would be illegal to write:

  ```
  Worker_Salary := 1.05 * Worker_Salary;
  ```

  Instead, something like the following circumlocution is required:

  ```
  Worker_Salary := Dollars(Some_Type(1.05) * Worker_Salary);
  ```

  The need for the programmer to supply either these explicit conversions or an explicit overloading of `"*"`, is somewhat embarrassing. In COBOL the equivalent functionality can be obtained directly:

  ```
  MULTIPLY WORKER-SALARY BY 1.05.
  ```

- The previous example illustrates another serious problem: Ada fixed point does not give a well-defined result for the conversion of values. That is, the language does not guarantee whether the result of a fixed point conversion is to be rounded or truncated. In fact, different evaluations of the same expression in the same program could yield different results, an unwelcome nondeterminism.

- Facilities such as edited output, and a division operation delivering both a quotient and remainder, are not defined for fixed point types.

F.1.3 Decimal Arithmetic through Decimal Types

Since fixed point comes reasonably close to satisfying the requirements for decimal arithmetic, our decision was to use that facility as the basis for a solution. Ada 95 thus introduces a new class of fixed point types, the *decimal* types, distinguished syntactically by the presence of a positive integer **digits** value following the **delta** in a fixed point definition. The **delta** .. **digits** .. syntax, suggested by David Emery [Emery 91], has the advantage of identifying the type immediately as a special kind of fixed point type (the **delta**) without requiring new reserved words.

The **delta** value must be a power of 10. For example:

```
type Francs is delta 0.01 digits 9;
```

This declaration is similar in effect to the Ada 83 fragment:

```
type Francs is delta 0.01 range -(10.0**9 - 1.0) .. 10.0**9 - 1.0;
for Francs'Small use 0.01;
```

The **digits** value in a decimal fixed point type definition thus implies a range constraint. For a decimal type with delta `D` and digits `N` (both of which must be static), the implied range is `-(10.0**N - 1.0)*D .. (10.0**N - 1.0)*D`. Moreover, a range constraint may be further supplied at the definition of a decimal type or subtype, and at the declaration of objects. For example:

```
type    Salary is delta 0.01 digits 8 range 0.00 .. 100_000.00;
subtype Price  is Francs              range 0.00 .. 1000.00;
Worker_Salary : Salary range 0.00 .. 50_000.00;
```

The ordinary fixed point operations, such as the arithmetic operators and fixed point attributes, are available for decimal types. There are, however, several important differences:

- For a decimal subtype `S`, the conversion `S(`*expr*`)` where *expr* is of some numeric type is defined to truncate (towards 0) rather than having an unspecified effect.

- To obtain a rounded result for an expression *expr* having a real type, the function attribute `S'Round(`*expr*`)` can be used. This attribute is not available for ordinary fixed point types.

- Other attributes apply only to decimal subtypes: for example, `S'Digits` and `S'Scale`. The former reflects the value of digits supplied for the declaration of `S`. The latter is the number of digits after the decimal point in values of `S` and is related to `S'Delta` by the equation

  ```
  10.0**(-S'Scale) = S'Delta
  ```

A stylistic issue noted above, namely the inability in Ada 83 to write simple statements such as:

```
Worker_Salary := 1.05 * Worker_Salary;
```

has been solved in Ada 95 for fixed point types in general. The revised rules permit a *universal_fixed* value to be implicitly converted to a specific target type if the context uniquely establishes the target type. Thus there is no need to convert to `Salary` the product on the right side of the assignment. Another new rule allows a *universal_real* value to be used as an operand to a fixed point `"*"` and `"/"`; thus there is no need to convert the literal `1.05` to a specific type. Although these enhancements are motivated by considerations with decimal types, it makes no sense either from an implementation or user viewpoint to apply the new rules only to decimal types, and thus they have been generalized for ordinary fixed point types as well.

