2

Catalogue of Interface Features and Options for the Ada Runtime Environment

Release 3.0

July, 1991

Ada Runtime Environment Working Group
Interfaces Subgroup

Association for Computing Machinery
Special Interest Group for Ada

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
July 1991

3. REPORT TYPE AND DATES COVERED
Final Report

4. TITLE AND SUBTITLE
Catalogue of Interface Features and Options
for the Ada Runtime Environment

5. FUNDING NUMBERS

6. AUTHOR(S)
Ada Runtime Environment
Working Group (ARTEWG)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Runtime Environment Working Group
Charles McKay, Chair of the Interface Subgroup
University of Houston Clear Lake
2700 Bay Area Blvd.
Houston, TX 77058

8. PERFORMING ORGANIZATION REPORT NUMBER
Release 3.0

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Special Interest Group on Ada (SIGAda),
Association for Computing Machinery, Inc. (ACM)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
To be published in Ada Letters
Supersedes Release 2.0

12a. DISTRIBUTION/AVAILABILITY STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This catalogue describes interface features and options for tailorable Ada runtime environments. Embedded systems in particular, often require scheduling regimens, pre-elaboration, fast interrupt handlers and other features that are not typically provided with compilers. The catalogue proposes common interfaces that address these issues.

14. SUBJECT TERMS
Ada, Runtime Environments

15. NUMBER OF PAGES
174

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

Preface

April 8, 1991

Dear Colleagues,

On May 6, 1985 in Clear Lake, Texas, the principal members of the Ada Runtime Environment Working Group (ARTEWG, pronounced "artwig") held their first working group meeting.

The ARTEWG is sponsored by the Special Interest Group on Ada (SIGAda) of the Association for Computing Machinery (ACM). The goals of the ARTEWG are to contribute to the advancement of Ada runtime environments that facilitate the reusability and transportability of Ada program components, improve the performance of those components, and provide a framework which can be used to evaluate Ada runtime systems. The ARTEWG serves as a mechanism for users to interface effectively with Ada implementors, thereby encouraging development of runtime environments that meet user's needs. The Ada Joint Program Office (AJPO) recognizes and encourages the objectives of the ARTEWG, and is receptive to the consolidated voice of the Ada community as represented by the ARTEWG on Ada runtime environment issues.

This is the third release of the *Catalogue of Interface Features and Options* (CIFO). It differs from the previous releases because of at least four significant factors:

refinement made possible by feedback from actual implementations and applications of previous releases,

convergence of efforts and progress with the Extensions des services Temps Reels Ada (ExTRA) team,

benefits of insights gained in participating in the Ada 9X process (ARTEWG strongly supports the Ada 9X process!) and of insights gained in the processes of addressing runtime environment issues in some major new programs, and

the current perspective of the ARTEWG regarding the perceived role for a release of the CIFO vis-a-vis the evolution of Ada 9X and secondary standards. (This perspective is elaborated in the introductory sections of the document.)

The feedback that has resulted from the baselining of CIFO by projects and programs such as the Space Station Freedom Program, combined with vendor and user implementations, has been both helpful and rewarding. Although a few features and options in the earlier releases have been superseded or dropped because of the lessons learned, the main thrusts of the feedback have been to tighten and clarify the semantics of the Entries and to add new Entries that are clearly needed by a growing number of users.

Using Release 2.0 of the CIFO, the ExTRA team (sponsored by the French Armament Board) produced several interface features that addressed the needs of embedded avionics systems. Because of the obvious similarities of many of the goals and objectives of the two teams, a series of meetings was held to explore the potential benefits of merging their efforts and products. The two teams decided to combine forces and Release 3 already reflects many of the benefits anticipated for this convergence.

The third major influence on this release is derived from efforts to integrate these lessons from experience with the insights gained from participating in the Ada 9X process and contributing to the resolution of issues faced in some major new programs such as the Joint Integrated Avionics Working Group (JIAWG). Collectively, this has resulted in a reorganization of the CIFO and some new Entries scheduled for future releases (past 3.0).

As explained in the introductory sections of the document, the current perspective of the ARTEWG has been greatly influenced by experience with the first three factors. The ARTEWG strongly endorses the Ada 9X

Project and looks forward to its language based solutions. However, CIFO 3.0 clearly provides some needed support to our runtime environment constituents now--considerably in advance of the scheduled Ada 9X resolutions.

Soon after the conclusion of Ada 9X, another release of the CIFO will reflect significant changes made possible by the standardization of the Ada 9X final products. However, new and revised CIFO Entries are envisioned which address the needs of critical, embedded application domains that may not be possible or appropriate to completely satisfy in Ada 9X or subsequent language advancement programs. Such CIFO releases may well be appropriate candidates for "secondary standards" for certain application domains.

In closing, we sincerely believe these improvements will benefit the Ada community. This represents the best efforts of knowledgeable and experienced volunteers from cross-sections of users, implementors, vendors, government representatives, and academicians on what we consider to be the most important single deficiency in today's Ada implementations. ARTEWG and the Interface Subgroup needs and appreciates your comments and review of this work.

Thank you on behalf of ARTEWG and the Interface Subgroup.

Charles McKay, Chair of the
ARTEWG Interface Subgroup
University of Houston-Clear Lake
2700 Bay Area Blvd
Houston, Tx 77058
(713) 283-3830

J. Michael Kamrad, II
Chair of ARTEWG
Unisys-Computer Systems Div.
PO Box 64525
St. Paul, MN 55164-0525
(612) 456-7315

Table of Contents

Objectives

The Ada language intentionally (necessarily) leaves the details of many important capabilities of the Runtime System to the individual implementation, such as scheduling regimen, interrupt control, storage management, and so on. In many applications, such capabilities and services are essential to the successful realization of the project, and would have to be provided. Since the user's interface to these capabilities is often not specified in great detail, implementations will differ as they emerge.

An objective of this document is to propose and describe a common set of user-runtime environment interfaces from a user's perspective. These interfaces are described as Entries in this *Catalogue of Interface Features and Options* (CIFO). The Entry descriptions represent 'contracts' provided to the application builders, not to the CIFO builders. If CIFO builders of an Entry need to know the implementation details of another Entry, then an implementation of the other Entry must provide a separate contract for the Entry that includes these details.

With an implementation of these CIFO Entries, a programmer can both request services of the runtime environment (RTE) and tailor the RTE to meet application-specific requirements. By the word 'common', we mean that implementations are intended to provide the capabilities expressed by the language-oriented descriptions of these CIFO Entries in such a way that programmers need not learn new interfaces when using a different Ada implementation.

It should be emphasized that although by no means perfect, we do not view the language as being deficient in all of the areas addressed (although implementations often are). Highly application-dependent requirements are not appropriately found in a language reference manual. However, in keeping with the concept of 'extensibility', implementations should be expected to extend the language, in the manner prescribed by the Language Reference Manual (LRM), to meet application-specific requirements. Thus, many 'language deficiencies' are more correctly expressed as 'procurement issues'. Those implementations which expect to succeed in a particular marketplace must expect to supply the capabilities necessary to meet the specific requirements of that marketplace. The Entries which follow are an attempt to provide commonality at the user level for some of the capabilities that will often be required.

Another important objective for CIFO Release 3.0 is to provide some needed support to our runtime environment constituents now - considerably in advance of the scheduled Ada 9X resolutions. Furthermore, the demanding, embedded applications of the future are unlikely to be completely satisfied by the more general solution approaches appropriate for Ada 9X.

Soon after the conclusion of Ada 9X, another release of the CIFO will reflect significant changes made possible by the standardization of the Ada 9X final products. However, new and revised CIFO Entries are envisioned which address the needs of critical, embedded application domains that may not be possible or appropriate to completely satisfy in Ada 9X or subsequent language advancement programs. Such CIFO releases may well be appropriate candidates for "secondary standards" for certain application domains.

Using This Catalogue

Conformance

For the implementation of Entries from this catalogue, it is important that the catalogue not be viewed as one monolithic entity. Rather, this is a collection of Entries and most of the Entries are independent of each other. (When interactions exist, they are documented at the end of the discussion of each Entry and are cross-referenced in the interactions matrix at the end of this CIFO.)

For an example of independence of Entries, the Entry for "Interrupt Management" is independent of the one for "PreElaboration". Either interface can be implemented independently of the other. Even if both are included in a particular runtime environment, their implementations will most likely have no relationship.

Moreover, there are application areas, as well as hardware architectures, for which one of the Entries is irrelevant, while the other is of crucial importance. In such a situation, exporting the irrelevant Entry to the application RTE will actually be detrimental. It will only burden the system with something that is not used, and that may not even have meaning in the particular context of the application. For such reasons as safety, security, and maintenance, ARTEWG believes that implementations of CIFO Entries should be tailorable to avoid having to export irrelevant Entries to the RTE of an application. Such tailoring also helps to minimize the risk of inappropriate Entry interactions when changes in the requirements of the application require changes in its RTE.

"Conformance" to this catalogue should therefore never refer to the catalogue as a whole, but always to specific Entries or groups of Entries from the catalogue. Conformance to an Entry by an implementation means that the implementation fully supports the required semantics described in the CIFO Entry. This includes any requirements on interactions with other Entries.

By the same token, an implementation that includes "more" of the Entries from this catalogue should not automatically be judged to be a "better" implementation.

Furthermore, it is the ARTEWG's goal to define a validation test suite and procedures to certify such conformance. In a similar way in which the validation procedures for Ada disallow subsets or supersets of the language, subsets or supersets of individual CIFO Entries would be prohibited by the CIFO validation test suite and procedures.

It is recognized that some hardware/operating system combinations may be unable to completely support all components of an individual CIFO Entry. There are also situations in which an implementation, while fully supporting the semantics of a CIFO Entry, may not be able to implement it with the exact syntax documented here. Such partial or variant implementations are ***not*** considered to conform to CIFO Release 3.0. Both of these issues will be addressed in the next release of the document.

Disclaimer

The CIFO is the result of a strong effort by a number of qualified volunteers. However, it has not had the amount of rigorous outside review that is given to a formal standard, nor has it been through a balloting process as rigorous as that of a formal standard. Furthermore, ARTEWG has endeavored to release this version quickly, so that it can be relevant before culmination of the Ada9X effort.

ARTEWG is proud of our efforts to make this version of the CIFO as usable as possible. We are convinced that this version is a significant improvement over previous releases. However, **users are warned that semantic ambiguities may remain.**

Rationale

As implementation-dependent features of the Ada Runtime Environment (RTE) emerge, a divergence is developing in syntactic and semantic details. Such features are to be widely used and it is desirable, in order to derive the maximum benefits from the choice of Ada as the preferred high order language, that there be conventions about the interfaces providing access to these features.

Common interfaces are clearly needed to make the development of high-quality software practical for the full range of applications that Ada was intended to serve, especially the domain of embedded realtime systems. This *Catalogue of Interface Features and Options* is being developed to promote a commonality with respect to such implementation dependencies, to promote reusability and transportability of Ada program components, and to improve the performance of those components. Major issues such as process scheduling, debugging, and distributed processing are involved. The following is a discussion of some of the major areas of concern. While this discussion is not complete, it does represent a significant number of the issues and points out the need for the common set of conventions proposed in subsequent sections of the document.

Control of Memory Management

Memory management functions provided by the RTE are often critical to the operation of realtime applications. The LRM does not require return of deallocated storage to an available pool, potentially allowing for exhaustion of storage resources. Furthermore, any RTE function not under control of the application program that can significantly impact performance can cause constrained applications to fail. Storage reclamation is such a function. However, limitations on resources may in some cases make such capabilities necessary. Thus, conventions providing application-level control over such RTE functions are desirable.

Distributed Systems

Computers are increasingly used in applications which require high reliability, because they impact life and property. The failure of such an application can be disastrous. By decentralizing the system architecture, systems can be built that not only do not have single points of failure, but that are fault-tolerant.

Furthermore, extensibility of performance and functionality is often vital. When the software system is designed with distribution as a design criterion, the resulting modularity provides a design that need not be radically changed for increases in processing power (for performance) or for the addition of new modules (for additional functionality). In a system intended to have a long, evolving life cycle, this is a major issue.

Multiprogramming

Experience has taught us that many large scale and realtime systems have requirements for multiprogramming support. By "multiprogramming", we mean that multiple, independent programs may coexist in some state of execution which is supported by the underlying RTE. Multiprogramming is not at all new, and is in fact a highly typical configuration in today's systems. One example is a critical application that requires changes to the software without stopping the software first. Another example is found in those systems with requirements for fail-soft operation where emergency conditions may result in the suspension and preemption of lower priority programs so that the system resources may be devoted to servicing the higher priority programs until the emergency is past.

A program represents an executable model of a solution to a problem. Because a large number of modern problems possess inherent parallelism, Ada's multitasking permits a natural model of the solution to be expressed. (By "multitasking", we mean that, within a single program, multiple threads of control may coexist in some state of execution which is supported by the underlying RTE.) Furthermore, an increasing number of large scale systems contain several subsystems, some of which may be real-time and some of which may not be. Some subsystems are critical to the point of having to run at all times, without stopping. Nevertheless, changes, upgrades and fixes to such software are unavoidable. As a collection of individual programs, such requirements may be met. There is often a desire to subcontract the subsystems to different contractors, each of which will address their assignment with an appropriate Ada program. For purposes of final verification, validation, and integration of the subsystems into a working system, it is desirable to retain the independence (and the accountability) of the individual programs. Similar benefits accrue during the maintenance and operation phase.

Application Access to the Machine

Certain applications must access special features of the target machine in order to meet requirements for performance, accuracy, functionality, and so on. These programs have inherently non-transportable components. To minimize the impact of this lack of transportability on both software and programmer, conventions for accessing typical features are desirable.

In some situations, hardware interrupts will be intolerable. The ability to disable and enable interrupts is often required of real-time systems. Also, there is the issue of supporting more sophisticated operations, such as selectively controlling specific interrupts. To minimize the dangers of RTE-conflicts that an ad-hoc approach would imply, an RTE-interface is proposed.

The LRM suggests that interrupt handlers be implemented by associating interrupts with task entries. The overhead of scheduling and dispatching a task in response to a hardware interrupt will at times be unacceptable in highly-constrained applications. Thus, a standardized interface specifying the minimization of interrupt handler latency is desirable.

Predictable Timing

The need often arises in real-time systems for a task (or tasks) to execute with a very small variation between a requested time of execution and the actual time of execution. Currently there is no way to guarantee bounds on the time of execution. This can cause serious problems, particularly during overload situations.

Control of Scheduling

In order for some real-time applications to meet their demanding performance requirements, explicit control over the scheduling and dispatching regime is often utilized. An interface specification providing periodic scheduling of tasks that are to be executed at a fixed frequency, with very small variation, is thus a very desirable capability in many Ada-based application solutions.

In real-time applications there are situations, such as load-shedding in response to failures, in which the design approach requires the ability to dynamically adjust task priorities. The predefined priority scheme in Ada is static. Thus, an interface to the RTE specifying such a capability is desirable.

Along with dynamic priorities, task identifiers beyond those of the inherent lexical identifiers may play a large part in defining conventions allowing control of the scheduling and dispatching decisions.

Fault Tolerance

Many applications require support for fault-tolerant operation. The approach may be based on a transparent detection-response scheme below the application, an explicit handling at the application level, or a combination of both. This involves not only hardware support, but also software support via the RTE. Common interfaces which provide applications with capabilities to detect (or be informed of) failures, and respond appropriately, at the application level and below, are desirable.

Pre-Elaboration

Many embedded systems cannot afford the time or memory for elaboration of all entities to occur at runtime. There is a need for a means of requesting that some or all of those entities that may be elaborated prior to runtime be so handled.

Critical Sections

Some runtime applications are required to ensure that certain sections of code be executed to completion without interruption by other application tasks. For example, there may be a timing constraint on the segment. A means of specifying this sort of control is needed.

Conclusion

Clearly there are a number of application domains in which common interfaces, if widely used, could promote the Ada language as a standard. This is especially evident for real-time, embedded systems, because of the inherent additional constraints on time and memory resources. Use of common interfaces established in this *Catalogue of Interface Features and Options* can minimize the impact of non-transportability for both software and people and thus promote the building of superior software systems. However, many of the interfaces in this Catalogue not only enhance reusability and transportability, but also provide essential functional capability for the application designer. By providing a common set of interfaces between the application designer and the RTE, use of the Ada language in applications for which it was designed will be significantly facilitated and enhanced.

Catalogue Entries

Introduction

The Entries which follow are not intended to be exhaustive, nor are thev intended to exclude other approaches to si⁻ilar problems. Rather, they represent an attempt to provide a common set of interfaces to programmers who need to make use of such capabilities.

It should be noted that only one of these proposed Entries, Synchronization Discipline, is a change to the language. This Entry addresses a well understood problem with the language that is being addressed in the Ada 9X process. Until the results of Ada 9X are available, this Entry is believed to be a feasible and viable contribution to a solution in the interim. The other Entries represent legal enhancements in the sense that the LRM prescribes various means of providing additional functionality at the language level (packages, pragmas, etc.). Thus these Entries do not alter the semantics of existing LRM-defined constructs. Instead, they represent Entries in a *Catalogue of Interface Features and Options* (CIFO), that may be used to tailor a set of Ada runtime services and resources for differing application environments.

The use of pragmas has been minimized for a number of reasons, primarily since a compiler can ignore them ithout informing the user. In contrast, packages, subprograms, etc. which are not supported must be rejected at compile-time. Furthermore, the proposed features, expressed as generics, packages, and subprograms, can, in the main, be implemented as independent RTE modules. Pragmas and attributes would absolutely require the intermediation of a cognizant compiler. However, when the semantics of a pragma are truly appropriate (or unavoidable), they will be specified.

The declarations for these Entries are intended to be available to users via an Ada program library. The implementations are logically unit Entries in the Runtime Library, . at may be implemented by compiler-emitted code or combinations of both. (See the ARTEWG document Framework for Describing Ada Runtime Environments for a detailed discussion of runtime system architectures and organizations.)

Changes from Release 2.0

Release 3.0 contains several new Entries believed to be of value to the Ada community. Release 3.0 also contains revisions to some of the Entries introduced in earlier releases. These revisions are the result of feedback from implementations and applications as well as new insights gained from ARTEWG participation in support of Ada 9X and other important programs. In addition, two new subtopics have been added to the documentation of each Entry: Interactions With Other CIFO Entries, and Changes From CIFO Release 2.0. This release also identifies some deleted Entries along with the rationale for their deletion.

The new Entries are:

Queuing Discipline

Priority Inheritance Discipline

Two Stage Task Suspension

Asynchronous Task Suspension

Synchronization Discipline

Resources

Events

Pulses

Buffers

Blackboards

Broadcasts

Barriers

Asynchronous Transfer of Control

Shared Locks

Access Values That Designate Static Objects

Passive Task Pragma

Unchecked Subprogram Invocation

Data Synchronization Pragma

Dynamic Storage Management.

The revised Entries are:

Task Identifiers

Synchronous and Asynchronous Task Scheduling

Dynamic Priorities

Time Critical Sections (Renamed from "Nonpreemptible Sections")

Abortion via Task Identifiers

Task Suspension (Renamed from "Controlling When a Task Executes")

Time Slicing

Mutually Exclusive Access to Shared Data (Supersedes "Generic Asynchronous Communication")

Signals (Supersedes the old Entry "Asynchronous Entry Calls")

Interrupt Management

Trivial Entries

Fast Interrupt Pragmas

Pre-Elaboration of Program Units.

The renamed Entries are:

Nonpreemptible Sections (Renamed to "Time Critical Sections")

Controlling When a Task Executes (Now called "Task Suspension").

The superseded Entries are those with functionality that has been substantially retained but is now provided in a different way. In addition, the original functionality may have been extended. The superseded Entries are:

Generic Asynchronous Communication (Now "Mutually Exclusive Access to Shared Data".)

Asynchronous Entry Calls (Now "Signals".)

The deleted Entries are:

Special Delays

Transmitting Task Identifiers Between Tasks.

Basic Entries

This section contains two CIFO entries that are used by many of the other CIFO entries. Neither of these two entries is very useful by itself. However, both supply a standard way for other CIFO entries to perform two important functions; (1) specify tasks, irrespective of their types and (2) specify queueing disciplines.

Task Identifiers

Issue

Several RTE extensions described in this catalogue require a means of specifying tasks as parameters to RTE procedures, where the Ada typing rules would not permit using the name of the task. In addition, in the writing of general schedulers and resource managers [V' .ings] there is need for a means of storing information about tasks in tables, and for a way of associating a storable identifier with each task, beyond the basic capabilities provided by the language.

Proposal

An implementation-defined package:

```
package TASK_IDS is

    type TASK_ID is private;

    TASK_IDS_CHECKED : constant BOOLEAN := <implementation-defined>;

    TASK_ID_ERROR, PASSIVE_TASK_ERROR : exception;

    function NULL_TASK return TASK_ID;
    function SELF return TASK_ID;
    function MASTER_TASK return TASK_ID;
    function CALLER return TASK_ID;
    function CALLABLE( T : TASK_ID ) return BOOLEAN;
    function TERMINATED( T : TASK_ID ) return BOOLEAN;

private
    type TASK_ID is <implementation-defined>;
end TASK_IDS;
```

This package provides the type TASK_ID and several functions for obtaining the identifier of a particular task and determining its status. Note that the main program has a task ID; it is the identifier of the environment task which called the main program (LRM 10.1 paragraph 8).

Task IDs are assumed to be unique across an Ada program's life time. If TASK_IDS_CHECKED is true then each task created by the program will be guaranteed to have a unique id. If an operation is performed on a task which is no longer in existence then the exception TASK_ID_ERROR will be raised. If TASK_IDS_CHECKED is false then no check will be performed. The assumption here is that the program either consists of a fixed set of tasks, or that the programmer has carried out the checking. It is not defined what happens if these assumptions are violated.

The function NULL_TASK returns the task ID of the null task.

The function SELF returns the task ID of whatever task calls it. If the main program calls SELF then the task ID of the environment task is returned. If the calling task is passive (see Passive Task Pragma CIFO Entry) then the exception PASSIVE_TASK_ERROR is raised.

The function MASTER_TASK returns the task ID of the task on which the task calling the function is directly dependent (see LRM 9.4 for a detail definition of dependent tasks and their masters). The MASTER_TASK of the main program in a uniprogramming system is the NULL_TASK. The MASTER_TASK of the main program in a multiprogramming system is implementation defined. The MASTER_TASK of library tasks is the environment task. The MASTER_TASK of the NULL_TASK is the NULL_TASK. If the MASTER_TASK of the calling task is passive then the exception PASSIVE_TASK_ERROR is raised.

The function CALLER, if called from within an accept, returns the task ID of the partner task. (The partner task is the task that made the entry call being accepted.) In the case of nested accepts CALLER returns the task ID of the partner in the innermost enclosing accept. If called from outside an accept, CALLER returns NULL_TASK. If the partner task is passive then the exception PASSIVE_TASK_ERROR is raised. An implementation may choose not to implement this function, or to raise TASK_ID_ERROR if for some reason (e.g. security) the implementation does not wish to return the task ID of the caller task. If a task entry is called by an "agent task" (see Signals CIFO Entry), the function CALLER returns the NULL_TASK.

The function CALLABLE(I) returns the value of the attribute 'CALLABLE for the task corresponding to the task ID I, and the function TERMINATED(I) returns the value of the attribute 'TERMINATED for the task corresponding to the task ID I. The NULL_TASK is always TERMINATED and never CALLABLE.

The exception TASK_ID_ERROR may also be raised by other packages which build upon the services provided here.

Discussion

This package provides a universal way of specifying tasks, irrespective of their types. It does potentially extend the "visibility" of tasks via the function CALLER in a way that might be dangerous. Therefore, an implementation may choose not to fully implement this function, as described above. By itself, this package is not very useful. Its chief purpose is to support other RTE interface features described in this catalogue. However, one can imagine a key-server which verifies that at most one key is allocated to each task, as sketched below:

```
task KEY_SERVER is
    entry REQUEST( K : out KEY );
    ...
end KEY_SERVER;
task BODY KEY_SERVER is
    use TASK_IDS,
    TK : KEY;
begin
    loop
        accept REQUEST( K : out KEY) do
            begin
                LOOKUP(CALLER,TK);
                if TK = NULL_KEY then
                    ASSIGN_KEY( CALLER, TK );
                end if;
                K := TK;
            exception
                when PASSIVE_TASK_ERROR =>
                    ...
            end;
        end REQUEST;
    end loop;
end KEY_SERVER;
```

Here we assume that LOOKUP(CALLER,TK) returns the key of CALLER in TK, if one has been assigned, and returns NULL_KEY otherwise. We also assume that ASSIGN_KEY(CALLER,TK) allocates a new key for caller, records the assignment in a table, and returns the key in TK.

Implementation Considerations

If unique task IDs are checked then implementing task IDs as a simple pointer to a task control block (TCB) may not be a sufficient implementation. A possible implementation would be a record; one component would be the TCB pointer; another would be a "key" integer field, to assure uniqueness of task IDs. This would require that memory that is once allocated to be a TCB can never be reused for any purpose other than being a TCB; thus, by storing the key in the TCB, any use of a task ID can be preceded by a check that the TCB is still valid for this task ID. More efficient storage implementations are possible.

TASK_ID is a private type to protect the CIFO implementation from inadvertent use of the TASK_ID value. This is the "contract" that CIFO provides to the application builder (non-CIFO builder). If CIFO builders need to know any implementation details of TASK_ID then an implementation must provide a separate contract for this package that includes the implementation details of the TASK_ID type.

When referencing a passive task, the action performed is implementation-defined. Several options are possible:

1) Raising an exception. This requires extra overhead on the part of the RTE to recognize when a task has called a passive task. An advantage of this approach is that a passive task will not perform operations while appearing to be a different task.

2) Returning the TASK_ID of the task that called the passive task. This approach has the advantage that the RTE need not update information when entering or exiting a passive task, and that application code can be guaranteed that the value returned by SELF will be a valid TASK_ID. A disad· antage is that a passive task can perform actions while appearing to be another task.

3) Returning NULL_TASK. This approach is really just a variant of option 1 that has the added disadvantage that the exception may not be caught until the TASK_ID value is later used in a call to another CIFO entry.

Interaction With Other CIFO Entries

In general this Entry is self-contained, although it assumes that tasks can be passive. Note that it is not possible to obtain a task ID for a passive task.

Other Entries may use task IDs and the interaction will be specified there.

Changes From The Previous Release

It has been made explicit that the main program has a TASK_ID and that all the defined functions can be applied to the main program. The function ENCLOSING_TASK has been deleted because it was found to be difficult to implement without compiler support. The function PARENT has been renamed MASTER_TASK to avoid confusion with other uses of the term parent. The MASTER_TASK of the main program has been defined to be the NULL_TASK in a uniprogramming system. The MASTER_TASK of the main program in a multiprogramming system is implementation defined. The MASTER_TASK of library tasks is defined to be the environment task.

The function ID_OF has also been deleted because it was not safe and allowed an ID to be obtained for a non-task object. All task IDs are unique and can be passed freely around the system; any protection mechanism must be built on top of task IDs. TASK_IDS_CHECKED has also been added to allow an implementation the flexibility to check or not to check for valid task IDs and to check for guaranteed uniqueness of task IDs.

The Entry now considers the possibility that tasks may be passive.

References:

[Wellings] Wellings, et.al., Ada Letters, January 1984, pp. 112-123.

Queuing Disciplines

Issue

Several RTE extensions described in this catalogue provide a means of blocking and scheduling a task. In general there are several queuing disciplines that can be used with these Entries.

Proposal

This Entry defines the type which will be used to indicate which discipline is required when instantiating other CIFO Entries.

```
package QUEUING_DISCIPLINE is

    type DISCIPLINE is (    ARBITRARY_QUEUING, FIFO_QUEUING,
                            PRIORITY_QUEUING, SPINNING   );

    UNSUPPORTED_DISCIPLINE : exception;

end QUEUING_DISCIPLINE;
```

If ARBITRARY_QUEUING is selected, the queuing discipline is implementation-defined. If FIFO_QUEUING is selected then queuing is on a first-in, first-out basis. If PRIORITY_QUEUING is selected queuing is determined by the priority of the tasks involved. If SPINNING is selected then there is no queuing. The exact spinning policy depends on the implementation, but an example might be for a multiprocessor system a task would continually attempt to get the service until the service was available, only releasing the processor if preempted.

With priority queuing, it may happen that a queue contains more than one task at the same priority. In this case, the "PRIORITY" discipline uses FIFO ordering. Note that a task that is already waiting in the queue may be given a different (higher or lower) priority. For such a task, changing the priority has an effect equivalent to removing the task from the queue and requeuing it at its new priority.

Other queuing disciplines may be added.

An implementation should never raise UNSUPPORTED_DISCIPLINE when ARBITRARY_QUEUING is selected.

Discussion

Arbitrary queuing should be selected if the user is unconcerned with how the queue is ordered. Spinning is intended to be implemented with fast mutual exclusion primitives and may be used in high performance multiprocessor systems.

We considered defining a mechanism by which the spin delay could be set. This allowed the implementation to support both uniprocessor and multiprocessor environments for spinning. Without a delay, the uniprocessor would spin forever since nobody else could get control to release the resource required. Within the spinning code, the spin delay (if non-zero) would be used in an Ada delay statement executed each "spin". Now the intention is that any CIFO Entry which supports the spinning discipline should set its own delay as it deems appropriate. This mechanism would be part of the individual CIFO Entry that supported SPINNING queuing discipline.

Interactions With Other CIFO Entries

In general this Entry is self contained but will be used by other Entries, for example Shared Locks. Their interaction with this Entry is defined with their Entry. Note that if dynamic priorities or priority inheritance is implemented in the CIFO, then queues may need to be reordered as task priorities change.

Changes From The Previous Release

This is a new CIFO Entry.

Scheduling Control

This collection of Entries is designed to implicitly control the execution of tasks in an application by manipulating the properties of tasks, or explicitly control execution by directly specifying the scheduling or dispatching decisions. The Entries are grouped as follows:

a. Synchronization Discipline, Priority Inheritance Discipline, and Dynamic Priorities.

b. Time Critical Sections

c. Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension

d. Synchronous and Asynchronous Task Scheduling and Time Slicing

e. Abort Via Task Identifiers.

The first group controls scheduling and dispatching by manipulating the priorities of tasks and how the priorities of those are used for dispatching executable programs, selection of waiting callers in task entry queues, and selection of alternatives in selective wait statements. Time Critical Sections are used to guarantee uninterrupted execution for a time critical sequence of statements. Group C Entries are designed to directly control the suspension and resumption of specific tasks for simple blocking for services like I/O, intertask communication and synchronization, and asynchronous suspension and resumption. The fourth group of Entries are designed to control the scheduling of tasks by defining the events or time these tasks will execute or terminate. Finally the Abort Via Task Identifiers is designed to provide the same semantics of the abort statement for tasks specified by their Task Identifiers.

Obviously these Entries provide semantics that have *very* pervasive effects on the execution of the application. Despite this they have been included in the CIFO because of the well founded requests for them from application builders and the unsafe and nonportable mechanisms that would be used in their place. They should be used judiciously, especially when several of them are used within the same application. ARTEWG has taken great effort to explain interactions that these Entries have with Ada features and among themselves to aid the user in the choice and application of these Entries. So in the words of Sgt. Philip Freemason Esterhouse, "Be careful out there!"

Several concepts concerning priorities and the preemptibility of sequences of statements are introduced and used throughout these Entries. Courtesy of the Ada 9X Project, the concept of "base" and "active" priority is used to explain the impact of priority on the execution of tasks. The base priority is given to a task at task creation or changed by the use of Dynamics Priorities CIFO Entry. The active priority is used by the runtime environment for dispatching and resource allocation. Generally speaking, the active priority is determined by the runtime environment and is the maximum of the base priority of the task and the active priorities of all tasks "waiting" on it. More details about their definitions and uses can be found in the Dynamics Priorities CIFO Entry. Throughout the text of the CIFO Entries whenever there is a need to specifically refer to base and active priorities then the text does so; otherwise the use of priority without qualification is implying the use of active priority. Three notions of preemptibility of sequence of statements are introduced in the Entries in this section, namely time critical sections, nondispatchable sections, and holdable section. As it name implies, time critical sections (defined in the Time Critical Sections CIFO Entry) of statements are statements of code meant to be executed in a critical amount of time, so that no dispatching and only runtime environment critical interrupts are performed. Nondispatchable sections (defined in the Task Suspension CIFO Entry) of statements are designed to be atomically executed without unnecessary preemption from other tasks. Holdable sections (defined in the Asynchronous Task Suspension CIFO Entry) of statements are statements of code that the application builder has determined asynchronous task suspension is safe and can be permitted. Avoid mixing the use of these sections within an application.

Synchronous and Asynchronous Task Scheduling

Issue

The common mode of task scheduling in realtime systems is via explicit synchronous (cyclic) and asynchronous scheduling.

The Ada language definition does not support explicit task scheduling. Rather, it defines only the most rudimentary level of task control, leaving it up to the user to devise his own methodology for implementing higher abstractions of task scheduling. This proposal defines a scheduling package to implement a synchronous and asynchronous scheduling capability. . .ıe proposal is based on the process scheduling paradigm used with the HAL/S language currently in use in the Shuttle avionics software.

Proposal

The package supports the explicit assertion of scheduling requirements for each task via a procedure call to the runtime environment. The procedure calls would be legal from within the task to be scheduled and from outside the task from anywhere that the task id could be obtain... through the TASK_IDS package defined elsewhere. The schedule package would be defined as follows:

```
with TASK_IDS, DYNAMIC_PRIORITIES, EVENTS, CALENDAR;

package SCHEDULER is

    TASK_OVERRUN : constant EVENTS.EVENT := EVENTS.CREATE;

    type TASK_PRIORITIES is (CURRENT, ALTERED);

    type TASK_OVERRUNS is (IGNORE, REPORT);

    type TASK_INITIATIONS is
        (IMMEDIATELY, AT_TIME, AFTER_DELAY, ON_EVENT_SET, ON_EVENT_RESET);

    type TASK_REPETITIONS is (NONE, REPEAT_EVERY, REPEAT_AFTER);

    tvpe TASK_COMPLETIONS is ( NONE, AT_TIME, ON_EVENT_SET, ON_EVENT_RESET );

    type INITIATION_INFO( INITIATION : TASK_INITIATIONS := IMMEDIATELY ) is
        record
            case INITIATION is
                when IMMEDIATELY => null;
                when AT_TIME => T : CALENDAR.TIME;
                when AFTER_DELAY => D : DURATION;
                when ON_EVENT_SET | ON_EVENT_RESET =>
                    E : EVENTS.EVENT := EVENTS.CREATE;
            end case;
        end record;
```

```
type REPETITION_INFO( REPETITION : TASK_REPETITIONS := NONE ) is
    record
        case REPETITION is
            when NONE => null;
            when REPEAT_EVERY | REPEAT_AFTER => D : DURATION;
        end case;
    end record;

type COMPLETION_INFO( COMPLETION : TASK_COMPLETIONS := NONE ) is
    record
        case COMPLETION is
            when NONE => null;
            when AT_TIME => T : CALENDAR.TIME;
            when ON_EVENT_SET | ON_EVENT_RESET =>
                E : EVENTS.EVENT := EVENTS.CREATE;
        end case;
    end record;

type PRIORITY_INFO( PRIO : TASK_PRIORITIES := CURRENT ) is
    record
        case PRIO is
            when CURRENT => null;
            when ALTERED => P : DYNAMIC_PRIORITIES.DYNAMIC_PRIORITY;
        end case;
    end record;

type OVERRUN_INFO( OVERRUN : TASK_OVERRUNS := IGNORE ) is
    record
        case OVERRUN is
            when IGNORE => null;
            when REPORT =>
                OVERRUN_EVENT : EVENTS.EVENT := EVENTS.CREATE;
        end case;
    end record;

IMMEDIATE         : constant INITIATION_INFO := (INITIATION => IMMEDIATELY);
NO_REPETITION     : constant REPETITION_INFO := (REPETITION => NONE);
NO_COMPLETION     : constant COMPLETION_INFO := (COMPLETION => NONE);
CURRENT_PRIORITY  : constant PRIORITY_INFO := (PRIO => CURRENT);
IGNORE_OVERRUNS   : constant OVERRUN_INFO := (OVERRUN => IGNORE);


procedure SCHEDULE
        (   SCHEDULED_TASK    : in TASK_IDS.TASK_ID;
            INITIATION        : in INITIATION_INFO;
            REPETITION        : in REPETITION_INFO;
            COMPLETION        : in COMPLETION_INFO := NO_COMPLETION;
            PRIORITY          : in PRIORITY_INFO := CURRENT_PRIORITY;
            OVERRUNS          : in OVERRUN_INFO := IGNORE_OVERRUNS );
    procedure WAIT_FOR_SCHEDULE
          (   RELEASE_AFTER_DESCHEDULE  : in BOOLEAN := False;
              DESCHEDULED               : out BOOLEAN );

    procedure DESCHEDULE
          (   SCHEDULED_TASK    : in TASK_IDS.TASK_ID;
              STOP_TASK         : in BOOLEAN := False );

    function IS_DESCHEDULED( SCHEDULED_TASK : in TASK_IDS.TASK_ID )
          return BOOLEAN;

    INVALID_SCHEDULE : exception;

end SCHEDULER;
```

The above definition requires some specific semantic rules to fully explain the capabilities and limitations of the package.

1. The SCHEDULE procedure causes the RTE to permit the task referenced by the call to return from a call to WAIT_FOR_SCHEDULE whenever the conditions expressed by its parameters are met. That is, whenever a task calls WAIT_FOR_SCHEDULE, it begins to wait; it will be released, so that it is eligible for execution and can return from the call, as soon as the conditions expressed by the last call to SCHEDULE for that task have been met. In the following discussion then, the terms "wait" and "release" are used in precisely this sense, only to refer to waiting for permission to return from a call to WAIT_FOR_SCHEDULE and for the granting of permission to return from such a call.

A task is descheduled in one of two ways: the criteria for the task to complete (as specified in a call to SCHEDULE) have become true, or the task becomes descheduled via a call to procedure DESCHEDULE.

The procedure SCHEDULE with a TASK_ID task T as an argument may be called from anywhere that TASK_ID is valid. In addition, any number of calls may be made to SCHEDULE. The scheduling attributes provided by the last call to SCHEDULE supersede all previous calls. If TASK_ID denotes an abnormal task, a terminated task, or denotes no task, the exception INVALID_SCHEDULE is raised.

1a. SCHEDULED_TASK - The argument SCHEDULED_TASK identifies the task to which this scheduling call applies.

1b. INITIATION - This argument specifies a condition for initiation, i.e., first release, of the task. Once a task has initiated execution, these arguments have no effect on the task's execution. The possible initiation conditions are specified in the enumerated type TASK_INITIATIONS. The effect of each is provided below (note that TIME_OF_SCHEDULE represents the value of CLOCK at the time of the call to SCHEDULE):

IMMEDIATELY - If the specified task is waiting at a call to WAIT_FOR_SCHEDULE, it is immediately released. If not, it gets immediately released on that task's next call to WAIT_FOR_SCHEDULE.

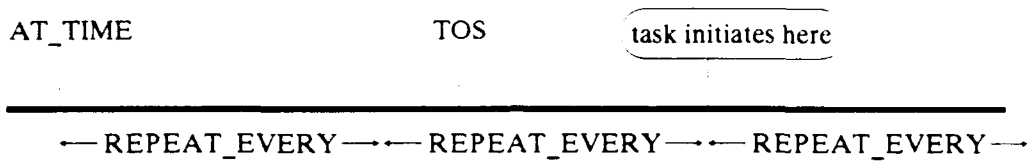
AT_TIME - This argument specifies a clock time at which the task should be next released. This time is affected by the repetition specification: If REPEAT_EVERY is not specified, the task is released as soon as CLOCK >= AT_TIME.

If REPEAT_EVERY is specified and AT_TIME >= TIME_OF_SCHEDULE, then the task is released at

AT_TIME. If AT_TIME < TIME_OF_SCHEDULE, then the task is released at the following time:

TOS + REPEAT_EVERY - ((TOS - AT_TIME) modulo REPEAT_EVERY)

(where TOS is TIME_OF_SCHEDULE). This has the effect of establishing fixed release points as if the task is first released at AT_TIME. In this way, subsequent releases are still relative to AT_TIME.



AFTER_DELAY - This argument specifies a duration of time the task's initiation is delayed from TIME_OF_SCHEDULE. The effect of this call is as if a delay statement is inserted at the point of the WAIT_FOR_SCHEDULE, and removed from the code as soon as the delay expires. Note, though, that the delay duration is based on the call to SCHEDULE, not on the call to WAIT_FOR_SCHEDULE. The accuracy of the delay statement is at least the same as for a normal delay statement.

ON_EVENT_SET - This argument specifies that the task is to be released as soon as the expression given for ON_EVENT_SET becomes true. If it is already true at the time the task begins to wait following a call to WAIT_FOR_SCHEDULE, it is released immediately.

ON_EVENT_RESET - This argument specifies that the task is to be released as soon as the expression given for ON_EVENT_RESET becomes false. If it is already false at the time the task begins to wait following a call to WAIT_FOR_SCHEDULE, it is released immediately.

1c. REPETITION - This argument specifies a condition for the repetitive release of the task. The possible repetition conditions are specified in the enumerated type TASK_REPETITIONS. Their results are given below.