Given that decimal types come equipped with their own operations, it is natural to introduce a category of generic formal type that can only be matched by decimal subtypes. The syntax for such a generic formal type is what one would expect:

```
type T is delta <> digits <>;
```

The actual subtype supplied for a formal decimal type must be a decimal subtype. This makes sense, since an ordinary fixed point subtype does not have all the necessary operations. On the other hand, there is a design issue whether to permit an actual decimal subtype to match a formal fixed point type (one given by **`delta`** `<>`). Such permission would seem to be useful, since it would allow existing Ada 83 fixed point generics to be matched by Ada 95 decimal subtypes. However, it would introduce some implementation difficulties, especially for those compilers that attempt to share the code of the template across multiple instances. The fact that some operations (in particular numeric conversion) behave differently for decimal and ordinary fixed point would also cause complications if decimal subtypes were permitted to match formal fixed point types. Thus the decimal fixed point types are defined to form a class disjoint from ordinary fixed point types with respect to generic matching.

Formal decimal types are exploited to provide edited output (see below) as well as division delivering both a quotient and a remainder.

One of the requirements for information systems applications is the ability to perform edited output of decimal quantities. We considered introducing decimal subtype attributes for this effect; for example `S'Image(X, Picture)` would return a `String` based on the value of `X` and the formatting conventions of `Picture`. However, this approach would have introduced implementation complexity out of proportion to the notational benefit for users. The type of `Picture` is defined in an external package, making such an attribute rather atypical, and support would affect the compiler and not simply require a supplemental package. Instead, picture-based output is obtained via generics, as described below.

F.1.4 Internal Representation for Decimal Objects

Ada and COBOL have a somewhat different philosophy about internal data representation. Through the `USAGE` clause the COBOL programmer furnishes information about how numeric items will be represented, either explicitly (such as `BINARY`, `DISPLAY`, `PACKED-DECIMAL`) or by default (`DISPLAY`). COBOL's default representation opts for data portability versus computational efficiency.

Ada's approach to data representation, for types in general and not just decimal, is to let the compiler decide based on efficiency, and to let the programmer override this choice explicitly when necessary. For decimal types this is achieved through the `Machine_Radix` attribute and the corresponding attribute definition clause.

An object of a decimal type, as with fixed point in general, may be viewed as the product of an integer mantissa (represented explicitly at run time) and the type's delta (managed at compile time). The type's Machine_Radix determines the representation of the mantissa: a value of 2 implies binary, while a value of 10 implies decimal. The compiler will choose an implementation-defined machine radix by default, which the programmer can override with an explicit attribute definition clause. Consider the following example, where the implementation's default for all decimal types is binary machine radix:

```
type Money_2  is delta 0.01 digits 18;
type Money_10 is delta 0.01 digits 18;
for Money_10'Machine_Radix use 10;
```

An object of type Money_2 is represented in binary; on typical machines it will occupy 64 bits (including a sign).

An object of type Money_10 is represented in decimal; it will take 18 digits (and a sign). The exact representation is unspecified, and in fact different machines have different formats for packed decimal concerning how the sign is encoded. If a decimal type's machine radix is 10, then the compiler may also generate packed-decimal instructions for arithmetic computation. Whether it chooses to do so, rather than converting to/from binary and using binary arithmetic, depends on which is more efficient.

The only difference in behavior between decimal and binary machine radix, aside from performance, is that some intermediate results might overflow in one case but not the other. For example, if Money_10 values are represented in 19 digits (an odd number is typical for packed decimal, since the sign can be stored in the same byte as a digit), and Money_2 values occupy 64 bits, then a computation such as (100.0 * Money)/100.0 will overflow if Money has type Money_10, but not if Money has type Money_2, where Money is 10.0**18 - 1.0.

Implementations using packed decimal are encouraged to exploit subtype digits constraints for space economization. For example:

```
Pay : Money_10 digits 9;
```

The compiler can and should represent Pay in 9 digits rather than 18 as would be needed in general for Money_10.