NONE - The task is never released after its initial release (i.e., the one resulting from the INITIATION conditions). If the task calls WAIT_FOR_SCHEDULE, then it can not return from that wait until another call to SCHEDULE for this task changes its scheduling conditions. If a task is specified with no repetition, then the specification of that task's completion has no effect on the task.

REPEAT_EVERY - This argument specifies a delay to be imposed between the time a task may be released from the initiated time, in absolute terms, i.e., at fixed points in time. The effect begins the next time the task is released and continues thereafter until the task is rescheduled. Thus, if the task is next released at time T1, and calls WAIT_FOR_SCHEDULE at a later time T2, then it will be released at the earliest time T1 + K*(REPEAT_EVERY) >= T2. Note that this differs from REPEAT_AFTER, where the interval between successive releases is relative to time T2. For this argument to work effectively requires that the Initiation argument specify a discriminant of AT_TIME.

REPEAT_AFTER - This argument specifies the duration of time the task waits to be released since it called WAIT_FOR_SCHEDULE. It applies to the second and subsequent occasions when the task is released after this call to SCHEDULE. (Any delay required for the task's first release is specified by the INITIATION component AFTER_DELAY.) The effect is similar to inserting a delay statement at the places of the relevant calls to WAIT_FOR_SCHEDULE.

Thus, if the task is released, and starts to wait (by calling WAIT_FOR_SCHEDULE) again at time T, it will be released at time T + REPEAT_AFTER. To repeatedly release a task without any delay, a zero value for REPEAT_AFTER is specified.

1d. COMPLETION - This argument specifies the condition under which the execution of the task is completed, i.e., after which the task will no longer be released. If the condition is met at the same time that the initiation criteria is met, the task becomes completed. The possible conditions are specified in the enumeration type TASK_COMPLETIONS. Their effects are given below:

NONE - There are no conditions under which this task will complete. This task is sometimes called an "infinite" task.

AT_TIME - This argument specifies the time after which this task will not be released.

ON_EVENT_SET - This argument specifies an event condition after which the task completes. The task will not again be released when this event condition is set to "up", unless it is rescheduled. The argument ON_EVENT_SET is evaluated once when SCHEDULE is called, and is reevaluated on subsequent event occurrences that may changes its value.

ON_EVENT_RESET - This argument specifies an event condition after which the task completes. The task will not again be released when this event condition is set to "down", unless it is rescheduled. The argument ON_EVENT_RESET is evaluated once when SCHEDULE is called, and is reevaluated on subsequent event occurrences that may changes its value.

NOTE: The completion conditions specified in this section are separate from the completion conditions for Ada tasks and are intended for use in ending the scheduling of a periodic task. They do not place any restrictions on the use of the Ada terminate option or abort feature. A terminated task cannot be scheduled.

1e. PRIORITY - This argument specifies whether the task keepS its current priority, or is assigned a new priority. The options for selecting the priority are specified in the enumerated type TASK_PRIORITIES. The effect of each is provided below:

CURRENT - This specifies that the task retains its current priority.

ALTERED - This specifies a new priority for the task, found in the PRIORITY parameter.

1f. OVERRUNS - This argument provides a mechanism for setting events when a task misses its timing deadline, or "overruns". A task overruns whenever there is an opportunity to release the task according to the specified scheduling conditions, but the task cannot be released because it is not waiting on a call to WAIT_FOR_SCHEDULE. Task overrun applies only to the REPEAT_EVERY scheduling condition specified by the REPETITIONS_INFO record. If a task misses an opportunity to be released on an event, an overrun does not occur. By "opportunity to release the task", we mean a time T such that if the task were waiting immediately before T it would have been released at time T. The options for reporting overruns are specified in the enumerated type TASK_OVERRUNS, which have the effect shown below:

IGNORE - This argument specifies that the RTE can not set any events when this task misses its scheduled deadline.

REPORT - This argument enables the overrun event mechanism for this task. If the specified task misses its deadline, then the RTE sets the following events: the event TASK_OVERRUN, and the event specified by OVERRUN_EVENT. After a call to SCHEDULE (for this task) or a subsequent call to WAIT_FOR_SCHEDULE (by this task), both TASK_OVERRUN and the event specified in OVERRUN_EVENT are reset.

Note that the procedure SCHEDULE provides two options for handling a task overrun. One option is to set only event TASK_OVERRUN. This is a global event that can be set by any task which uses the

REPORT_OVERRUN option. It therefore is useful to an external service that is monitoring overruns of many tasks, but is not interested in any particular task (and thus would need only to monitor one event).

Alternatively, a second event, associated with this particular task, can be set in addition to TASK_OVERRUN. This provides for overrun detection of a particular task. Detection of an event is provided through package EVENTS. These events must be explicitly reset after their use.

2. WAIT_FOR_SCHEDULE causes the RTE to hold the execution of the calling task until the conditions, as specified in the last procedure SCHEDULE call, allow the task to be released. Each task that is scheduled for execution by the RTE through a SCHEDULE call should have at least one instance of a corresponding call to WAIT_FOR_SCHEDULE. Normally, it appears immediately at the top of the task or immediately after a call to SCHEDULE. However, it can appear anywhere within the task.

A call to WAIT_FOR_SCHEDULE resets the calling task's overrun event(s) which had become set (if any).

There are two parameter to this procedure:

2a. RELEASE_AFTER_DESCHEDULE - This argument takes effect after the task has been descheduled by a call to DESCHEDULE. After a task has been descheduled and has had its final release, a call to WAIT_FOR_SCHEDULE can have two possible effects: (1) indefinitely suspend the caller, or (2) release it immediately. If RELEASE_AFTER_DESCHEDULE = true, a call to WAIT_FOR_SCHEDULE immediately releases the calling task. If it is false, the caller can not be released. If the task completes normally via the COMPLETION_INFO criteria, this argument has no effect. A call to WAIT_FOR_SCHEDULE will always block in that case.

This feature allows a task to keep its looping structure intact after being descheduled and the use of WAIT_FOR_SCHEDULE is no longer desired.

2b. DESCHEDULED - This parameter returns a Boolean value indicating whether the calling task has been descheduled. A value of "true" is returned if the task has been descheduled via a call to DESCHEDULE.

3. When called, procedure DESCHEDULE removes all scheduling conditions for the task, and the task is executed as if there had been no call to SCHEDULE. The scheduling parameters for this task, as they were established by procedure SCHEDULE, lose their effect immediately. If TASK_ID denotes an abnormal task, a terminated task, or denotes no task, the exception INVALID_SCHEDULE is raised.

The STOP_TASK argument allows the DESCHEDULE routine to force the task to immediately block on a call to WAIT_FOR_SCHEDULE. This argument will not allow the release of the task even if RELEASE_AFTER_DESCHEDULE = true.

When STOP_TASK = true, the task will never be released from a call to WAIT_FOR_SCHEDULE.

When the task has been descheduled and STOP_TASK = false, the task will be released once more on the next call to WAIT_FOR_SCHEDULE. If the task is already at a WAIT_FOR_SCHEDULE when the descheduling takes place, it is immediately released.

Note that descheduling a task does not necessarily preclude its subsequent execution. For example, if the task is ready to execute, but cannot because a higher priority task is running, descheduling the task will not prevent its execution once it has the opportunity. It should also be noted that since the normal Ada tasking rules apply, the task, once released, may suspend itself by issuing a call to delay, or may even reschedule itself (via a call to SCHEDULE).

4. When called, function IS_DESCHEDULED returns a Boolean value indicating whether the specified task has been descheduled. A task which has never been scheduled using the SCHEDULE procedure will return false. A value of true will be returned if either the task has been descheduled via a call to DESCHEDULE, or the task has completed its scheduled execution, as established by the completion criteria in procedure

SCHEDULE, or if REPETITION_INFO is NONE. If the task subsequently is rescheduled, via another call to procedure SCHEDULE, this function will return false.

5. The objects CURRENT_PRIORITY, IGNORE_OVERRUNS, IMMEDIATE, NO_REPETITION and NO_COMPLETION provide the application with default record objects for use in the SCHEDULE procedure.

Discussion

This entry is designed to employ a well-established and well-tested paradigm for handling loosely coupled tasks with a high level of periodicity. In this approach, the "main activity" tasks of interest are executed in regular intervals or at the occurrence of specific events. It is these tasks that would be configured with the use of this entry. Furthermore not all of the tasks in the application are necessarily configured by this entry. These tasks are designed to support these "main activity" tasks by handling interrupts, acting as servers or monitors, or executing in the background in a time sliced manner. These support tasks are better handled outside of this entry with native Ada or with other CIFO Entries such as Fast Interrupt Pragma, Resources, or Time Slicing.

In order that individual tasks may be monitored for task overrun, the SCHEDULE routine takes a parameter OVERRUN_EVENT, of type EVENT, that will be associated with this task's overrun. When the task overruns, both TASK_OVERRUN and event OVERRUN_EVENT will be set.

Use of this approach is shown in the schedule management task below:

```
task body TASK_MANAGEMENT_1 is
    TASK_A_OVERRUN : EVENT;
    TASK_B_OVERRUN : EVENT;
    ...
begin
    ...
    SCHEDULE( ID_LIST(TASK_A), ..., (REPORT,TASK_A_OVERRUN) );
    SCHEDULE( ID_LIST(TASK_B), ..., (REPORT,TASK_B_OVERRUN) );
    ...
    -- after detecting that TASK_OVERRUN has become set
    if EVENTS.STATE( TASK_A_OVERRUN ) = UP then
        DESCHEDULE( ID_LIST(TASK_A) );
    elsif EVENTS.STATE( TASK_B_OVERRUN ) = UP then
        null;
    end if;
end TASK_MANAGEMENT_1;
```

The user supplies an event to SCHEDULE, rather than have the procedure return a referenced event, to allow the user to create classes of events (the same event can be used for more than one task). Such a class could represent the overruns of a number of tasks.

Interactions With Other CIFO Entries

Usage with other CIFO scheduling control Entries, such as Task Suspension, Two Stage Task Suspension, Asynchronous Task Suspension and Time Slicing may result in indeterminate or erroneous results. Effects similar to these CIFO Entries can be accomplished with the judicious use of DESCHEDULE and SCHEDULE procedure calls.

If Priority Inheritance Discipline is in effect, the result of specifying an altered priority is in accordance with that discipline.

Fast Interrupt Pragmas: A task enclosing a Medium Fast Interrupt entry may be affected by the services of this proposal in the usual way. Such a task may also call these services in the usual way, but not from within the interrupt rendezvous. The services of this CIFO Entry can affect the enclosing task of a Fast Interrupt entry, but not the Fast Interrupt rendezvous. In general, the set of tasks that are managed by this CIFO Entry should not include the tasks with Fast Interrupt entries.

Changes From The Previous Release

The EVENT type is now provided by a separate EVENTS package in CIFO entry EVENTS.

Argument PRIORITY in procedure SCHEDULE was changed to record PRIORITY_INFO to easily allow keeping the current priority of the task. Since this will likely be common in use, the default value leaves the priority at its current value.

In order to better provide detection of a task overrun both within and without the task itself, exception TASK_OVERRUN has been replaced by a mechanism to set events when a task overruns (and was scheduled to report overruns). The global event TASK_OVERRUN allows a scheduling management task to monitor many tasks for an overrun, since each task for which reporting is requested will set this event when it overruns.

The previous version of type INITIATION_INFO allowed a task to initiate on an event being set; this has been enhanced to allow a task to initiate on an event's reset. The semantics of AT_TIME have been rewritten to bring this description in line with that of the HAL/S semantics; in particular, if AT_TIME > CLOCK, then the task should be initiated at AT_TIME. This description was also deemed clearer in meaning.

The component names of type COMPLETION_INFO have been changed to reflect their use as criteria for completion, rather than in terms of repetition.

Procedure WAIT_FOR_SCHEDULE has acquired a parameter RELEASE_AFTER_DESCHEDULE to allow a task to continue to execute its loop, as coded, but without having to suspend a call to WAIT_FOR_SCHEDULE after it has been descheduled. Use of the default value for this argument is identical to use of the WAIT_FOR_SCHEDULE procedure in version 2.0.

Procedure DESCHEDULE now has an option to keep the descheduled task from being released from a call to WAIT_FOR_SCHEDULE. Normal use of DESCHEDULE allows a final release from a call to WAIT_FOR_SCHEDULE in order to allow the task to take appropriate action on its descheduling. This feature has been added to prevent an undesired extra cycle of execution by the descheduled task.

A task can find out if it has been descheduled by making a call to IS_DESCHEDULED. This is provided as an additional function, rather than a parameter of WAIT_FOR_SCHEDULE, because more than one task may want to know if a certain task has been descheduled.

Successive calls to procedure SCHEDULE for some task supersede all previous calls.

The objects CURRENT_PRIORITY, IGNORE_OVERRUNS, IMMEDIATE, NO_REPETITION, and NO_COMPLETION are provided so that the user need not create these common objects at each place the SCHEDULE procedure is called.

Priority Inheritance Discipline

Issue

Some applications require software technology methods that can be used to achieve analyzable and predictable system behavior. One such method, Rate Monotonic Scheduling (RMS) [Liu73] uses the dynamic preemptive nature of the Ada tasking and task communication model and requires certain behavior of specific implementation dependent portions of the Ada language. Namely, all unbounded priority inversion (i.e., unbounded blocking of a high priority task by the execution of a lower priority task) must be eliminated. This may occur in implementation dependent areas of the language; in open alternatives of the select statement, in entry calls to an accept body that may result in FIFO queuing, and when a server task is executing outside its rendezvous [Sha89] and is blocking a higher priority task.

Unbounded priority inversion, that may be caused when no priority discipline is in effect on open alternatives of the select statement and on entry queues, can be eliminated using the Synchronization Discipline CIFO Entry to select the appropriate queuing discipline. Note that though a global priority discipline is intended for both entry queues and open select alternatives when the PRIORITY_INHERITANCE_DISCIPLINE is in effect, it is not necessary but the effect of priority inheritance will be extremely limited.

Proposal

This Entry affects both "static" priorities assigned via pragma PRIORITY, as well as dynamic priorities assigned via the Dynamic Priorities CIFO Entry.

Two mechanisms are proposed for specifying priority inheritance discipline:

```
package PRIORITY_INHERITANCE_DISCIPLINE is
    procedure SET_PRIORITY_INHERITANCE_CRITERIA;
    procedure RESET_PRIORITY_INHERITANCE_CRITERIA;
end PRIORITY_INHERITANCE_DISCIPLINE;
```

or

```
pragma SET_PRIORITY_INHERITANCE_CRITERIA;
```

The properties of the procedures in this package are as follows:

```
procedure SET_PRIORITY_INHERITANCE_CRITERIA;
```

When this procedure is in effect, all tasks in the program have the following properties:

1. A task's (active) priority is at least as high as the highest priority among all the tasks currently suspended in ANY of its entry queues (including closed alternatives). If a new task of higher (active) priority arrives in any entry queue of this task, the priority of the task is elevated to that of the newly arrived task.

Tasks suspended on a single entry queue are serviced according to the queuing discipline in effect for that queue. If the queuing discipline has not been changed (e.g., using the Synchronization Discipline CIFO Entry) then the suspended tasks are processed in FIFO order. The queuing discipline for one or more of the entry queues may be a priority discipline (e.g., established by SET_ENTRY_CRITERIA or SET_GLOBAL_ENTRY_CRITERIA capabilities from the Synchronization Discipline CIFO Entry).

A task may be removed from any of the entry queues because it is aborted or because the delay of a timed entry call expires as specified by the LRM. In this case, the (active) priority of the task owning the entry is adjusted accordingly if necessary.

2. A rendezvous is executed at the higher of the two (active) priorities of the tasks engaged in the rendezvous as specified by the LRM. If the priority of one or both of these tasks change while the rendezvous is in progress, the priority of the rendezvous is adjusted accordingly, if necessary. For example, task A making the entry call has an active priority of 10 and the accepting task B has a an active of priority of 20. If, during the rendezvous, a third task C with an active priority of 30 makes an entry call to task A, then the active priority of task A is temporarily raised to 30 and during the rendezvous the accepting task executes for the remainder of the rendezvous at priority 30 (barring any further changes). On the other hand, if the active priority of Task A is changed to 15 during the rendezvous, the priority of the accepting task remains unchanged as it is still executing at the higher of the two task's active priorities (i.e., 20).

An implementation may choose not to support this priority inheritance to any nested rendezvous, but it must document the cases in which priority inheritance is not supported.

3. Any newly created task executes its activation at a priority not lower than that of the task whose execution created the new task. This is a consequence of the binding interpretation accepted by ISO/WG9 and the ARG.

4. Whenever a master completes, all the tasks that depend directly or indirectly on this master have their priorities elevated to at least the level of the task under whose control the completing master executes.

5. Whenever the priority of a task changes, either through the SET_PRIORITY procedure of the DYNAMIC_PRIORITIES package or as a consequence of the rules 1-4 above, the priorities of other tasks are adjusted to satisfy rules 1-4. This adjustment takes effect immediately (or at least no later than the next dispatching point of all processors) and applies both to the raising and lowering of priorities.

6. Priority inheritance (i.e., Rules 1, 2, and 5) is also applied when tasks are competing for other resources and it is sensible to do so. If a task is waiting for a shared lock (see the Shared Locks CIFO Entry) held by a lower priority task then the task holding the lock has its priority elevated to that of the waiting task. When SET_PRIORITY_INHERITANCE_CRITERIA is in effect then the priority inheritance rules apply to the Resources, Buffers, Shared Locks, and Mutually Exclusive Access to Shared Data CIFO Entries.

Note: SET_PRIORITY_INHERITANCE_CRITERIA is intended to be used in conjunction with the priority discipline for both SET_GLOBAL_ENTRY_CRITERIA and SET_GLOBAL_SELECT_CRITERIA (see the Synchronization Discipline CIFO Entry). This allows such approaches as the Rate Monotonic Theory to be used. System behavior for other combinations used in conjunction with SET_PRIORITY_CRITERIA is currently not well understood.

```
procedure RESET_PRIORITY_INHERITANCE_CRITERIA;
```

This procedure has the effect of having the processing of tasks by the runtime environment be exactly as it was prior to the invocation of SET_PRIORITY_INHERITANCE_CRITERIA. If SET_PRIORITY_INHERITANCE_CRITERIA has not previously been invoked then RESET_PRIORITY_INHERITANCE_CRITERIA has no effect.

Alternative Proposal

An equally important interface to the functions characterized in this CIFO Entry is the pragma. While the procedural interface does not necessarily require modifications to the compiler, it does not permit some types of optimizations. A pragma interface would permit early detection and perhaps allow additional optimizations to be performed.

In some situations these additional optimizations may be necessary to meet efficiency requirements. In others changing the compiler may not be a viable solution. Therefore, this CIFO Entry defines an alternative solution. If either package PRIORITY_INHERITANCE_DISCIPLINE or the pragma defined below is supported, then this CIFO Entry is satisfied. The following pragma supports this CIFO Entry.

```
        pragma SET_PRIORITY_INHERITANCE_CRITERIA;
```

This pragma has the properties as described for the SET_PRIORITY_INHERITANCE_CRITERIA procedure above. This pragma must be located in the main procedure specification or the declarative part of the main procedure in the event that the main procedure does not have a specification. For obvious reasons pragma RESET_PRIORITY_INHERITANCE_CRITERIA is not defined.

Discussion

This package provides the application with the ability to specify a priority inheritance discipline to be in effect for the application program. The intention is to use this CIFO Entry in conjunction with the Synchronization Discipline CIFO Entry to eliminate all unbounded priority inversion, permitting the application use of analyzable and predictable analysis techniques such as the Rate Monotonic Scheduling (RMS) theory. Specifically, SET_GLOBAL_ENTRY_CRITERIA and SET_GLOBAL_SELECT_CRITERIA (from the Synchronization Discipline CIFO Entry) are used to select the priority discipline for select alternatives and entry queues. SET_PRIORITY_INHERITANCE_CRITERIA, from this Entry, is utilized to eliminate further unbounded priority inversions that can occur in a task executing outside its rendezvous (i.e., accept bodies).

Interactions With Other CIFO Entries

Queuing_Discipline: As noted in the discussion, one intended use of Priority Inheritance utilizes Priority Queuing. This creates a dependency on the Queuing Discipline CIFO Entry.

Time Slicing: Time Slicing and Priority Inheritance are generally conflicting strategies for scheduling and dispatching tasks. Care should be taken to keep tasks that need to be timed sliced separated from tasks that need priority inheritance.

Synchronization Discipline: The Priority Inheritance Discipline CIFO Entry expects that global priority discipline be specified for entry queues and select alternatives

If Priority Inheritance is in effect, it affects not only entry queues, but also all queues for CIFO Entries that use queues. This includes Resources, Buffers, Shared Locks, and Mutual Exclusive Access to Shared Data CIFO Entries.

Changes From The Previous Release

This is a new CIFO Entry.

References

[ARTEWG87] "Catalogue of Ada Runtime Implementation Dependencies",ACM Special Interest Group on Ada, Ada Runtime Environment Working Group, (1987).

[Borger89] "Implementing Priority Inheritance Algorithms in an Ada Runtime System", Mark W. Borger and Ragunathan Rajkumar, Technical Report, CMU/SEI-89-TR-15, (1989)

[Liu73] "Scheduling Algorithms for Multiprogramming in Hard Real Time Environments", Liu, C. L. and

Layland J. W., JACM 20 (1):46-61, (1973).

[Sha88] "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", Lui Sha, Ragunathan Rajkumar and John P. Lehoczky, Technical Report, CMU-CS-87-181, (1987)

[Sha89] "Real-Time Scheduling Theory and Ada", Sha, L. and Goodenough, J., Technical Report, SEI, CMU, (1989)

Dynamic Priorities

Issue

The minimal priority scheme defined by the LRM provides static priorities only. Many applications require a more dynamic priority assignment capability. For instance, degraded operation implies that the work performed in some task may be much less important than in normal operation, and should be assigned a less urgent priority (if retained at all).

Proposal

This proposal consists of a package and an associated pragma:

```
with TASK_IDS;
package DYNAMIC_PRIORITIES is

    subtype DYNAMIC_PRIORITY is INTEGER range 0 .. <31 or greater>;

    procedure SET_DYNAMIC_PRIORITY( OF_TASK   : in TASK_IDS.TASK_ID;
                                    TO        : in DYNAMIC_PRIORITY );

    function DYNAMIC_PRIORITY_OF( THE_TASK : TASK_IDS.TASK_ID )
        return DYNAMIC_PRIORITY;

    PRIORITY_CHANGE_NOT_ALLOWED : exception;

end DYNAMIC_PRIORITIES;


pragma INITIAL_PRIORITY( <static_expression> );
```

The package DYNAMIC_PRIORITIES provides a capability to assign priorities to tasks dynamically. The DYNAMIC_PRIORITY subtype is defined independently of the static Ada priority. If a task specification contains pragma PRIORITY, then this package cannot be used to modify its priority and any calls to the SET_DYNAMIC_PRIORITY procedure will raise PRIORITY_CHANGE_NOT_ALLOWED. A new pragma, INITIAL_PRIORITY, may be provided by the compiler to allow the user to specify an initial priority for the task. If neither pragma is included, then the priority is initially undefined but can be set using this package. If a task is specified with both the PRIORITY and INITIAL_PRIORITY pragmas, the INITIAL_PRIORITY pragma is ignored; implementations should give a warning if this occurs.

The semantics of this proposal are defined in terms of base and active priorities. The base priority is given to a task at task creation or changed by the use of this package. When pragma PRIORITY is used, the base priority is determined statically and can never be modified by the user. When the pragma INITIAL_PRIORITY is used, it establishes the initial base priority which can later be modified by the SET_DYNAMIC_PRIORITY procedure. When neither pragma is used, the base priority is unspecified but can be later modified by the SET_DYNAMIC_PRIORITY procedure. Tasks with static priorities and tasks with dynamic priorities are both permitted in the same Ada program.

The active priority is used by the runtime environment for dispatching and resource allocation. Generally speaking, the active priority is determined by the runtime environment and is the maximum of the base priority of the task and the active priorities of all tasks "waiting" on it. Specifically the rules for calculating the task's active priority are:

1. When a task is the acceptor of a rendezvous, the caller contributes to the acceptor's active priority.

2. With priority inheritance (see Priority Inheritance Discipline CIFO Entry), tasks waiting to rendezvous with another task contribute to the acceptor's active priority.

3. During activation and termination of dependent tasks, the master contributes to the active priority of the dependent tasks.

The base priority of the affected task is set before the operation SET_DYNAMIC_PRIORITY returns, as perceived by the calling task. If the affected task is the caller, or is not executing and is eligible to execute on the same processor, any effect on its active priority is also felt before the operation returns. If the affected task is executing on a different processor, or is not eligible to execute on the same processor, such an effect occurs no later than the next time the affected processor reaches a dispatching point. Any subsequent calls (without intervening calls to SET_DYNAMIC_PRIORITY) to see the dynamic priority of the affected task, via DYNAMIC_PRIORITY_OF, returns the new base priority.

Priority changes affect every place in which priorities are considered. In all such circumstances, priorities of subtype SYSTEM.PRIORITY or DYNAMIC_PRIORITIES.DYNAMIC_PRIORITY are treated equivalently by the runtime environment.

1. Changing a target task's priority affects its position in the run queue; it also affects its position on entry queues if priority entry queuing is specified (see Entry for QUEUING_DISCIPLINE). (The scheduling policy determines the new location of the target task in any priority queue, relative to any other tasks of equal priorities.)

2. If the target task is the caller in a rendezvous, the active priority of the acceptor task is set (though only temporarily, until the end of the rendezvous); the acceptor's new priority is the maximum of the caller's new priority and acceptor's old priority.

3. If the target task is the acceptor in a rendezvous, the priority of the caller is not affected.

4. If Priority Inheritance Discipline is in effect (see Priority Inheritance Discipline CIFO Entry), the active priority of a task is always the maximum of its own base priority and the active priorities of all the tasks that are currently "waiting for it". For example, if a task becomes the highest priority task in an entry queue (as a result of changing its priority), then the active priority of the acceptor is elevated as well, even though the two are not currently engaged in a rendezvous.

The SET_DYNAMIC_PRIORITY procedure sets the new base priority of the task specified by OF_TASK, which may be the task performing the call or any other task whose task ID is known. Note that a call to SET_DYNAMIC_PRIORITY which changes the priority of a task in rendezvous requires a recomputation of active priorities.

The DYNAMIC_PRIORITY_OF function returns the base priority of the specified task regardless of how it was last set. In particular, in the absence of both pragmas and prior to a call to SET_DYNAMIC_PRIORITY, the base priority must be assigned by the runtime environment. The exception TASK_ID_ERROR, defined in package TASK_IDS, is raised if a nonexistent task is specified.

Discussion

A task can set its own dynamic priority as follows:

```
with DYNAMIC_PRIORITIES;
with TASK_IDS;
...
task body SOME_TASK is
    SOME_DYNAMIC_PRIORITY : DYNAMIC_PRIORITIES.DYNAMIC_PRIORITY;
begin
    ...
    DYNAMIC_PRIORITIES.SET_DYNAMIC_PRIORITY( OF_TASK => TASK_IDS.SELF,
                                             TO => SOME_DYNAMIC_PRIORITY );
    ...
end SOME_TASK;
```

A simpler and more straightforward approach to providing a dynamic priority capability would be to use pragma PRIORITY to specify the initial priority and then the SET_DYNAMIC_PRIORITY procedure to modify it. This method would eliminate the need for the new pragma, INITIAL_PRIORITY, and the subtype and exception of package DYNAMIC_PRIORITIES. Unfortunately the Ada Rapporteur Group has not interpreted the LRM broadly enough to allow this solution.

The compiler must inform the runtime environment when pragma PRIORITY is specified so that an attempt to change priority dynamically will raise an exception. Compilers must provide a warning if a task is specified with both the PRIORITY and INITIAL_PRIORITY pragmas.

Interactions with Other CIFO Entries

Time Critical Sections: If a target task is in a Time Critical Section and its base priority is lowered, its active priority is not affected until it exits the Time Critical Section.

The task suspension Entries and Time Slicing: If a target task is in the suspended state, the priority change should take place immediately. Some multiprocessor implementation may not be capable of guaranteeing immediate effect; in those cases, the priority changes should take place but no later than the next synchronization point of that processor.

Synchronization Discipline: If priority queuing is the discipline in use, the priority change may force a reordering of queues.

Priority Inheritance Discipline: If priority inheritance is combined with global priority discipline in entry queues and select alternatives, the priority change may force a reordering of queues in acceptor tasks where the affected tasks await and, in turn, the reordering of queues where the acceptor tasks which may be waiting, and so on.

Synchronous and Asynchronous Task Scheduling: This Entry uses only the DYNAMIC_PRIORITY type in this package to fulfill the requirement of specifying a new priority for a task.

Fast Interrupt Pragmas: Fast interrupts and interrupt rendezvous are unaffected due to their close connection to hardware priorities of the hardware interrupts. Changing dynamic priorities does affect the task enclosing the Medium Fast Interrupt Entry.

Asynchronous Cooperation Mechanism Entries: The queues that are used with asynchronous cooperation objects (e.g. EVENTS, BUFFERS, or RESOURCES) will respond to priority changes in exactly the same way as queues associated with Ada task entries.

Changes from the Previous Release

The suggestion to avoid the mixing of Ada static priorities and dynamic priorities has been dropped. Since support for task identifiers is now required for CIFO Entries, the alternate package was deleted from the Entry.

The word (and type) priority is no longer overloaded. Priorities described in this package are now consistently described as "dynamic priorities" (or subtype DYNAMIC_PRIORITY).

The concepts of base and active priorities for tasks are used to explain the semantics of this Entry.

Time Critical Sections

Issue

Certain time-critical sections of code must be guaranteed to be executed to completion without preemption and with minimal interruption.

Developers of real-time software have expressed a need for the ability to ensure that a given segment of code is executed without interruption. In particular, there may be a timing constraint on the segment. Minimal standard Ada provides no way of ensuring that the processor is not preempted by the RTE from a task at any time. Moreover, it is difficult to even ensure against preemption by other Ada tasks of the same priority, since the RTE is permitted to use time-slicing to interleave execution of such tasks. In some applications it could be catastrophic to switch tasks during such a section.

Proposal

An implementation-defined package:

```
package TIME_CRITICAL_SECTIONS is
    procedure BEGIN_TIME_CRITICAL_SECTION;
    procedure END_TIME_CRITICAL_SECTION;
    function TIME_CRITICAL return BOOLEAN;
end TIME_CRITICAL_SECTIONS;
```

or an implementation-defined generic procedure:

```
generic
    with procedure TIME_CRITICAL_SECTION;
procedure CALL_TIME_CRITICAL_SECTION;
```

The effect of calling procedure BEGIN_TIME_CRITICAL_SECTION is to guarantee that the processor is not preempted from the calling task, until it next calls END_TIME_CRITICAL_SECTION. The intention is that the timing of execution between such calls should be predictable from examination of the intervening code.

The effect of calling an instance of the generic procedure is to guarantee that the procedure parameter is executed without preemption. The effect is the same as if BEGIN_TIME_CRITICAL_SECTION is called followed by a call to the procedure NON_PREEMPTIBLE_SECTION followed by a call to END_TIME_CRITICAL_SECTION (see the example). In particular, any exception raised within a call to an instance of CALL_TIME_CRITICAL_SECTION ends the time critical section and re-raises the exception.

Ideally, a Time Critical Section should not even be interruptable by hardware interrupts, but this may not always be practical. At a minimum, calls to BEGIN_TIME_CRITICAL_SECTION and to END_TIME_CRITICAL_SECTION should have the same effect as disabling dispatching and enabling dispatching respectively. An implementation may permit a task with preemption disabled to be interrupted by hardware interrupt handlers that are absolutely essential to the correct operation of the system (e.g., interruptions for timers or heart beats), but the aggregate fraction of processor time potentially lost to such interrupts should be bounded a priori, so that execution timing is still predictable.

Any interrupts permitted by the RTE implementation within Time Critical Sections should be documented, with sufficient information to permit prediction of running times, as well as any limitations on the safe use of these procedures. It is the responsibility of a programmer using this feature to ensure that all documented rules for safe use are followed.

Any Ada operation or CIFO service call that would block the further execution of the task should be avoided within a Time Critical Section. However, if such an operation is performed, the effect is implementation-defined. Any implementation claiming conformance with this Entry must explicitly define the effect of executing a blocking operation from within a Time Critical Section.

The function TIME_CRITICAL returns TRUE if and only if the calling task is currently within a Time Critical Section on its processor. Calling BEGIN_TIME_CRITICAL_SECTION when the processor is within a Time Critical Section, or END_TIME_CRITICAL_SECTION when the processor is no longer within a Time Critical Section, is permitted, and has no effect.

Discussion

These features might be used as follows:

```
declare
    use PREEMPTION_CONTROL;
begin
    BEGIN_TIME_CRITICAL_SECTION;
    – do time-critical work.
    END_TIME_CRITICAL_SECTION;
exception
    when others = >
        END_TIME_CRITICAL_SECTION;
        raise;
end;
```

Or:

```
procedure MY_TIME_CRITICAL_SECTION is
begin
    –do time-critical work.
end MY_TIME_CRITICAL_SECTION;

procedure CALL_MY_TIME_CRITICAL_SECTION is
    new CALL_TIME_CRITICAL_SECTION ( MY_TIME_CRITICAL_SECTION );

–when needed:
CALL_MY_TIME_CRITICAL_SECTION; --same effect as above.
```

While some have criticized this feature as being dangerous, or inconsistent with the standard Ada semantics for priorities, it is clear that realtime programmers need this capability and will get the equivalent effect some way, reverting to machine-code to disable interrupts if necessary. Because the alternatives seem more hazardous and less transportable, the ARTEWG feels providing this feature is preferable.

As for the Ada standard, the wording concerning priorities only requires that a lower priority task be preempted when a higher priority task can "sensibly" be executed using the same resources. By using this feature, the programmer is simply telling the RTE that the current task cannot be sensibly (safely) preempted. It is the responsibility of the programmer to ensure that a task does not remain within a Time Critical Section for very long, and that use of this feature does not nullify the intent of any specified priorities.

An alternative solution has been suggested that all such Time Critical operations might be concentrated in the entries of one task, of highest priority. The worst failing of this is that it does not deal with the problem of interruptions by the RTE. It also introduces new problems. Since an implementation need not support any particular range of priorities (or any at all), any solution that depends on priorities is already implementation-dependent. This approach is also dangerous because it depends on there being no other task in the system of higher priority. That is difficult to ensure, especially in a system that is evolving over time, or where there is multiprogramming. Using priorities in this way is also problematic in a multiprocessor system, since it would preclude more than one processor from being in a time-critical section at one time, forcing unnecessary waiting.

Yet another alternative to this feature is use of explicit interrupt control, as provided in a separate catalog Entry (see Interrupt Management CIFO Entry). This has the disadvantage of being more machine-dependent, and therefore less portable. It also is less reliable, since an RTE implementation may not provide for explicit masking of all hardware interrupts -- in order to protect against interference with interrupts used by the RTE.

This feature is presented as two options because of implementation considerations. The generic is considered "safer" to use since it is not possible to forget to exit a Time Critical Section at the end. However, to implement this option efficiently requires that pragma INLINE be available. If this is not the case, the generic requires one extra procedure call over the separate explicit calls. The generic can even be thought of in terms of a call to an Entry in a high priority RTE task (similar to interrupts), which then calls the procedure at its priority.

Note that these procedures are expected to be implemented in-line, if possible.

Although this feature can be used to guarantee mutually exclusive access to shared resources in a uniprocessor environment, it is recommended that other, more efficient catalogue features be used for that purpose whenever possible. *In a multiprocessor environment or for those applications which require portability to* multiprocessor environments, this feature can not be used to guarantee mutually exclusive access to resources shared by the processors.

If a delay is desired within a Time Critical Section, it should be programmed as an busy-loop, rather than with the standard Ada delay statement.

Interactions with Other CIFO Entries

Dynamic Priorities: If a target task is in a Time Critical Section and its base priority is lowered, its active priority is not affected until it exits the Time Critical Section.

Task Suspension: Calling SUSPEND_SELF within a Time Critical Section is implementation defined and should be avoided. If a Time Critical Section is nested within the region in which dispatching is disabled, exiting the Time Critical Section (via a call to END_TIME_CRITICAL_SECTION) should not implicitly enable dispatching. From within a Time Critical Section, there is no additional benefit from disabling dispatching. Consequently the user should avoid such nested regions. If dispatching is disabled then enabled from within a Time Critical Section, the call to ENABLE_DISPATCHING does not implicitly enable preemption.

Two Stage Task Suspension: Calling SUSPEND_SELF within a Time Critical Section is implementation defined and should be avoided.

Asynchronous Task Suspension: Enabling holding within a Time Critical Section is dangerous and should be avoided. Calling HOLD_TASK within a Time Critical Section to hold yourself is implementation defined and should be avoided.

Time Slicing: If a task's slice expires while the task is within a Time Critical Section, the task will continue to execute until it exits the Time Critical Section.

Abort Via Task Identifiers: The abort takes place at the next dispatching point which shoulu be the end of the Time Critical Section.

Resources, Events, Pulses, Buffers, Blackboards, Broadcasts, Barriers, Mutually Exclusive Access to Shared Data, Shared Locks, Trivial Entries, Dynamic Storage Management: Use of any of these entries within a Time Critical Section may cause blocking and should be avoided.

Asynchronous Transfer of Control: Avoid the use of Time Critical Section within the actual procedure supplied to the formal generic parameter AGENT_ACTIONS.

Signals: If Signals is implemented properly, it should not cause blocking and it is safe to use within a Time Critical Section.

Interrupt Management: In considering the interaction of this Entry with the Interrupt Management CIFO Entry, it became clear that this Entry did not define its effects on user controlled interrupt vectors. This was not considered feasible for CIFO Release 3.0, and will be added to the list of possible future CIFO enhancements. The user may choose to bracket Time Critical Sections with calls to the procedures defined in the Interrupt Management CIFO Entry to augment the user-controlled interrupts handled by Time Critical Sections.

Fast Interrupt Pragmas: Defining a Time Critical Section within a Fast Interrupt task or within an interrupt rendezvous associated with a Medium Fast Entry is not possible. Putting a Fast Interrupt task or a task with a Medium Fast Entry within a Time Critical Section is redundant.

Synchronous and Asynchronous Task Scheduling: The effect of calling WAIT_FOR_SCHEDULE in a Time Critical Section is implementation-defined and should be avoided.

Changes from Previous Release

This Entry was previously named Nonpreemptible Sections. Interactions with any Ada feature or CIFO Entry were more clearly defined.

Abortion via Task Identifier

Issue

It is sometimes necessary to abort a task that is not visible. This capability partially addresses the problem of writing reusable executives and failure-recovery tasks. If such a component is reusable, it cannot have visibility of those other tasks which it manages, since these are different for each application. Furthermore, even if visible, a given task may be one of many visible tasks, and thus may not be discernible at the point at which abortion is necessary, such as within an accept body or while traversing a list of task identifiers.

Proposal

This proposal consists of one library procedure:

```
with TASK_IDS;
procedure ABORT_TASK( I : in TASK_IDS.TASK_ID);
```

Calling this procedure would request the RTE to abort the task corresponding to identifier I. If a null TASK_ID is passed to ABORT_TASK then the exception TASK_ID_ERROR is raised.

Discussion

The semantics of this procedure are intended to be the same as that of LRM 9.10. It does, however, extend the capability to those task objects that are never accessible except via their TASK_ID (i.e., the main subprogram).

Interactions with Other CIFO Entries

This Entry depends on task identifiers.

Aborting tasks in Ada is a very dangerous action. Resources which have been allocated to an aborted task may be lost. In general if a CIFO Entry has well defined Ada semantics (namely,Mutually Exclusive Access to Shared Data, Events, Pulses, Blackboards, Broadcasts, and Barriers) then the effect of aborting a task, currently accessing the resources provided by that Entry, will have the normal Ada semantics. This may result in resources being lost. If an Entry doesn't have well defined Ada semantics then aborting a task, currently using that Entry, will

1) result in the resources controlled by the Entry being lost, and/or

2) require an additional description of interactions between the abort and that Entry.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: Aborting a suspended task with the abort statement or this Entry occurs no later than the next dispatching point of the processor where aborted task resides.

Time Critical Sections: The abort takes place at the next dispatching point which should be the end of the Time Critical Section.

Shared Locks, Resources, Buffers: Tasks which use the resources of these CIFO Entries and are aborted may lose these resources. Additionally a task which holds one of these resources and is aborted may cause deadlock.

Interrupt Management: Aborting a task which is disabling or enabling interrupts is potentially very dangerous.

Fast Interrupt Pragmas: Aborting a task for a Fast Interrupt or with a Medium Fast Interrupt Entry detaches that task from the interrupt.

Passive Task Pragma: This Entry cannot be applied directly to a passive task.

Dynamic Storage Management: Tasks which use the resources of this CIFO Entry and is aborted may lose these resources.

Changes from the Previous Release

There have been a few wording changes to the Entry, but the procedure specification remains the same.

Task Suspension

Issue

Realtime systems need efficient mechanisms to allow a task to temporarily block itself from executing. For example, the implementation of a messaging system needs to suspend a task awaiting the arrival of a message, and resume the task when the bus interrupt indicates that a message has arrived [Powers].