Ada does not provide the equivalent of DISPLAY usage for decimal data, since computation on character strings would be inefficient. If the programmer wishes to store decimal data in an external file in a portable fashion, the recommended approach is to convert via the To_Display function in Interfaces.COBOL.Decimal_Conversions; see B.3.

F.1.5 Compliance

The decimal type facility is part of the core language; thus the syntax for decimal types and for formal generic decimal types must be supported by all implementations. However, since a compiler needs to implement ordinary fixed point only for values of Small that are powers of 2, it may reject the declaration of a decimal type and also the declaration of a generic unit with a formal decimal type parameter. To be compliant with the Information Systems Annex a compiler must implement decimal types and must also allow digits values up to at least 18.

We had considered requiring support for decimal types (but without the 18 digit capacity) for all implementations. However, this was judged a heavy implementation burden for a facility whose usage is fairly specialized.

F.2 Edited Output

A facility essential for financial and other Information Systems applications and long established in COBOL is the ability for the programmer to dictate the appearance of numeric data as character strings, for example for reports or for display to human readers. Known as edited output, such a facility allows control over the placement and form of various elements:

- The sign;
- The radix mark, which separates the integer part of the number from the fraction;
- The digits separator, which separates groups of digits to improve readability;
- The currency symbol;
- The treatment of leading zeros, for example whether they should appear explicitly as `'0'` characters, as blank space, or as a string of occurrences of a "check protection" character.

COBOL's approach is to associate a "picture string" with the target data item for the edited output string. When a numeric value is moved to that target item, the associated picture determines the form of the output string. For example:

```
05 OUTPUT-FIELD PIC S$,ZZ9.99.
05 DATA-1       PIC S9999V99   VALUE -1234.56.
...
MOVE DATA-1 TO OUTPUT-FIELD.
```

The contents of `OUTPUT-FIELD` after the move are `"-$bb1,234.56"` where `'b'` denotes the blank character.

F.2.1 General Approach

Textual I/O for decimal types is obtained in the same fashion as for other numeric types, by generic instantiation. The generic package `Decimal_IO` in `Text_IO` supplies `Get` and `Put` procedures for a decimal type with analogous effects to `Get` and `Put` in `Text_IO.Fixed_IO` for an ordinary fixed point type. Supplementing these facilities is a child package `Text_IO.Editing` in the Information Systems Annex, which provides several facilities:

- A private type `Picture` and associated operations. A `Picture` object encodes the formatting information supplied in a "picture string" concerning the placement of so-called "editing characters" in an output string;
- Constants for the default localization elements. These elements comprise the currency string, and the characters for fill of leading zeros, for digits separation, and for the radix mark;
- A generic package `Decimal_Output` allowing COBOL-style edited output for decimal types.

The `Decimal_Output` package supplies an `Image` function and several `Put` procedures, each taking an `Item` parameter (of the decimal type), a `Pic` parameter (of type `Picture`), and parameters for the localization effects. The default values for the localization parameters can be supplied as generic parameters; if not, then the default values declared in the enclosing package `Text_IO.Editing` are used.

An alternative that we considered for the picture parameter was to have it directly as a `String`, but this would make optimizations difficult. Hence package `Editing` supplies a private type `Picture`, conversion functions between `String` and `Picture`, and a function `Valid` that checks a string for well-formedness. Since picture strings are dynamically computable, the approach provides substantial flexibility. For example, an interactive program such as a spreadsheet could obtain the picture string at run time from the user. On the other hand, if the programmer only needs static picture strings, the compiler can exploit this and produce optimized inline expansions of calls of the edited output subprograms.