Proposal

This proposal consists of the following package:

```
with TASK_IDS;
package TASK_SUSPENSION is

    procedure ENABLE_DISPATCHING;

    procedure DISABLE_DISPATCHING;

    function DISPATCHING_ENABLED return BOOLEAN;

    procedure SUSPEND_SELF;

    procedure RESUME_TASK( T : in TASK_IDS.TASK_ID );

end TASK_SUSPENSION;
```

This package provides a means for a task to control its own execution.

The DISABLE_DISPATCHING and ENABLE_DISPATCHING operations provide a mechanism whereby a task can define specific regions of code in which the processor can not be involuntarily reassigned to another task.

The DISABLE_DISPATCHING service guarantees that the processor will not be involuntarily relinquished from the calling task.

The ENABLE_DISPATCHING service allows the processor to be reassigned to another task. All tasks are activated with dispatching enabled.

The SUSPEND_SELF service suspends the current task.

The RESUME_TASK service resumes the specified task if the task has called SUSPEND_SELF. If the specified task id is non-existent or invalid, the exception TASK_IDS.TASK_ID_ERROR is raised in the calling task. If the specified task has not performed SUSPEND_SELF, the RESUME_TASK call has no effect. Furthermore resuming a task that is not yet activated or is abnormal, completed, or terminated has no effect.

A task that becomes abnormal (via an abort statement or a call to ABORT_TASK (see the Abortion Via Task Identifiers CIFO Entry)) while suspended at a call to SUSPEND_TASK becomes completed. Subsequent termination of such an abnormal task does not await a call to RESUME_TASK, and follows the normal rules for task termination.

Discussion

For the purposes of discussion, we introduce the following terms:

Scheduling - the process of adding or removing tasks from the set of tasks that are eligible to execute using available processors.

Dispatching - the act of initiating the execution of a specific task on a specific processor. Dispatching a task on a processor necessarily implies that the task previously executing on that processor stops executing on the processor.

Blocking operation - any Ada construct or CIFO service that blocks the execution of the current task. Examples of blocking operations include: a delay statement; an accept statement for an entry which has no pending calls, or an entry call to a task that is not waiting at a corresponding accept; a WAIT on an event that is not in the UP state; a RECEIVE on an empty buffer, etc. A blocking operation is said to be COMPLETE when the condition causing the blocking operation no longer exists (e.g. the delay has expired, the event is set,etc.)

The DISABLE_DISPATCHING service prevents the dispatching of a task except as a consequence of a blocking operation. Calls to DISABLE_DISPATCHING do not, however, prevent task scheduling nor do they prevent the execution of interrupt handler code. Specifically DISABLE_DISPATCHING does not disable interrupts.

The dispatching state (enabled/disabled) is a binary attribute of a task and is preserved as part of the task's context. If a task is executing with dispatching disabled when it executes a blocking operation, the task will be suspended and another task will be dispatched for execution. Tasks resume execution with the same dispatching state that was present when they last suspended. A newly created task initially executes with dispatching enabled.

A call to ENABLE_DISPATCHING cancels the effect of a preceding call to DISABLE_DISPATCHING. Additionally, a call to ENABLE_DISPATCHING will cause the highest priority task that is eligible for execution on the current processor to execute (i.e. if there is a task eligible to execute on the current processor that is of higher priority than the current task, the higher priority task will be dispatched). The user is encouraged to keep the pairing of DISABLE_DISPATCHING and ENABLE_DISPATCHING (which defines a nondispatchable section) calls within a single program unit.

The SUSPEND_SELF procedure removes the current task from the set of tasks eligible to execute and forces the dispatching of a new task.

The RESUME_TASK procedure returns a task to the set of tasks eligible to execute. If dispatching is enabled, and the target task is of higher priority than the current task, the target task will be immediately dispatched.

DISABLE_DISPATCHING and ENABLE_DISPATCHING should not be used to attempt to guarantee mutually exclusive access to any resources other than the one processor on which the section of code is executing, and any other resources which are implied by possession of the processor (such as local I/O ports). Thus, it cannot be used to protect data against concurrent updates in a multiprocessor configuration, nor can it be used to protect data against concurrent updates by interrupt handlers.

The following example demonstrates the use of the facilities of this Entry to control access to a buffer shared by two tasks. Please note that this example is for illustrative purposes only, and is not intended as an optimal solution to the problem of managing a shared buffer (see the BUFFERS and BLACKBOARDS Entries).

In this example, the tasks PRODUCER and CONSUMER repeatedly operate on a shared buffer BUFF. PRODUCER generates and deposits data into BUFF CONSUMER retrieves and performs continued processing on data from BUFF. The actual operations on BUFF are synchronized via calls to the facilities defined in the Resources CIFO Entry. If CONSUMER is ready to process additional data from BUFF, and BUFF is empty, CONSUMER must suspend pending data availability. Similarly, when PRODUCER deposits data into an empty BUFF, it must resume the (potentially) suspended CONSUMER.

```
...
use TASK_SUSPENSION;
...
package CONTROLLER is new RESOURCES;

type BUFFER is
    record
        RES : CONTROLLER.RESOURCE := CONTROLLER.CREATE;
        DATA : ITEM;
    end record;

procedure ADD_TO_BUFFER( BUFF : in BUFFER; DATA : in ITEM );
procedure REMOVE_FROM_BUFFER( BUFF : in BUFFER; DATA : out ITEM );
function IS_EMPTY( BUFF : BUFFER ) return BOOLEAN;

BUFF : BUFFER;
pragma SHARED_DATA(BUFF);
CONSUMER_ID : TASK_IDS.TASK_ID;
...
task body PRODUCER is
    PRODUCED_DATA : ITEM;
begin
    loop
        – produce new data item
        CONTROLLER.GET( BUFF.RES );
        ADD_TO_BUFFER( BUFF, PRODUCED_DATA );
        CONTROLLER.RELEASE( BUFF.RES );
        RESUME_TASK( CONSUMER_ID );          –(A)
    end loop;
end PRODUCER;

task body CONSUMER is
    DATA_TO_CONSUME : ITEM;
begin
    CONSUMER_ID := TASK_IDS.SELF;
    CONTROLLER.GET( BUFF.RES );
    if IS_EMPTY( BUFF ) then
        DISABLE_DISPATCHING;                 –(1)
        CONTROLLER.RELEASE( BUFF.RES );      –(B)
        SUSPEND_SELF;                        –(C)
        ENABLE_DISPATCHING;                  –(2)
        CONTROLLER.GET( BUFF.RES );
    end if;
    REMOVE_FROM_BUFFER( BUFF, DATA_TO_CONSUME );
    CONTROLLER.RELEASE( BUFF.RES );
end CONSUMER;
```

In the above example, after release of the buffer resource BUFF.RES at point (B), a race condition exists between PRODUCER resuming CONSUMER at (A) and CONSUMER suspending itself at (C). On a uniprocessor this race condition is eliminated by the introduction of the DISABLE_DISPATCHING and ENABLE_DISPATCHING calls at (1) and (2). Once dispatching is disabled by the call at (1), CONSUMER may not be dispatched by either the call to RESOURCES.RELEASE at (B) or via CONSUMER becoming eligible to execute via any other means.

Calls to DISABLE_PREEMPTION and ENABLE_PREEMPTION are not equivalent to raising the current task to the highest possible priority, followed by lowering the priority. In particular, if the task's priority is modified while dispatching is disabled (either by the task itself, or by another task executing on another processor) the effect of disabling preemption will not be canceled but when preemption is enabled the task will compete for the processor at its new priority. A call to DYNAMIC_PRIORITY_OF indicating a task executing with dispatching disabled will yield the base priority of the target task.

Another example of the use of DISABLE_PREEMPTION and ENABLE_PREEMPTION is provided in the Entry on Asynchronous Task Suspension.

Interactions With Other CIFO Entries

Two Stage Task Suspension: If this Entry is implemented in conjunction with the Two Stage Task Suspension CIFO Entry, the following interactions exist.

a. ENABLE_DISPATCHING, DISABLE_DISPATCHING and DISPATCHING_ENABLED behave exactly as documented here.

b. TASK_SUSPENSION.RESUME_TASK will behave identically to TWO_STAGE_TASK_SUSPENSION. RESUME_TASK. Namely if the indicated task has executed a call to either TASK_SUSPENSION.SUSPEND_SELF or TWO_STAGE_TASK_SUSPENSION. SUSPEND_SELF, the indicated task will be resumed. If the indicated task has called TWO_STAGE_TASK_SUSPENSION.WILL_SUSPEND, its state will be updated such that a call to TWO_STAGE_TASK_SUSPENSION. SUSPEND_SELF will "fall-through."

c. TASK_SUSPENSION.SUSPEND_SELF will unconditionally suspend the calling task, without regard to whether TWO_STAGE_TASK_SUSPENSION.WILL_SUSPEND has been called, or whether a call to RESUME_TASK has been issued since the last call to TWO_STAGE_TASK_SUSPENSION.WILL_SUSPEND. No other state value necessary for the implementation of TWO_STAGE_TASK_SUSPENSION will be updated (e.g. the WILL_SUSPEND state will not be updated, nor will any "latched" RESUME_TASK be cleared).

Asynchronous Task Suspension: If a task named in a RESUME_TASK call is currently in a hold state due to a prior call to ASYNCHRONOUS_TASK_SUSPENSION.HOLD, the resume will have the effect of making the task eligible to execute only when it is named in a subsequent call to ASYNCHRONOUS_TASK_SUSPENSION.RELEASE_TASK.

Task Identifiers: As with all services that relies on task identifiers, TASK_IDS.TASK_ID_ERROR may be raised for any operation that takes a task identifier as a parameter. This exception will only be raised if the implementation performs validity checks (as indicated by the TASK_IDS_CHECKED boolean), and a task identifier designating a non-existent task is used as parameter.

Ada and CIFO blocking operations: The suspension state defined here is distinct from that of blocking operations. If a task which has executed a blocking operation that has not completed and that task is named in a call to RESUME_TASK, that call to RESUME_TASK will not complete the blocking operation, and consequently will not be scheduled or dispatched. It is impossible to execute a SUSPEND_SELF call while suspended by a blocking operation.Dynamic Priorities: If a target task is in the suspended state, the priority change should take place immediately. Some multiprocessor implementations may not be capable of guaranteeing immediate effect; in those cases, the priority changes should take place but no later than the next synchronization point of that processor.

Synchronous and Asynchronous Task Scheduling: Usage of these two Entries on the same tasks may result in indeterminate or erroneous results. Effects similar to task suspension can be accomplished with the judicious use of DESCHEDULE and SCHEDULE procedure calls.

Time Slicing: If the time quanta for a task expires while dispatching for that task is disabled, the task does not relinquish control of the CPU until dispatching is enabled. The occurrence of a new time slice does not resume a suspended task. A task loses the rest of its time slice if it is suspended.

Time Critical Sections: If a Time Critical Section is nested within a region in which dispatching is disabled, exiting the Time Critical Section (via a call to ENABLE_PREEMPTION) should not implicitly enable dispatching. Calling SUSPEND_SELF within a Time Critical Section is implementation defined and should be avoided.

Abort via Task Identifiers: Aborting a suspended task with the abort statement or this Entry occurs no later than the next dispatching point of the processor where the aborted task resides.

Asynchronous Transfer of Control: If a task is executing in a section with dispatching disabled when it is subject to an asynchronous transfer of control, the asynchronous transfer is deferred until scheduling is enabled.

Shared Locks: Suspending a task while it possesses a shared lock can cause deadlock.

Fast Interrupt Pragmas: No calls to any procedures in this Entry are permitted within a fast interrupt handler.

Passive Task Pragma: Since it is impossible to obtain the TASK_IDENTIFIER of a passive task, a passive task can not be named in a RESUME_TASK call. The exception PASSIVE_TASK_ID is raised if a passive task calls SUSPEND_SELF because there would be no valid task id that can be used for the RESUME_TASK procedure.

Dynamic Storage Management: Users should avoid calls to dynamic storage management facilities when dispatching is disabled.

Changes From The Previous Release

This Entry, along with the Two Stage Task Suspension and the Asynchronous Task Suspension, are derived from the Entry Controlling When a Task Executes. Three Entries were created to more specifically address the needs for task suspension: simple blocking for services like I/O, intertask communication and synchronization, and asynchronous suspension and resumption, respectively.

References

[Powers] Powers R. D., Roark C., "Ada Support for Real-Time Systems", *Proceedings of the Third International Workshop on Real-Time Ada Issues*, ACM SIGAda Ada Letters, Vol X, No. 4, pp 114-118.

Two Stage Task Suspension

Issue

Real-time systems need efficient mechanisms for building intertask communication and synchronization constructs that allow tasks to be temporarily blocked from executing. Such tasks must be able to specify their intention to suspend before unlocking a shared resource and then suspending. This is necessary in order to avoid race conditions if a task waiting on the shared resource will resume the task that is suspending itself.

Proposal

This proposal consists of the following package:

```
with TASK_IDS;
package TWO_STAGE_TASK_SUSPENSION is
    SUSPENSION_ERROR : exception;
    procedure WILL_SUSPEND;
    procedure SUSPEND_SELF;
    procedure RESUME_TASK( T : in TASK_IDS.TASK_ID );
end TWO_STAGE_TASK_SUSPENSION;
```

This package provides a means for a task to safely suspend its own execution The suspended task must be subsequently resumed by some other task.

These operations are probably most useful when used to build blocking synchronization constructs. They can be combined with non-blocking lock operations (e.g. test-and-set on uniprocessors, spin locks on multiprocessors) to build a wide range of blocking task synchronization and communication primitives.

The services in this package are designed to be used in conjunction with low level locking primitives (see example below). If simple task suspension is all that is required, the TASK_SUSPENSION package can be used instead.

The WILL_SUSPEND and SUSPEND_SELF procedures are used to implement a "two-stage" suspend operation. This allows the calling task to unlock any locks or semaphores it may be holding and suspend itself safely, without entering into a race condition with another task that may be contending for the lock and calling RESUME_TASK. A detailed example of the problem and the solution supported by this package is described below.

The WILL_SUSPEND procedure notifies the implementation that the task is about to suspend itself.

The RESUME_TASK procedure resumes the specified task if the task has called both WILL_SUSPEND and SUSPEND_SELF. If the specified task id is non-existent or invalid, the exception TASK_IDS.TASK_ID_ERROR is raised in the calling task. If the specified task has called WILL_SUSPEND but not SUSPEND_SELF, the RESUME_TASK is "latched" by the implementation and will have the effect of immediately resuming the task when it next calls SUSPEND_SELF. The call to RESUME_TASK before that task has called a WILL_SUSPEND has no effect. Furthermore resuming a task that is not yet activated or that is abnormal, completed, or terminated has no effect.

A task that becomes abnormal (via an abort statement or a call to ABORT_TASK (see the Abortion Via Task Identifiers CIFO Entry)) while suspended at a call to SUSPEND_TASK becomes completed. Subsequent termination of such an abnormal task does not await a call to RESUME_TASK, and follows the normal rules for task termination.

The SUSPEND_SELF service suspends the current task if that task has called WILL_SUSPEND, unless some other task has called RESUME_TASK on it since the last time the current task called WILL_SUSPEND. In that case the task is not suspended (the resume was "latched") and the SUSPEND_SELF has no effect.

If a task calls SUSPEND_SELF without a previous call to WILL_SUSPEND, the exception SUSPENSION_ERROR will be raised in the calling task and it will not be suspended.

Discussion

Many blocking task communication constructs can be built using simple locks and the operations of this package.

The following example demonstrates the use of the facilities of this Entry to control access to a buffer shared by two tasks. Please note that this example is for illustrative purposes only, and is not intended as an optimal solution to the problem of managing a shared buffer (see the BUFFERS and BLACKBOARDS Entries).

In this example, the tasks PRODUCER and CONSUMER repeatedly operate on a shared buffer BUFF. PRODUCER generates and deposits data into BUFF. CONSUMER retrieves and performs continued processing on data from BUFF. The actual operations on BUFF are synchronized via calls to the facilities defined in the Resources CIFO Entry.

If CONSUMER is ready to process additional data from BUFF, and BUFF is empty, CONSUMER must suspend pending data availability. Similarly, when PRODUCER deposits data into an empty BUFF, it must resume the (potentially) suspended CONSUMER.

```
...
use TWO_STAGE_TASK_SUSPENSION;
...
package CONTROLLER is new RESOURCES;

type BUFFER is
    record
        RES : CONTROLLER.RESOURCE := CONTROLLER.CREATE;
        DATA : ITEM;
    end record;

procedure ADD_TO_BUFFER( BUFF : in BUFFER; DATA : in ITEM );
procedure REMOVE_FROM_BUFFER( BUFF : in BUFFER; DATA : out ITEM );
function IS_EMPTY( BUFF : BUFFER ) return BOOLEAN;

BUFF : BUFFER;
pragma SHARED_DATA(BUFF);

CONSUMER_ID : TASK_IDS.TASK_ID;
...
```

```
task body PRODUCER is
    PRODUCED_DATA : ITEM;
begin
    loop
        - produce new data item
        CONTROLLER.GET( BUFF.RES );
        ADD_TO_BUFFER( BUFF, PRODUCED_DATA );
        CONTROLLER.RELEASE( BUFF.RES );
        RESUME_TASK( CONSUMER_ID );-(A)
    end loop;
end PRODUCER;

task body CONSUMER is
    DATA_TO_CONSUME : ITEM;
begin
    CONSUMER_ID := TASK_IDS.SELF;
    CONTROLLER.GET( BUFF.RES );
    if IS_EMPTY( BUFF ) then
        WILL_SUSPEND;   -(1)
        CONTROLLER.RELEASE( BUFF.RES );-(B)
        SUSPEND_SELF;   -(C)
        CONTROLLER.GET( BUFF.RES );
    end if;
    REMOVE_FROM_BUFFER( BUFF, DATA_TO_CONSUME );
    CONTROLLER.RELEASE( BUFF.RES );
end CONSUMER;
```

In the above example, after release of the buffer resource BUFF.RES at point (B), a race condition exists between PRODUCER resuming CONSUMER at (A) and CONSUMER suspending itself at (C). By using the two stage task suspension facility, either outcome of the race is correctly handled. If, after the WILL_SUSPEND call at (1), the RESUME_TASK at (A) occurs prior to the SUSPEND_SELF at (C), the resume will be latched, and the suspension will not occur. If the SUSPEND_SELF call occurs first, the RESUME_TASK will correctly cancel it. This facility is intended to work on either a uniprocessor or multiprocessor implementation.

Interactions with Other CIFO Entries

Task Suspension: If this Entry is implemented in conjunction with the TASK_SUSPENSION Entry, the following interactions exist.

a. TWO_STAGE_TASK_SUSPENSION.RESUME_TASK will behave identically to TASK_SUSPENSION.RESUME_TASK, namely, if the indicated task has executed a call to either TASK_SUSPENSION.SUSPEND_SELF or TWO_STAGE_TASK_SUSPENSION. SUSPEND_SELF the indicated task will be resumed. If the indicated task has called TWO_STAGE_TASK_SUSPENSION.WILL_SUSPEND, its state will be updated such that a call to TWO_STAGE_TASK_SUSPENSION. SUSPEND_TASK will "fall-through."

b. Calls to TASK_SUSPENSION.DISABLE_DISPATCHING and TASK_SUSPENSION. ENABLE_DISPATCHING have the same effect on the SUSPEND_SELF and RESUME_TASK procedures of TWO_STAGE_TASK_SUSPENSION as they do on their counterparts in TASK_SUSPENSION.

Asynchronous Task Suspension: If a task named in a RESUME_TASK call is currently in a hold state due to a prior call to ASYNCHRONOUS_TASK_SUSPENSION.HOLD, the resume will have the effect of making the task eligible to execute only when it is named in a subsequent call to ASYNCHRONOUS_TASK_SUSPENSION.RELEASE_TASK.

Ada and CIFO blocking operations: The suspension state defined here is distinct from that of blocking operations. If a task has executed a blocking operation that has not completed and that task is named in a call to RESUME_TASK, that call to RESUME_TASK will not complete the blocking operation, and consequently will not be scheduled or dispatched. It is impossible to execute a SUSPEND_SELF call while suspended by a blocking operation.

Task Identifiers: As with all services that rely on task identifiers, TASK_IDS.TASK_ID_ERROR may be raised for any operation that takes a task identifier as a parameter. This exception will only be raised if the implementation performs validity checks (as indicated by the TASK_IDS_CHECKED boolean), and a task identifier designating a non existent task is used as a parameter.

Dynamic Priorities: If a target task is in the suspended state, the priority change should take place immediately. Some multiprocessor implementations may not be capable of guaranteeing an immediate effect; in those cases, the priority changes should take place but no later than the next synchronization point of that processor.

Synchronous and Asynchronous Task Scheduling: Usage of these two Entries on the same tasks may result in indeterminate or erroneous results. Effects similar to task suspension can be accomplished with the judicious use of DESCHEDULE and SCHEDULE procedure calls.

Time Slicing: If the time quantum for a task expires while dispatching for that task is disabled, the task does not relinquish control of the CPU until dispatching is enabled. The occurrence of a new time slice does not resume a suspended task. A task loses the rest of its time slice if it is suspended.

Time Critical Sections: Calling SUSPEND_SELF within a Time Critical Section is implementation defined and should be avoided.

Abort via Task Identifiers: Aborting a suspended task with the abort statement or this Entry occurs no later than the next dispatching point of the processor where aborted task resides.

Asynchronous Transfer of Control: Its effect takes place when the target task is resumed.

Shared Locks: Suspending a task while it possesses a shared lock can cause deadlock.

Fast Interrupt Pragmas: No calls to any procedures in this Entry are permitted within a fast interrupt handler.

Passive Task Pragma: Since it is impossible to obtain the TASK_IDENTIFIER of a passive task, a passive task can not be named in a RESUME_TASK call. The exception PASSIVE_TASK_ID is raised if a passive task calls SUSPEND_SELF because there would be no valid task id that can be used for the RESUME_TASK procedure.

Changes from the Previous Release

This Entry, along with Task Suspension and Asynchronous Task Suspension, is derived from the Entry Controlling When a Task Executes. Three Entries were created to more specifically address the needs for task suspension: simple blocking for services like I/O, intertask communication and synchronization, and asynchronous suspension and resumption, respectively.

Asynchronous Task Suspension

Issue

Some Real-time systems require that one task have the ability to prevent the further execution of another. This capability is inherently dangerous, but in some instances unavoidable. It is hoped that most applications will not require the use of this package, and that other packages defined in this catalogue can be used in its place, e.g. the TASK_SUSPENSION package that provides the SUSPEND_SELF operation.

However, there are applications that require this capability. For example, some applications may implement a mode change by holding and releasing groups of tasks. Fault-tolerant applications might use HOLD_TASK to aid in recovering CPU resources from tasks caught in infinite loops. We therefore include this package in the catalogue, in spite of the dangers it poses.

The ARTEWG has strived to provide a definition of this operation that is both reasonably safe and reasonably efficient to implement.

Proposal

This proposal consists of the following package:

```
with TASK_IDS;
package ASYNCHRONOUS_TASK_HOLDING is
    HOLDING_ERROR : exception;
    procedure ENABLE_HOLDING;
    procedure DISABLE_HOLDING;
    function HOLDING_ENABLED return BOOLEAN;
    procedure HOLD_TASK (T : in TASK_IDS.TASK_ID);
    procedure HOLD_TASK (T : in TASK_IDS.TASK_ID; HOLDING : out BOOLEAN);
    procedure RELEASE_TASK (T: in TASK_IDS.TASK_ID);
end ASYNCHRONOUS_TASK_HOLDING;
```

This package provides a means for a task to temporarily prevent the further execution of any other task (i.e., to "hold" the task). The task may be subsequently released to continue its execution.

A task may use this package to prevent its own execution, however if that is the only application need for preventing the task's execution, the TASK_SUSPENSION package provides a more efficient method.

Tasks are created and activated with holding enabled.

Once disabled (e.g., by a call to DISABLE_HOLDING), the ENABLE_HOLDING procedure must be called by a task before it can again be held by HOLD_TASK.

The DISABLE_HOLDING procedure cancels the effects of the ENABLE_HOLDING procedure. The calling task is no longer eligible for holding via the HOLD_TASK procedure.

The implementation must allow calls to ENABLE_HOLDING and DISABLE_HOLDING to be nested. For example, if two calls to DISABLE_HOLDING are made by a task, two calls to ENABLE_HOLDING are required before the task is again eligible for holding.

The HOLDING_ENABLED function returns TRUE if the calling task has made itself holdable via a call to ENABLE_HOLDING.

The HOLD_TASK procedures asynchronously hold the execution of the specified task if that task is currently eligible for holding. If the task id specified is non-existent or otherwise invalid the exception TASK_IDS.TASK_ID_ERROR is raised in the calling task. HOLD_TASK is provided in two overloaded forms. The first procedure contains a single parameter of type TASK_IDS.TASK_ID. If the specified task has not made itself eligible for holding via a call to ENABLE_HOLDING the exception HOLDING_ERROR is raised in the calling task -- even if the specified task is abnormal, complete, terminated, or out-of-scope.

The second form provides a parameter, HOLDING, of type BOOLEAN. If the specified task has not made itself eligible for holding via a call to ENABLE_HOLDING the value of the HOLDING parameter will be FALSE. Conversely, if the specified task has made itself eligible to be held via a call to ENABLE_HOLDING the value of the HOLDING parameter will be TRUE.

A call to HOLD_TASK designating a task that is already held has no effect. A call to HOLD_TASK designating a task that had holding enabled when it became abnormal, complete, or terminated, has no effect.

The RELEASE_TASK procedure cancels the effect of a previous HOLD_TASK call. If the task id specified is not held via a previous call to HOLD_TASK this operation has no effect. If a task is otherwise eligible for execution when it is released, it will be considered for execution at the time of the RELEASE_TASK call. If the task is not otherwise eligible for execution (e.g. it has a pending delay, or is waiting for a rendezvous) a RELEASE_TASK call has no effect other than to remove the hold.

A task that is named in an abort statement or a call to ABORT_TASK while in a held state becomes abnormal (see Abortion Via Task Identifier CIFO Entry). Completion of such an abnormal task depends on the execution of the task when it was named in the HOLD_TASK call, and follows the normal language rules. Specifically, a held task whose execution was suspended at an accept statement, a select statement, or a delay statement, becomes completed. Furthermore, a held task whose execution was suspended at an entry call, and it is not yet in a corresponding rendezvous, becomes completed and is removed from the corresponding entry queue. A task that had not yet started its activation can not be in a held state, as it can not have executed a call to ENABLE_HOLDING.

The completion of any other abnormal task which is held can occur no later than when the task reaches its next "abortion" synchronization point (RM 9.10:6). If an implementation can immediately affect the task's completion, without having to dispatch the task to reach a abortion synchronization point, then the task is completed prior to the completion of the abort statement. If the implementation must wait for the task to reach its next abortion synchronization point before completing the task, the task can not be dispatched to reach its next abortion synchronization point prior to being named in a RELEASE_TASK call.

Discussion

The asynchronous HOLD_TASK procedure provided by this package is an inherently unsafe operation. A task may be held while it has exclusive access to a critical system resource, thus causing blocking in other parts of the system. Or due to the asynchronous nature of the suspension it may be held "too late". In the general case, there is no guarantee that the task will ever become held.

If a compilation system supports this Entry, it shall set the state of all tasks to "holdable" upon creation. The choice of creating and activating tasks with holding enabled was made to ensure that when an application needs this capability (for drastic actions, like reconfiguration) that all the tasks in the system cooperate. Unless enabling of holding is a default, this cooperation may be overlooked. This fact becomes significant when an application uses this Entry; otherwise the holdability of a task is a moot point.

The following example illustrates a potential use of the package ASYNCHRONOUS_TASK_HOLDING. In this example, arbitrary tasks may detect a transient overload situation, and signal the overloa [illegible] a centralized overload manager. The overload manager is responsible for load shedding via temporarily shutting down a low priority subsystem within the application. The subsystem consists of a collection of tasks. When the end of the overload situation is detected, the process is reversed.

There are a set of low priority tasks which will be held in the overloaded situation. In general they should use the ENABLE_HOLDING and DISABLE_HOLDING in the following format:

```
with ASYNCHRONOUS_TASK_HOLDING, ABORT_TASK;
...
task LOWLY_TASK;
...
task body LOWLY_TASK is
...
begin
    -- implicit call to ATH.ENABLE_HOLDING;
    ...
    -- implicit call to ATH.DISABLE_HOLDING;
end ASYNCHRONOUS_TASK_HOLDING;
```

Then there is OVERLOAD_MANAGER, responsible for controlling when low priority tasks are held.

```
with ASYNCHRONOUS_TASK_HOLDING;
...
with ABORT_TASK;
...

    type TASK_SET is array ( NATURAL range <> ) of TASK_IDS.TASK_ID;

    LOW_PRIORITY_SUBSYSTEM      : TASK_SET( 1 .. N );

    MEDIUM_PRIORITY_SUBSYSTEM   : TASK_SET( 1 .. N );

    HIGH_PRIORITY_SUBSYSTEM     : TASK_SET( 1 .. N );

    URGENT_SUBSYSTEM            : TASK_SET( 1 .. N );

    OVERLOAD    : EVENTS.EVENT := EVENTS.CREATE;

    UNDERLOAD   : EVENTS.EVENT := EVENTS.CREATE;

    task SYSTEM_STATUS_MANAGER is
        entry LOG_OVERLOAD( TIME_STAMP : in CALENDAR.TIME );
    end SYSTEM_STATUS_MANAGER;

    task OVERLOAD_MANAGER is
        pragma PRIORITY( SYSTEM.PRIORITY'last );
    end OVERLOAD_MANAGER;
```

```
task body OVERLOAD_MANAGER is
    WAS_HELD : BOOLEAN;
    package ATH renames ASYNCHRONOUS_TASK_HOLDING;
begin
    loop
        EVENTS.WAIT( OVERLOAD );
        TASK_SUSPENSION.DISABLE_DISPATCHING;
        EVENTS.RESET( OVERLOAD );
        for INDEX in LOW_PRIORITY_SUBSYSTEM'range loop
            DO_HOLD : begin
                ATH.HOLD_TASK( LOW_PRIORITY_SUBSYSTEM(INDEX), WAS_HELD );
                if not WAS_HELD then
                    ABORT_TASK( LOW_PRIORITY_SUBSYSTEM(INDEX) );
                end if;
            exception
                when TASK_IDS.TASK_ID_ERROR =>
                    null;
                when others =>
                    TASK_SUSPENSION.ENABLE_DISPATCHING;
                    raise;
            end DO_HOLD;
        end loop;
        TASK_SUSPENSION.ENABLE_DISPATCHING;
        SYSTEM_STATUS_MANAGER.LOG_OVERLOAD( CALENDAR.CLOCK );
        EVENTS.WAIT( UNDERLOAD );
        TASK_SUSPENSION.DISABLE_DISPATCHING;
        EVENTS.RESET( UNDERLOAD );
        for INDEX in LOW_PRIORITY_SUBSYSTEM'range loop
            DO_RELEASE : begin
                ATH.RELEASE_TASK( LOW_PRIORITY_SUBSYSTEM(INDEX) );
            exception
                when TASK_IDS.TASK_ID_ERROR =>
                    null;
                when others =>
                    TASK_SUSPENSION.ENABLE_DISPATCHING;
                    raise;
            end DO_RELEASE;
        end loop;
        TASK_SUSPENSION.ENABLE_DISPATCHING;
        SYSTEM_STATUS_MANAGER.LOG_OVERLOAD( CALENDAR.CLOCK );
    end loop;
end OVERLOAD_MANAGER;
```

Interaction with Other CIFO Entries

Task Suspension, Two Stage Task Suspension: The HOLD_TASK and RELEASE_TASK procedures defined in this Entry are independent of suspend and resume procedures in the Task Suspension and Two Stage Task Suspension CIFO Entries. In particular, a RESUME_TASK operation from TASK_SUSPENSION or TWO_STAGE_TASK_SUSPENSION does NOT cancel the effect of a call to ASYNCHRONOUS_TASK_SUSPENSION.HOLD_TASK. Similarly, a call to

ASYNCHRONOUS_TASK_SUSPENSION.RELEASE_TASK does not cancel the effect of a SUSPEND_SELF from TASK_SUSPENSION or TWO_STAGE_TASK_SUSPENSION.

It is possible for a task to have suspended itself via a call to SUSPEND_SELF from TASK_SUSPENSION or TWO_STAGE_TASK_SUSPENSION and subsequently be named in a call to ASYNCHRONOUS_TASK_SUSPENSION. HOLD_TASK. Under these circumstances, calls to both RESUME_TASK from TASK_SUSPENSION or TWO_STAGE_TASK_SUSPENSION and ASYNCHRONOUS_TASK_SUSPENSION.RELEASE_TASK are necessary before the task can be dispatched again.

If a task executes a blocking Ada construct (e.g. delay statement, an accept statement for an entry which has no pending calls, or an entry call to a task that is not waiting at a corresponding accept) and that task is subsequently named in a call to ASYNCHRONOUS_TASK_SUSPENSION.HOLD_TASK, the subsequent completion of the blocking Ada construct can not dispatch the task until the task is named in a call to ASYNCHRONOUS_TASK_SUSPENSION.RELEASE_TASK. The same is true for a task that executes a CIFO operation which blocks the calling task (e.g. a WAIT on an event that is not in the UP state, a RECEIVE on an empty buffer, etc.).

Task Identifiers: As with all services that rely on task identifiers, TASK_IDS.TASK_ID_ERROR may be raised for any operation that takes a task identifier as a parameter. This exception will only be raised if the implementation performs validity checks (as indicated by the TASK_IDS_CHECKED boolean), and a task identifier designating a non-existent task is used as a parameter.

Dynamic Priorities: If a target task is in the held state, the priority change should take place immediately. Some multiprocessor implementations may not be capable of guaranteeing immediate effect; in those cases, the priority changes should take place but no later than the next synchronization point of that processor.

Synchronous and Asynchronous Task Scheduling: Usage of these two Entries on the same tasks may result in indeterminate or erroneous results. Effects similar to asynchronous task suspension can be accomplished with the judicious use of DESCHEDULE and SCHEDULE procedure calls.

Time Slicing: The occurrence of a new time slice will not release a held task. A task loses the rest of its time slice if it is suspended.

Time Critical Sections: Enabling holding within a Time Critical Section is dangerous and should be avoided. The effect of calling HOLD_TASK within a Time Critical Section is implementation defined and should be avoided.

Abort via Task Identifiers: Aborting a held task with the abort statement or this Entry occurs no later than the next dispatching point of the processor where the aborted task resides.

Asynchronous Transfer of Control: Its effect takes place when the target task is released. For a held task containing an instantiation of ASYNCH_AGENT, use of this Entry has no effect until the task is released.

Shared Locks: Holding a task while it possesses a shared lock can cause deadlock.

Interrupt Management: Avoid disabling interrupts while holding is enabled.

Fast Interrupt Pragmas: No calls to any procedures in this Entry are permitted within a fast interrupt handler. Enabling holding has no effect on a Fast Interrupt or an interrupt rendezvous associated with Medium Fast Interrupt Entry.

Passive Task Pragma: Since it is impossible to obtain the task id of a passive task, a passive task can not be named in a HOLD_TASK or RELEASE_TASK call.

Changes from the Previous Release

This Entry, along with Two Stage Task Suspension and Task Suspension, are derived from the Entry entitled Controlling When A Task Executes. Three Entries were created to more specifically address the needs for task suspension: simple blocking for services like I/O, intertask communication and synchronization, and asynchronous suspension and resumption, respectively.

Time Slicing

Issue

The minimal scheduling rules defined in the LRM do not provide for setting time slices, nor do they provide a way to dynamically assign or change slices for tasks.

Proposal

This proposal consists of the following package:

```
with TASK_IDS;
package TIME_SLICING is
    subtype SLICE is DURATION <implementation-defined>;
    procedure SET_TIME_SLICE(  OF_TASK : in TASK_IDS.TASK_ID;
                               TO : in SLICE );
    procedure TURN_OFF_TIME_SLICE( OF_TASK : in TASK_IDS.TASK_ID );
end TIME_SLICING;
```

Time slicing is defined to be preemptive round-robin dispatching with a fixed amount of time allocated to each task. In the absence of a call to SET_TIME_SLICE for a particular task, that task is not subject to time slicing. Such a task, once it begins execution, continues until it blocks itself (e.g. for a delay or rendezvous), or until it is preempted by a higher priority task.

In contrast, once SET_TIME_SLICE(OF_TASK => T, TO => D) is called, task T is dispatched preemptively, with time slice D and executes for at most D units of time (or until blocked), before yielding its place to any other tasks of equal priority. The time slice of a task can be changed by calling SET_TIME_SLICE dynamically. The implementation shall document the precision with which time accounting is done. If a task's time slice is dynamically set while it is executing, the new value takes effect immediately.

The procedure TURN_OFF_TIME_SLICE is used to eliminate any time slicing on a specified task. If the specified task has no time slice, the procedure TURN_OFF_TIME_SLICE has no effect.

Discussion

This Entry will permit adjusting the relative proportion of processor time allocated to several ongoing ("background") tasks in a way that cannot be accomplished by means of priorities.

Implementations exist which allow for setting time slices via a pragma, but these require the slice size to be statically set. Another implementation exists that allows the specification of time slicing for all tasks of a specific priority; this implementation does not allow the specification of time slices for individual tasks, unless they each have unique priorities.

Note that this Entry may be incompatible with some operating systems or realtime executives.

Interactions With Other CIFO Entries

The use of time slicing to define scheduling algorithms must consider the conflicts and complex interactions present if other scheduling features are used in conjunction. Users should beware of conflicts involving the use of Time Slicing with the following: Dynamic Priorities, Priority Inheritance Discipline, Synchronous and Asynchronous Task Scheduling, Task Suspension, Two Stage Task Suspension, and Asynchronous Task

Suspension. Generally, the user will want to isolate time sliced tasks from tasks that are configured by those other Entries.

It must also be noted that the use of Time Slicing will negatively affect the following CIFO resource sharing mechanisms, which are generally expected to run quickly and efficiently and could result in deadlock: Shared Locks, Buffers, and Resources.

Dynamic Priorities: If a target task is blocked and waiting for its next time slice, the priority change should take place immediately. Some multiprocessor implementations may not be capable of guaranteeing an immediate effect; in those cases, the priority changes should take place but no later than the next synchronization point of that processor.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: If the time quanta for a task expires while dispatching for that task is disabled, the task does not relinquish control of the CPU until dispatching is enabled. The occurrence of a new time slice does not resume a suspended task. A task loses the rest of its time slice if it is suspended.

Time Critical Sections: If a task's slice expires while the task is within a Time Critical Section, the task will continue to execute until it exits the Time Critical Section.

Asynchronous Transfer of Control: The specification of the time slice for a task applies transitively to an implementation which employs a virtual agent task.

Mutually Exclusive Access to Shared Data: If a time slice expires while writing then the write does not complete until the writing task is rescheduled. Clearly no other tasks can access the shared data during the intervening period.

Interrupt Management: It is dangerous to disable interrupts in a time sliced task.

Fast Interrupt Pragmas: The specification of time slicing on an interrupt task is discouraged, due to the danger that the time slice may expire during an interrupt call and the interrupt fails.

Passive Task Pragma: If a passive task is called by a task with a time slice, the passive task inherits the time slice of the caller. Specifically, it will use the balance of the current time slice, and any of the calling task's future time slices if necessary, until the rendezvous is completed. In effect, a call to a passive task is treated no differently than a call to a procedure.

Changes From The Previous Release

Comments were added to the Discussion section, based on review comments received during the February 1991 ARTEWG meeting. Two sections, Interactions and Changes, were added to the Entry to conform to the new CIFO style.

Synchronization Discipline

Issue

The Ada Reference Manual has intentionally and appropriately left many areas of the Ada language to be defined by the implementation [ARTEWG87]. This has allowed implementations to take the best advantage of the underlying hardware and/or operating system and also permits implementations to adapt to new or different technology. However, many applications have specific requirements in some of these implementation dependent areas of the language. For example, applications may require that for entries of open alternatives in the select statement, the task to be dispatched is the task with the highest priority.

Proposal

Two alternative mechanisms are proposed for specifying criteria for queuing entry calls and selection criteria for selective waits: the packages COMPLEX_DISCIPLINE and SYNCHRONIZATION_DISCIPLINE or the set of four pragmas:

```
with QUEUING_DISCIPLINE;
use QUEUING_DISCIPLINE;
package COMPLEX_DISCIPLINE is

    type TIEBREAK is ( ARBITRARY, FIFO, LEXICAL );

    type SELECT_DISCIPLINE( QD : DISCIPLINE := ARBITRARY_QUEUING ) is
        record
            case QD is
                when QUEUING_DISCIPLINE.PRIORITY_QUEUING =>
                    T : TIEBREAK;
                when others =>
                    null;
            end case;
        end record;

    type SELECT_CRITERIA( LEXICAL_ORDER : BOOLEAN := FALSE ) is
        record
            case LEXICAL_ORDER is
                when TRUE =>     null;
                when FALSE =>    DISCIPLINE : SELECT_DISCIPLINE;
            end case;
        end record;

    ELECTION_ERROR : exception;

end COMPLEX_DISCIPLINE;
```

```
with TASK_IDS;
with QUEUING_DISCIPLINE;
with COMPLEX_DISCIPLINE;
package SYNCHRONIZATION_DISCIPLINE is

    procedure SET_ENTRY_CRITERIA
                  ( OF_TASK : in TASK_IDS.TASK_ID;
                    TO : in QUEUING_DISCIPLINE.DISCIPLINE );

    procedure SET_GLOBAL_ENTRY_CRITERIA
                  ( TO : in QUEUING_DISCIPLINE.DISCIPLINE);

    procedure SET_SELECT_CRITERIA
                  ( OF_TASK : in TASK_IDS.TASK_ID;
                    TO : in COMPLEX_DISCIPLINE.SELECT_CRITERIA );

    procedure SET_GLOBAL_SELECT_CRITERIA
                  ( TO : in COMPLEX_DISCIPLINE.SELECT_CRITERIA );

end SYNCHRONIZATION_DISCIPLINE;
```

Alternative Proposal Solution

An alternative implementation of these features takes the form of pragmas, as described below.

```
pragma SET_ENTRY_CRITERIA
          ( TO : in QUEUING_DISCIPLINE.DISCIPLINE );

pragma SET_GLOBAL_ENTRY_CRITERIA
          ( TO : in QUEUING_DISCIPLINE.DISCIPLINE );

pragma SET_SELECT_CRITERIA
          ( TO : in COMPLEX_DISCIPLINE.SELECT_CRITERIA );

pragma SET_GLOBAL_SELECT_CRITERIA
          ( TO : in COMPLEX_DISCIPLINE.SELECT_CRITERIA );
```

These packages provide the application with the ability to (1) specify the queuing discipline for managing entry queues, and (2) specify the arbitration policy to be used in selective wait statements.