An example of a typical usage style is as follows:

```
with Text_IO.Editing;
procedure Example is
   use Text_IO.Editing;
   type Salary is delta 0.01 digits 9;
   package Salary_Output is new Decimal_Output(Salary);
   S     : Salary;
   S_Pic : constant Picture := To_Picture("$*_***_**9.99");
begin
   S := 12345.67
   Salary_Output.Put(S, S_Pic);  -- Produces "$***12,345.67"
end Example;
```

We recognize that someone coming to Ada from COBOL may find the style somewhat unusual. In COBOL, performing edited output involves simply defining a picture and doing a `MOVE`, whereas in Ada 95 it is necessary to instantiate a generic, convert a string to a private type, and invoke a subprogram. However, this is principally a training and transition issue, which experience has shown to be solvable via an appropriate pedagogical style. Moreover, generics and private types are features of Ada that all programmers will need to understand and employ. Since these features apply naturally to the problem of edited output, there is little point in trying to disguise this.

F.2.2 Relationship to COBOL Edited Output

There are several reasons for basing the Ada 95 edited output facility directly on COBOL. First, the programmer population toward whom Ada 95's information systems support is targeted comprises largely COBOL users. Second, although enhanced edited output mechanisms have appeared in modern spreadsheet utilities, their proprietary nature makes commercial products an unappealing candidate as a source of specific features.

Still there was the issue of whether to adopt COBOL's "picture" approach as closely as possible, or to use it more loosely as the basis for a more comprehensive but possibly incompatible facility. We have taken the former approach for several reasons:

- To redesign the edited output facility from starting principles would have required a detailed review of the entire history behind the current COBOL standard's approach, an effort that would have been outside the scope of the Ada 9X project.

- Basing the Ada 95 edited output rules directly on COBOL obviously reduces the learning curve for COBOL programmers.

As a result the rules for picture string formation and interpretation for edited output are identical to those in ISO standard COBOL, except for the following:

- In Ada the picture characters for currency symbol, digits separator, and radix mark are not overridable. `'$'` and `'#'` are the currency symbols, `'_'` is the digits separator and `'.'` is the radix mark. No other characters can be used for these purposes *in the picture string*.

- On the other hand, Ada provides more flexibility than COBOL in the run-time localization of currency symbol, digits separator, radix mark, and "fill character" (also known as the "check protection character"). The programmer can arrange localization by passing explicit parameters to the edited output subprograms, or by instantiating the generic `Decimal_Output` package with values to be used as defaults for the localization elements.

- The currency symbol can be localized to a multi-character string; each of the other localization elements can be localized to any single character. The first (or only) occurrence of `'$'` in a picture string represents as many positions in the edited output result as there are characters in the current currency string. Subsequent occurrences of the symbol represent just one position in the edited output string.

- Ada allows a multi-character currency substring of the picture string to stand for a substring with the same length in the edited output string, if `'#'` is the currency symbol. This "length invariant" property can be useful in programs that need to deal with different currencies.

- Ada also allows the use of parentheses for negative quantities in the edited output string. The angle bracket characters `'<'` and `'>'` in the picture string denote positions where `'('` and `')'` can appear in the edited output. (The parentheses characters themselves have other meaning in picture strings, surrounding a count indicating repetition of the preceding picture character as in COBOL. The angle brackets were chosen since they look enough like parentheses to remind the user of their effect.)

- Ada allows the currency symbol to the right of the number as well as to the left.

There are several reasons why we have not adopted the COBOL-style permission to provide a single-character replacement in the picture string for the `'$'` as currency symbol, or to interchange the roles of `'.'` and `','` in picture strings:

- It would have introduced considerable complexity into Ada, as well as confusion between run-time and compile-time character interpretation, since picture strings are dynamically computable in Ada, in contrast with COBOL.

- Ada's rules for real literals provide a standard and natural interpretation of `'_'` as digits separator and `'.'` for radix mark; it is not essential to allow these to be localized in picture strings, since Ada does not allow them to be localized in real literals.

- The COBOL restriction for the currency symbol in a picture string to be replaced by a single character currency symbol is a compromise solution. In any event a mechanism is needed to localize the edited output to be a multi-character currency string. Allowing a single-character localization for the picture character, and a multiple-character localization for the currency string, would be an unwelcome complication.

The enhancement of the picture string form to allow parentheses for negative quantities is not in the current COBOL standard, but it is a real need in many financial applications. Thus the additional rules were judged to be worth the cost.