For the pragmas SET_ENTRY_CRITERIA and SET_GLOBAL_ENTRY_CRITERIA, the parameter must be static, or the pragma will be ignored. For the pragmas SET_SELECT_CRITERIA and SET_GLOBAL_SELECT_CRITERIA, the parameter must be an aggregate, and all of its subcomponents must either be aggregates or be static; otherwise the pragma will be ignored.

The properties of the procedures and type definitions in this package are as follows:

SET_ENTRY_CRITERIA

For the specified task all entry queues will be processed according to the discipline requested.

When the discipline specifies "PRIORITY" queuing, then the queues are managed such that processing entry queues has the effect that queued tasks are serviced in priority order. Since this is required for many embedded applications it seems likely that the next version of the Ada LRM will explicitly reflect the legality of this behavior (see [Sha89]).

It may happen that an entry queue contains more than one task whose priority is "highest", i.e. at least as high as that of any other task currently in the entry queue. In this case, the "PRIORITY" discipline chooses the task that has been in the queue at this priority level longest. Note that through priority inheritance, a task that is already waiting in the entry queue may be given a different (higher or lower) priority. For such a task, changing the priority has an effect equivalent to removing the task from the queue and requeuing it at its new priority.

SET_GLOBAL_ENTRY_CRITERIA

This procedure has the same effect as SET_ENTRY_CRITERIA except that it applies to all tasks in the program. The exception QUEUING_DISCIPLINE.UNSUPPORTED_DISCIPLINE should be raised by the RTE if the criteria elected by any of the SET_ services described above is not available. A Run-time Environment that implements this CIFO option does not need to support all the possible arguments for the SET_ subprograms. For example, the implementor may choose not to implement the SPINNING queue discipline and would raise QUEUING_DISCIPLINE.UNSUPPORTED_DISCIPLINE if the SPINNING discipline was requested. An implementation of this CIFO Entry must document the options that are supported.

SET_SELECT_CRITERIA

Whenever a task is executing a select statement for which more than one accept alternative is open, the alternative will be chosen according to the specified discipline.

If the discriminant LEXICAL_ORDER is TRUE, the decision among several eligible accept alternatives will be made according to the lexical order in which the accept alternatives appear in the source text. The first eligible alternative will be chosen.

An implementation does not need to support the case where LEXICAL_ORDER is TRUE. If LEXICAL_ORDER is not supported, this must be documented. In this case, specifying LEXICAL_ORDER in SET_SELECT_CRITERIA or in SET_GLOBAL_SELECT_CRITERIA (see below) should raise the exception COMPLEX_DISCIPLINE.ELECTION_ERROR.

The user can specify any discipline from QUEUING_DISCIPLINE by specifying LEXICAL_ORDER = FALSE and specifying the requested discipline in the field DISCIPLINE. As for SET_ENTRY_CRITERIA and SET_GLOBAL_ENTRY_CRITERIA, an implementation does not need to support all values. The only value required is ARBITRARY_QUEUING.

The implementation needs to document which disciplines are supported, and raise the exception UNSUPPORTED_DISCIPLINE if an unsupported discipline is specified.

When FIFO discipline is chosen, the task at the head of the relevant entry queues that has been in the queue longest is chosen.

When queuing discipline PRIORITY_QUEUING is specified, there may be more than one task at the head of a relevant entry queue such that that task's priority is highest. The user needs to specify(in the component T of type TIEBREAK) how this situation should be resolved.

The value LEXICAL indicates a policy like the one described earlier: eligible alternatives are considered in the lexical order in which they appear in the program text. The value ARBITRARY leaves the policy to the implementation. The value FIFO indicates that among the eligible "highest priority" tasks at the heads of the different entry queues, the one will be chosen that has been waiting at its current priority longest.

An implementation only needs to support the value ARBITRARY. It should be documented which values are supported, and if the user specifies an unsupported value, the exception COMPLEX_DISCIPLINE.ELECTION_ERROR should be raised.

SET_GLOBAL_SELECT_CRITERIA

This procedure has the same effect as SET_SELECT_CRITERIA except that the queuing discipline applies to all tasks in the program.

Alternative Pragma-based Proposal

An equally important interface to the functions characterized in this CIFO Entry is the pragma. While the procedural interface does not necessarily require modifications to the compiler and provides definite indications of support, it does not permit some types of optimizations. A pragma interface would permit early detection and perhaps allow additional optimizations to be performed.

In some situations these additional optimizations may be necessary to meet efficiency requirements. In other situations, changing the compiler may not be a viable solution. Therefore, this CIFO Entry defines an alternative solution. If either package SYNCHRONIZATION_DISCIPLINE or the pragmas as defined below are supported, then this CIFO Entry is satisfied. The above pragmas, when supported, satisfy this CIFO Entry.

These pragmas have the properties described by the corresponding procedures in the package above. Pragma SET_ENTRY_CRITERIA and SET_SELECT_CRITERIA must be in the task specification. The other pragmas must be located in the main procedure specification or the declarative part of the main procedure in the event that the main procedure does not have a specification.

This package requires the presence of the TASK_IDs package and the QUEUING_DISCIPLINE package. The TASK_IDS package exception TASK_ID_ERROR will be raised if an unknown task is specified.

Discussion

Until the default queuing disciplines are overridden by this package, they are as specified by the LRM (i.e., FIFO for entry queues and implementation dependent for select alternatives). The queuing disciplines may be changed on a task-by-task basis as well as on a global level. Changing the queuing discipline on a task-by-task basis will result in different queuing disciplines (at least momentarily, assuming DISABLE_PREEMPTION is not in effect during the altering of queue disciplines) for the various entry queues and/or open select alternatives. Under these types of circumstances the individual entry queues and select alternatives will be processed as specified by the queuing discipline in effect. As an example, one or more entry queues may be processed on a first-in, first-out order while others are processed in a priority order (reference the CIFO Queuing Discipline entry).One intended use of this CIFO entry is to use SET_GLOBAL_ENTRY_CRITERIA and SET_GLOBAL_SELECT_CRITERIA in conjunction with the SET_PRIORITY_INHERITANCE_CRITERIA capability defined in the Priority Inheritance Discipline CIFO Entry to eliminate all unbounded priority inversion. (See the Priority Inheritance Discipline CIFO Entry.) This would allow such approaches as the Rate Monotonic Analysis Theory to be used.

Interactions With Other CIFO Entries

Use of SET_GLOBAL_ENTRY_CRITERIA (or pragma SET_GLOBAL_ENTRY_CRITERIA) defines the queuing discipline for task entry queues. CIFO Entries implemented using task entry queues and dependent on a particular queuing discipline (e.g., FIFO) may be in jeopardy of undesirable entry queue behavior if the queuing discipline is altered. Shared Locks, Resource Control, Buffers and Mutually Exclusive Access to Shared Data may potentially suffer from this side effect.

Dynamic Priorities: If a task is suspended in an entry queue for which the queuing discipline is PRIORITY, and if this task's priority is changed, the task's position in the entry queue is changed accordingly.

Priority Inheritance: One intended use of the Synchronization Discipline CIFO Entry is to use SET_GLOBAL_ENTRY_CRITERIA and SET_GLOBAL_SELECT_CRITERIA in conjunction with the SET_PRIORITY_INHERITANCE_CRITERIA capability defined in the Priority Inheritance Discipline CIFO Entry to eliminate all unbounded priority inversion (see Priority Inheritance Discipline CIFO Entry). This would allow such approaches as the Rate Monotonic Theory to be used.

Asynchronous Task Suspension: If a task is blocked in an entry queue, then the operation HOLD_TASK for this task has no effect. The suspension due to HOLD_TASK will occur when the task is unblocked. Therefore, there is no interaction of this CIFO Entry with Asynchronous Task Suspension.

Time Slicing: Time slicing tasks can potentially run counter to the entry queuing disciplines and selection criteria disciplines provided by this CIFO Entrv.

Changes From The Previous Release

This is a new CIFO Entry.

References

[ARTEWG87] "Catalogue of Ada Runtime Implementation Dependencies", ACM Special Interest Group on Ada, Ada Runtime Environment Working Group, (1987).

[Sha89] "Real-Time Scheduling Theory and Ada", Sha. L. and Goodenough, J., Technical Report, SEI, CMU, (1989).

Asynchronous Cooperation Mechanisms with Reference Memory Model

The Ada language tasking model offers only one safe mechanism for cooperation between tasks : the rendezvous. The rendezvous provides both synchronization and communication facilities within the same mechanism.

Well-known asynchronous services (events, semaphores, ...) are not provided by Ada. Nevertheless, a real need for these services exists in the hard realtime Ada applications. The need comes from requirements to accommodate existing application designs; requirements to support asynchronous communication and signaling operations; requirements for better application performance; and requirements for better application portability and reuse.

Such services can be realized in Ada using the rendezvous mechanism. However, this is at the cost of extra server tasks and rendezvous operations. Some optimization techniques (such as task passivation) have been defined to reduce the costs of such servers. Unfortunately, such optimizations are not readily available in a portable way across a wide range of existing Ada implementations.

The entries in this chapter provide a full set of Asynchronous Cooperation Mechanisms and have a two-fold objective:

1) To define a uniform set of asynchronous services that fit the identified needs for functionality, performance and reuse in hard real-time systems.

2) To allow efficient implementations of these services on top of available Ada technologies by adapting (optimizing "by hand") their bodies to the underlying Ada Runtime Executives or to hardware solutions.

These objectives match the following rationale: uniformity of available services promotes the portability of Ada realtime applications design and programming; and specifications must not be such that existing RTE's are to be completely reworked to be able to implement the defined mechanisms efficiently.

Thus, the services have been designed using Ada abstractions and Ada semantics so that they can be implemented with Ada tasking facilities. A set of model bodies, entirely written in Ada, exists and provides a semantic reference for the defined services.

These services represent a coherent model of Asynchronous Cooperation Mechanisms that promote clean, efficient application architectures which avoid non-portable solutions.

This set of entries is included in a separate CIFO chapter because they share a common memory reference model, which is discussed below.

Asynchronous Cooperation Mechanisms

The proposed mechanisms can be classified in categories, each of which may contain two different mechanisms: one providing synchronization (without communication), and one providing synchronization with communication, through an associated message

The categories, and associated entries, are:

a. counters : the Resources Entry and the Buffers Entry;

b. states (level-sensitive synchronization): the Events Entry and the Blackboards Entry;

c. pulses (edge-sensitive synchronization): the Pulses Entry andthe Broadcasts Entry;

d. the Barriers Entry, which provides synchronization for a fixed number of tasks.

Discussion

The elements of this discussion apply to all the Asynchronous Cooperation Mechanisms entries in this chapter.

Ada Conformance

Defining cooperation mechanisms as strict Ada-conforming tools and implementing a model of them:

a. allows the development of full Ada semantic model bodies,

b. keeps the consistency of the Ada tasking model within the entire application,

c. enables the use of native Ada bodies when performance requirements are not stressing or when compiler optimizations are / will be sufficient to fulfill the performance requirements,

d. avoids most tricky semantics issues such as: what happens if a procedure is used instead of a task entry; what happens with exceptions raised inside the called chain; and how is the memory managed? These issues are avoided because the Ada model bodies provide a semantic reference for each service.

e. allows early efficient implementations over existing RTEs (Ada specific RTEs or more general ones): a simple passivation, by hand, of the different server tasks can provide significant improvements of such asynchronous services,

f. avoids being dependent on a particular group of RTEs.

Thus, compatibility with the Ada tasking model is a major principle of this set of Asynchronous Cooperation Entries. It follows that the interface to these mechanisms must be identical to making entry calls into tasks, and waiting in the queue associated with those entries. The behavior of the services must be identical to one or more tasks with accepts, possibly within accepts.

Note that, although the select statement can be used to make a "selective wait" for several accepts, a companion "selective call" does not exist. Thus a task cannot do several entry calls, while waiting for the first rendezvous to happen. Consequently, an Ada task cannot be waiting in more than one queue. This limitation prevents defining the handling of complex synchronization expressions. In particular, the Resources Entry does not support waiting for multiple resources and the Events Entry does not support waiting on more than one event.

Separate Entries for Mechanisms With Messages and Without Messages

The Buffer, Blackboard and Broadcast mechanisms are not just high level mechanisms that could be built efficiently using more basic and primitive mechanisms, such as the synchronization-only mechanisms contained in this chapter. Because Resources, Pulses, and Events are not Ada synchronization points, one cannot safely use them to build shared data structures, such as Buffers/Blackboards/Broadcasts; compiler optimizations could eliminate the necessary synchronizations. In addition, the optimization schemes implementing "with message" and "without message" services are quite different. By separating them, both cases can have a more efficient implementation, specifically the "without message" one, which is likely to be used often in applications.

Memory Management Model

All the cooperation mechanisms in this chapter are defined using abstract data types. The cooperation objects that can be created with these types are handled through "accessors". These types are private but non-limited, providing explicit creation and deletion operations as well as implicit assignment and comparison operations.

The "through accessor" handling scheme is provided to cope with hard real-time needs for static behavior and security of systems concerning the data structures handled by the runtime system (as no certifiable, efficient Ada memory management seems to be available at this time). It allows a RTE to handle Asynchronous Cooperation Objects in a private and separated area, managed by the runtime and possibly protected for the sake of security. This also means that the system (RTE) can be separately and statically tailored to the exact needs of the applications (several may coexist and communicate using these tools).

Two possible implementations are described as examples:

A. The first example is the usage of Ada access types to implement the "accessors":

The private part of the package specifications may appear as:

```
private
    type OBJECT_DESCRIPTOR;
    type OBJECT is access OBJECT_DESCRIPTOR;
    NULL_ACCESSOR: constant OBJECT := null;
end OBJECTS;
```

The implementation of CREATE and DESTROY may use the following:

```
function CREATE(  CAPACITY: in CAPACITY_RANGE;
                  NAME: in STRING :="") return OBJECT is
begin
    ...
    return new OBJECT_DESCRIPTOR;
end CREATE;

procedure DESTROY (B: in out OBJECT) is
    new UNCHECKED_DEALLOCATION ( OBJECT    => OBJECT_DESCRIPTOR;
                                 NAME      => OBJECT );
```

B. A second example uses a preallocated pool of cooperation objects that are accessed through some kind of index.

The private part of the package specifications may appear as:

```
private
    MAX_OBJECT: constant := 2**31 -1;
    type OBJECT is range 0 .. MAX_OBJECT;
    NULL_ACCESSOR: constant OBJECT := 0;
end OBJECTS;
```

The implementation of CREATE and DESTROY may use the following:

```
function CREATE(CAPACITY: in CAPACITY_RANGE;
         NAME: in STRING := "" ) return OBJECT is
begin
   -- the function GET_INDEX will pick up
   -- a free index, set it to used, and return it
   return GET_INDEX (CAPACITY, NAME);
end CREATE;

procedure DESTROY(B: in out OBJECT) is
begin
   ...
   -- the procedure DEALLOCATE_INDEX returns an index
   -- to the pool of available indices
   DEALLOCATE_INDEX (OBJECT);
   OBJECT := NULL_ACCESSOR;
   ...
end DESTROY;
```

Objectives of these implementations differ:

a. the first example uses the Ada dynamic memory capability. Hence safeness of this solution relies on the memory management scheme provided by the Ada compiler. It is very likely to be used on virtual memory systems.

b. the second example may use a common statically preconfigured safe set of objects that are managed as a pool.

Full usage of Ada memory management is not the proposed choice. In the model used to handle asynchronous cooperation data objects, objects are created by allocators (CREATE functions) and the object's "accessors" could be Ada access types. Such implementations do not have to rely upon an Ada garbage collector for deallocation when exiting their extended scopes. Dependance on garbage collection is perceived as contrary to the requirement for static configuration, as needed by the current state of the art in certification.

In this scheme, the accessor is a private type that gives access to a unique instance of a cooperation object. It is necessary to be able to handle (assign and compare) such accessor values in order to manage pools of cooperation objects, and to handle partial degradation gracefully. Therefore, the accessor types cannot be limited and their assignment and comparison have semantics similar to those of the Ada access types.

Resources

Issue

Real-time applications need a synchronization object to efficiently control access to a shared hardware resource.

Proposal

The interface is a generic package defining the private RESOURCE type and importing the queuing discipline, which is used to manage the queue of waiting tasks associated with each resource object.

```
with QUEUING_DISCIPLINE;
use QUEUING_DISCIPLINE;
generic
   RESOURCE_QUEUING_DISCIPLINE : DISCIPLINE := FIFO_QUEUING;
package RESOURCES is

   type RESOURCE is private;

   NULL_ACCESSOR: constant RESOURCE;
   IMMEDIATELY: constant DURATION := 0.0;

   NON_EXISTENT_RESOURCE: exception;
   INVALID_RESOURCE_NAME: exception;
   RESOURCE_CAPACITY_OVERFLOW: exception;
   RESOURCE_DESTROYED: exception;

   MAX_CAPACITY : constant POSITIVE := <implementation defined>;
   MAX_WAITING_TASKS : constant POSITIVE :=<implementation defined>;
   type CAPACITY_RANGE is range 1..MAX_CAPACITY;
   type COUNTER_RANGE is range -MAX_WAITING_TASKS..MAX_CAPACITY;
   type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

   function CREATE(  INITIAL : in CAPACITY_RANGE := CAPACITY_RANGE'FIRST;
                     CAPACITY: in CAPACITY_RANGE := CAPACITY_RANGE'LAST;
                     NAME:  in STRING := "" ) return RESOURCE;
   function CAPACITY (R: in RESOURCE) return CAPACITY_RANGE;
   procedure GET (R: in RESOURCE);
   procedure GET( R: in RESOURCE;
                  PASSED: out BOOLEAN;
                  TIME_OUT: in DURATION := IMMEDIATELY);
   procedure RELEASE (R: in RESOURCE);
   procedure DESTROY (R: in out RESOURCE);
   function COUNT (R: in RESOURCE) return WAITING_RANGE;
   function VALUE (R: in RESOURCE) return COUNTER_RANGE;
private
   type RESOURCE is <implementation defined>;
   NULL_ACCESSOR : constant RESOURCE := <implementation defined>;
end RESOURCES;
```

The RESOURCE type, called an accessor, is implementation defined. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

Once created (CREATE) a resource may be obtained (GET) and then released (RELEASE); it may be observed (COUNT, VALUE) and deleted (DESTROY). A resource may be simultaneously owned by a maximum of CAPACITY tasks.

Conceptually, a resource object is composed of:

a. an integer counter (initially positive, default 1),

b. and a queue of waiting tasks (initially empty).

A resource object always satisfies the following invariants:

counter >= 0 mutually implies queue empty

counter <= 0 mutually implies length of queue = abs(counter)

counter = (initial value minus number of successful GET operations plus number of RELEASE operations).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side.

The implicit comparison operator (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the RESOURCE declared operations except CREATE and DESTROY raise NON_EXISTENT_RESOURCE if the value used as an actual parameter for the accessor R is NULL_ACCESSOR.

The function CREATE "creates" a resource. A call to CREATE has the following effects:

a. The internal semantics of this creation are specific to each implementation.

b. The optional INITIAL parameter can be used to give an initial value to the resource counter. The default value is one, meaning that only one non-blocking GET can be successful prior to the first RELEASE.

c. The optional CAPACITY parameter can be used to specify the range of the counter (i.e. the maximum number of simultaneous owners of a "bounded resource"). The default value is the maximum allowed by the implementation.

if INITIAL > CAPACITY then raise RESOURCE_CAPACITY_OVERFLOW.

d. the accessor value that denotes the created resource is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_RESOURCE_NAME.