The approach to currency symbol localization is consistent with the directions that the ISO COBOL standardization group (WG4) is taking [Sales 92]. Thus we are attempting to preserve compatibility not just with the existing COBOL standard but also with the version currently under development.

In COBOL, the `BLANK WHEN 0` clause for a numeric edited item interacts with edited output. For example, if `OUTPUT-ITEM` is defined as follows:

```
05   OUTPUT-ITEM  PIC -9999.99 BLANK WHEN 0.
...
MOVE 0 to OUTPUT-ITEM.
```

then `OUTPUT-ITEM` will contain a string of 8 blanks. In the absence of the `BLANK WHEN 0` clause, `OUTPUT-ITEM` would contain `"b0000.00"`. The effect of the `BLANK WHEN 0` clause is considered in Ada to be part of the `Picture` value; thus the function `To_Picture` takes not just a picture string but also a Boolean value reflecting whether a 0 value is to be treated as all blanks.

The edited output rules in the Ada standard are given by a combination of BNF (for "well-formed picture strings") and expansion rules that define the edited output of a non-terminal in terms of the edited output for the right sides of the rules. We had considered defining the rules instead by a direct reference to the COBOL standard, but that would have had two undesirable consequences. First, it would have required the reader to be familiar with a rather complicated section of a document (the COBOL standard) that would not necessarily be easily accessible. Second, the reference would become obsolete when the COBOL standard is revised.

F.2.3 Example

The following example illustrates edited output with localization:

```
with Text_IO.Editing;
procedure Example is
   use Text_IO.Editing;
   type Money is delta 0.01 digits 8;
   package Money_Output is new Decimal_Output(Money);

   package Money_Output_FF is
      new Decimal_Output(
         Money,
         Default_Currency   => "FF",
         Default_Fill       => '*',
         Default_Separator  => '.',
         Default_Radix_Mark => ',');

   Amount     : Money range 0.0 .. Money'Last;
   Amount_Pic : constant Picture := To_Picture("$$$$_$$9.99");

begin
   Amount := 1234.56;

   Money_Output.Put(Item => Amount,
                    Pic  => Amount_Pic );
   -- Outputs the string "bb$1,234.56"
   -- where 'b' designates the space character

   Money_Output_FF.Put(Item => Amount,
                       Pic  => Amount_Pic );
   -- Outputs the string "bbFF1.234,56"

   Money_Output.Put(Item       => Amount,
                    Pic        => Amount_Pic,
                    Currency   => "CHF",
                    Fill       => '*',
                    Separator  => ',',
                    Radix_Mark => '.' );
   -- Outputs the string "bbCHF1,234.56"
```

```
      Money_Output.Put(Item       => Amount,
                       Pic        => To_Picture("####_##9.99")
                       Currency   => "CHF",
                       Fill       => '*',
                       Separator  => ',',
                       Radix_Mark => '.' );
      -- Outputs the string "CHF1,234.56"

   end Example;
```

F.3 Requirements Summary

The facilities of the Information Systems Annex relate to the requirements in 10.1 (Handling Currency Quantities for Information Systems), 10.2 (Compatibility with Other Character Sets), 10.3 (Interfacing with Data Base Systems), and 10.4 (Common Functions).

The requirement

R10.1-A(1) — Decimal-Based Types

is satisfied by the Ada 95 decimal fixed point type facility.

The study topic

S10.1-A(2) — Specification of Decimal Representation

is met in part by the `Machine_Radix` attribute definition clause.

The study topic

S10.2-A(1) — Alternate Character Set Support

is satisfied in part by the permission of an implementation to localize the declaration of type `Character`.

The study topic

S10.3-A(1) — Interfacing with Data Base Systems

is satisfied in part by the provision of decimal types and also the package `Interfaces.COBOL` (see B.3).

The study topic

S10.4-A(2) — String Manipulation Functions

is met in part by the edited output facilities.