The function CAPACITY (R := hw_resource) returns the maximum number of simultaneous owners of bounded resource hw_resource.

The procedure GET with one parameter unconditionally "gets" a resource. It waits indefinitely if the resource is "busy". A call to GET has the following effects:

```
counter := counter - 1
if counter < 0 (resource was already "busy") then
    the current task is blocked and added at the end of the waiting queue
    dispatching occurs
    this task may then be resumed by the RELEASE service (see below)
    thus finishing the GET procedure.
end if
```

The procedure GET with three parameters "gets" a resource. GET waits until the resource is not busy or the delay expires, whichever occurs first. The delay is "wall clock time". A "conditional get" (not waiting if the resource is busy) is a GET with a non-positive delay. IMMEDIATELY is a handy symbolic name for the "conditional get" TIME_OUT parameter. A call to GET has the following effects:

```
if counter <= 0 (resource is already busy) then
    PASSED := FALSE
    if TIME_OUT > 0 then
        counter := counter - 1
        the current task is blocked and added to the waiting queue
        a timer is initiated with duration TIME_OUT
        dispatching occurs
        this task may then be resumed:
            either by the expiration of the timer in which case:
                counter := counter + 1
                the task is removed from the waiting queue
            or by the RELEASE service in which case:
                PASSED := TRUE.
                cancel the associated timer.
            thus finishing the GET procedure
    end if
else (resource is free)
    counter := counter - 1
    PASSED := TRUE
end if
```

The procedure RELEASE "releases" a resource. A call to RELEASE has the following effects:

```
if counter = CAPACITY then
    raise RESOURCE_CAPACITY_OVERFLOW
end if
counter := counter + 1
if counter <= 0 (waiting queue was not empty) then
    the next eligible task of the waiting queue is removed from
    it and resumed.
    dispatching occurs.
end if
```

The procedure DESTROY "deletes" the resource denoted by the accessor given as the actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as the actual parameter is set to NULL_ACCESSOR

c. if the waiting queue is not empty all the waiting tasks are resumed and the exception RESOURCE_DESTROYED is propagated to them.

The function COUNT returns the current length of the waiting queue associated with a resource (i.e. the number of tasks waiting to get a resource). Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function VALUE returns the value of a resource (if positive it is the number of possible new owners; if negative it is the number of tasks waiting to get a resource). Using this function may be as unsafe as using the COUNT attribute on a task entry.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

With regard to their semantics, the resource operations are defined as if executed by an Ada server run at the highest software priority.

Discussion

Implementations may raise QUEUING_DISCIPLINE.UNSUPPORTED_DISCIPLINE when instantiated with values other than FIFO_QUEUING.

Resource objects MUST NOT BE USED FOR DATA SHARING because their operations are not synchronization points with respect to data sharing, and therefore optimizers can break the intended protections. The CIFO Entry for Mutually Exclusive Access to Shared Data should be used in order to achieve a safe data sharing.

Resources are NOT automatically released when a task holding them is aborted. Thus, aborting a task which holds a resource makes the resource permanently unavailable, and may lead to a deadlock. Similarly, care should be exercised in taking any action which may block or suspend a task which holds a resource. This entry does not provide protection against deadlock. In some cases the implementation of the RESOURCE operations may propagate an exception to the caller that the caller cannot handle (for example, the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a RESOURCE accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_RESOURCE when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated resources handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purposes. The semantics of the NAME parameter must be documented by each implementation.

In the RESOURCE_CONTROL package, there is no operation to get several resources at the same time. According to the principles stated in the introduction to this chapter, the implementation of such a GET operation (getting a list of resources) would require a task to wait for several resources without appearing in the waiting queues for these resources -- in effect to be queued for a particular list of resources. The "resources list" is thus a new kind of synchronizing object, for which the queuing order is preserved for waiting tasks, although queuing order is not preserved for a resource which appears in several resource lists.

The GET and RELEASE algorithms with "resources lists" are much more complex than in a single resource version. Numerous links between resources and resource list descriptors would be necessary. Dynamic allocation of queues associated with resource lists would also be necessary, as resource lists can be dynamically built with adequate constructors. Such a burden should be avoided and considered only if a serious need for it exists.

Interactions With Other CIFO Entries

Queuing Discipline is referenced in the specification of Resources to establish the behavior of the queues. The Queueing Discipline is thus defined to be static, but when it responds to global change requests, the semantics are exactly the same as for an Ada coded server. Suppliers must be cautious because providing this feature may defeat most known optimization strategies for such servers.

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes in exactly the same way as queues associated with Ada task entries.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration of Program Units: The CREATE operations of the Asynchronous Cooperation Objects are allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Time Slicing: Use of Time Slicing may cause additional blocking of a task when its slice expires. Care should be taken if Time Slicing is used on tasks which hold resources.

Changes From The Previous Release

This is a new Entry.

Events

Issue

Realtime applications need some kind of synchronization object in order to efficiently notify any waiting tasks upon the occurrence of a latched condition.

Proposal

The interface is a package defining the private EVENT type.

```
package EVENTS is

    type EVENT is private;

    type EVENT_STATE is (UP, DOWN);

    NULL_ACCESSOR: constant EVENT;

    IMMEDIATELY: constant DURATION := 0.0;

    NON_EXISTENT_EVENT: exception;
    INVALID_EVENT_NAME: exception;
    EVENT_DESTROYED: exception;

    MAX_WAITING_TASKS : constant := <implementation defined>;
    type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

    function CREATE(  INITIAL: in EVENT_STATE := DOWN;
                      NAME: in STRING := "") return EVENT;

    procedure WAIT (E: in EVENT);
    procedure WAIT(E: in EVENT;
                   PASSED: out BOOLEAN;
                   TIME_OUT: in DURATION := IMMEDIATELY);

    procedure SET (E: in EVENT);
    procedure RESET (E: in EVENT);
    procedure TOGGLE (E: in EVENT);

    procedure DESTROY (E: in out EVENT);
    function COUNT (E: in EVENT) return WAITING_RANGE;

    function STATE (E: in EVENT) return EVENT_STATE;
private
    type EVENT is <implementation defined>;
    NULL_ACCESSOR: constant EVENT := <implementation defined>;
end EVENTS;
```

Once created (CREATE) an event may be set (SET) to the state "condition has occurred" or reset (RESET). Tasks may wait for this occurrence (WAIT). It may be observed (COUNT, STATE) and deleted (DESTROY).

Conceptually, an event is composed of a bivalued state variable (the states are often called up and down), and a set of waiting tasks (initially empty).

An event object always satisfies the following invariants:

a. event = UP implies no waiting task, and

b. event = DOWN implies existence of at least one waiting task.

With regard to their semantics, the event operations are defined as if executed by an Ada server run at the highest software priority.

The EVENT type, called an accessor, is implementation defined. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side thus denoting the same object as the one denoted by the expression.

The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the EVENT declared operations except CREATE and DESTROY raise NON_EXISTENT_EVENT if the value used as an actual parameter for the accessor E is NULL_ACCESSOR.

The function CREATE "creates" an event. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. he initial state is given by the optional INITIAL parameter (default value is down)

c. the accessor value that denotes the created event is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_EVENT_NAME.

The procedure WAIT with one parameter unconditionally "waits" on an event. It waits indefinitely if the event is "down". A call to WAIT has the following effects:

```
if event is down then
    the current task is blocked and added to the set of waiting tasks
    dispatching occurs
    the task may then be resumed by the SET service (see below)
    thus finishing the WAIT procedure
end if
```

The procedure WAIT with three parameters "waits" on an event. It waits until the event is "up" or the delay expires, whichever occurs first. A "conditional wait" (not waiting if the event is down) is a wait with a non-positive delay. IMMEDIATELY is a handy symbolic name for the "conditional wait" TIME_OUT parameter value. A call to WAIT has the following effects:

```
if event is down then
    PASSED := FALSE
    if TIME_OUT > 0 then
        the current task is blocked and added to the set of waiting tasks
        a timer is initiated with duration TIME_OUT
        this task may then be resumed:
            either by the expiration of the timer in which case it is
                removed from the waiting queue
            or by the SET service in which case the timer is cancelled and
                PASSED := TRUE
            thus finishing the WAIT procedure.
    end if
else (event is up)
    PASSED := TRUE
end if
```

The procedure SET "sets" an event in the UP state. A call to SET has the following effects:

```
event := up
if the set of waiting tasks is not empty then
    remove all waiting tasks and resume them all at once
    dispatching occurs
end if
```

The resumes for waiting tasks should appear to be atomic, so that all tasks are available for dispatching at the same time. The order of subsequent execution should only depend on the dispatching policy, and not on the order in which the tasks were allowed to resume.

If several tasks are of the same priority, the application should not assume that they will be dispatched in any particular order, specifically the application should not assume FIFO order.

The procedure RESET "resets" an event in the DOWN state. A call to RESET has the following effects:

```
event := down
```

The procedure TOGGLE changes the state of an event to the other possible state. A call to TOGGLE has the following effects:

```
if event is down then
    call the SET procedure
else (event is up)
    call the RESET procedure
end if
```

The procedure DESTROY "deletes" the event denoted by the accessor given as the actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as the actual parameter is set to NULL_ACCESSOR

c. if the set of waiting tasks is not empty all the waiting tasks are resumed and the exception EVENT_DESTROYED is propagated to them.

The function COUNT returns the number of tasks waiting on an event. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function STATE returns the state of the event. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

Discussion

In some cases the implementation of the EVENT operations may propagate an exception to the caller that the caller cannot handle (for example, the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a EVENT accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_EVENT when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated events handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Two kinds of events are to be considered for asynchronous communication using events (with or without messages):

1.) Those for which model bodies (could) exist in Ada, as proposed here. Being built within the Ada tasking paradigm, they must be viewed as local to an Ada processing hierarchy (as defined by the task dependency and termination rules). However, despite this logical view, the proposed specification allows that these events could be implemented in a central area managed by the RTE.

2.) Semantically global events which, by nature, are often found within RTEs, and are even part of their architectures.

Experience shows that an efficient compromise between Ada and centrally controlled ones can be found. Further, it would be undesirable if the implementation of this package should require a full reworking of Ada Runtime Executives.

According to the principles stated in the introduction to this chapter, the implementation of multi-event waits would require a task to wait for several events without appearing in the waiting queues for these events - in effect to be queued for a particular set of events. The "event list" is thus a new kind of synchronizing object, for which the queuing order is preserved for waiting tasks, although queuing order is not preserved for an event which appears in several event sets. Such a burden is considered unnecessary.

In addition, event expressions are somewhat complicated to handle:

a. their semantics (abilities and restrictions) are specific to each RTE family (architecture); finding an agreement about real needs is not easy,

b. the memory management scheme of complex expressions and waiting tasks is also related to RTE architecture. (For example, an event could be a member of two different simultaneous waiting conditions that become simultaneously true.)

Interactions With Other CIFO Entries

The EVENT type defined by this Entry is used in the specification of the SCHEDULER package provided by the CIFO Entry Synchronous and Asynchronous Task Scheduling.

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes in exactly the same way as queues associated with Ada task entries do.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This Entry replaces the support for Events which was part of Synchronous and Asynchronous Task Scheduling. The current proposal only supports simple events; the previous version of Synchronous and Asynchronous Task Scheduling supported complex events.

Pulses

Issue

Realtime applications need some kind of synchronization object in order to efficiently notify any waiting tasks upon the occurrence of a pulsed condition.

Proposal

The interface is a package defining the private PULSE type.

```
        package PULSES is

            type PULSE is private;

            NULL_ACCESSOR: constant PULSE;

            IMMEDIATELY : constant duration := 0.0;

            NON_EXISTENT_PULSE: exception;
            INVALID_PULSE_NAME: exception;
            PULSE_DESTROYED: exception;

            MAX_WAITING_TASKS : constant POSITIVE := <implementation defined>;

            type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

            function CREATE (NAME: in STRING := "") return PULSE;

            procedure WAIT (P: in PULSE);
            procedure WAIT(P: in PULSE;
                            PASSED: out BOOLEAN;
                            TIME_OUT: in DURATION := IMMEDIATELY);

            procedure SET (P: in PULSE);

            procedure DESTROY (P: in out PULSE);

            function COUNT (P: in PULSE) return WAITING_RANGE;

        private
            type PULSE is <implementation defined>;
            NULL_ACCESSOR: constant PULSE := <implementation defined>;
        end PULSES;
```

Once created (CREATE) a pulse may be set (SET) to notify such an occurrence to the tasks waiting for it (WAIT). This occurrence is not latched. A pulse may be observed (COUNT) and deleted (DESTROY).

Conceptually, a pulse object is composed of a set of waiting tasks (initially empty).

With regard to their semantics, the pulse operations are defined as if executed by an Ada server run at the highest software pr:.ority.

The PULSE type, called an accessor, is implementation defined. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side.

The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the PULSE declared operations excep REATE and DESTROY raise NON_EXISTENT_PULSE if the value used as an actual parameter for the ac essor P is NULL_ACCESSOR.

The function CREATE "creates" a pulse. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the accessor value that denotes the created pulse is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_PULSE_NAME.

The procedure WAIT with one parameter unconditionally "waits" on a pulse. A call to WAIT has the following effects:

```
the current task is blocked and added to the set of waiting tasks
disp  ching occurs
the task may then be resumed by the SET service (see below)
thus finishing the WAIT procedure
```

The procedure WAIT with three parameters "waits" on a pulse. It waits until the pulse occurs or the delay expires, whichever occurs first. A call to WAIT has the following effects:

```
PASSED := FALSE
if TIME_OUT > 0 then
    the current task is blocked and added to the set of waiting tasks
    a timer is initiated with duration TIME_OUT
    dispatching occurs
    this task may then be resumed:
        either by the expiration of the timer in which case
            it is removed from the waiting queue
        or by the SET service in which case
            the timer is cancelled
            PASSED := TRUE
    thus finishing the WAIT procedure.
end if
```

The procedure SET "sets" a pulse. A call to SET has the following effects:

```
if set of waiting tasks is not empty then
    for all the tasks of the set of waiting tasks do
        remove the task from the set and resume it
    end do
    dispatching occurs
end if
```

The set of tasks waiting on a pulse are resumed in an order that is defined by the implementation. A program relying on this order is erroneous.

The procedure DESTROY "deletes" the pulse denoted by the accessor given as actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as actual parameter is set to NULL_ACCESSOR

c. if the set of waiting tasks is not empty all the waiting tasks are resumed and the exception PULSE_DESTROYED is propagated to them.

The function COUNT returns the number of tasks waiting on a pulse. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

Discussion

In some cases the implementation of the PULSE operations may propagate to the caller an exception that they cannot handle (e.g. the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a PULSE accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_PULSE when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated pulses handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Interactions With Other CIFO Entries

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes exactly the same way as queued associated to Ada task entries do.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration Of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical, virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This is a new Entry.

Buffers

Issue

Real-time applications need some kind of communication object in order to efficiently communicate asynchronously between tasks.

Proposal

The interface is a generic package defining the private BUFFER type.

```
with QUEUING_DISCIPLINE;
use QUEUING_DISCIPLINE;
generic
    type MESSAGE is private;
    DEFAULT_MESSAGE : in MESSAGE;
    RECEIVE_QUEUING_DISCIPLINE : DISCIPLINE := FIFO_QUEUING;
    SEND_QUEUING_DISCIPLINE : DISCIPLINE := FIFO_QUEUING;
package BUFFERS is

    type BUFFER is private;

    NULL_ACCESSOR: constant BUFFER;

    IMMEDIATELY: constant DURATION := 0.0;

    NON_EXISTENT_BUFFER: exception;
    INVALID_BUFFER_NAME: exception;
    NON_EXISTENT_MESSAGE: exception;
    BUFFER_DESTROYED: exception;

    MAX_CAPACITY : constant POSITIVE := <implementation defined>;
    MAX_WAITING_TASKS : constant POSITIVE := <implementation defined>;

    type CAPACITY_RANGE is range 1..MAX_CAPACITY;

    type MESSAGE_COUNT_RANGE is range 0..MAX_CAPACITY;

    type COUNTER_RANGE is range -MAX_WAITING_TASKS .. MAX_CAPACITY;

    type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

    function CREATE(  CAPACITY: in CAPACITY_RANGE := 1;
                      NAME: in STRING := "" ) return BUFFER;

    function CAPACITY (B: in BUFFER) return CAPACITY_RANGE;

    procedure RECEIVE( R: in BUFFER; M: out MESSAGE);
    procedure RECEIVE( B: in BUFFER;
                       M: out MESSAGE;
                       PASSED: out BOOLEAN;
                       TIME_OUT: in DURATION := IMMEDIATELY);

    procedure SEND( B: in BUFFER; M: in MESSAGE);

    procedure SEND(  B: in BUFFER;
                     M: in MESSAGE;
                     PASSED: out BOOLEAN;
                     TIME_OUT: in DURATION := IMMEDIATELY);

    procedure DESTROY( B: in out BUFFER);

    function RECEIVE_COUNT( B: in BUFFER) return WAITING_RANGE;

    function SEND_COUNT( B: in BUFFER) return WAITING_RANGE;

    function MESSAGE_COUNT( B: in BUFFER) return MESSAGE_COUNT_RANGE;

    function MESSAGE_VALUE( B: in BUFFER; I: in CAPACITY_RANGE :=1)
          return MESSAGE;

private
    type BUFFER is <implementation defined>;
    NULL_ACCESSOR: constant BUFFER := <implementation defined>;
end BUFFERS;
```

Once created (CREATE) a buffer may be used by tasks to send (SEND) typed messages. Such messages may be consumed (RECEIVE). CAPACITY messages may be kept in the buffer. A buffer may be observed (SEND_COUNT, RECEIVE_COUNT, MESSAGE_COUNT, MESSAGE_VALUE) and deleted (DESTROY).

Conceptually, a buffer object is composed of:

a. a bounded FIFO queue of messages (initially empty),

b. a counting semaphore to manage the tasks waiting to receive a message (the initial value of the semaphore is 0, meaning that it is free),

c. and a counting semaphore to manage the tasks waiting to send a message (the initial value of the semaphore is CAPACITY, which is the maximum number of messages in the BUFFER).

A buffer object always satisfies the following invariants:

a. queue of messages is not empty mutually implies "receive semaphore" is free

b. queue of messages is not full mutually implies "send semaphore " is free

c. queue of messages is empty mutually implies "receive semaphore" is busy

d. queue of messages is full mutually implies "send semaphore " is busy

e. length of message queue = (number of successful SEND operations minus number of successful

RECEIVE operations).

With regard to their semantics, the buffer operations are defined as if executed by an Ada server run at the highest software priority.

The formal generic parameter MESSAGE type conveys the description of the data copied during the communication. The use of access types within the data is as unsafe as handling access types in subprogram or entry parameters.

The formal generic parameters RECEIVE_QUEUING_DISCIPLINE and SEND_QUEUING_DISCIPLINE convey the disciplines used to manage the receiving and sending queues associated with each buffer object.

The BUFFER type, called an accessor, is implementation defined. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side.

The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).All the BUFFER declared operations except CREATE and DESTROY raise NON_EXISTENT_BUFFER if the value used as an actual parameter for the accessor B is NULL_ACCESSOR.

The function CREATE "creates" a buffer. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the length of the message queue is given by the CAPACITY parameter

c. the accessor value that denotes the created buffer is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_BUFFER_NAME.

The function CAPACITY returns the maximum number of messages in the buffer.

The procedure RECEIVE (with two parameter) unconditionally "receives" a message from a buffer. RECEIVE waits indefinitely if the buffer is "empty". A call to RECEIVE has the following effects:

```
GET ("get semaphore ");
take the first message from the buffer
RELEASE ("send semaphore ");
```

The procedure RECEIVE with four parameters "receives" a message from a buffer. It waits until the buffer is "not empty" or the delay expires, whichever occurs first. A "conditional receive" (not waiting if the buffer is empty) is a receive with a non-positive delay. IMMEDIATELY is a handy symbolic name for the "conditional receive" TIME_OUT parameter. A call to RECEIVE has the following effects:

```
GET ("get semaphore ", PASSED, TIME_OUT);
if PASSED then
    take the first message of the buffer
    RELEASE ("send semaphore ");
end if
```

The procedure SEND (with two parameters) unconditionally "sends" a message to a buffer. It waits indefinitely if the buffer is "full". A call to SEND has the following effects:

```
GET ("send semaphore ");
add the message at the end of the buffer
RELEASE ("get semaphore ");
```

The message M is a copy transmitted to SEND. After the call to SEND, the value used as the actual parameter may be updated without any effect on the message copy.

The procedure SEND with four parameters "sends" a message to a buffer. It waits until the buffer is "not full" or the delay expires, whichever occurs first. A "conditional send" (not waiting if the buffer is full) is a send with a non-positive delay. IMMEDIATELY is a handy symbolic name for the "conditional receive" TIME_OUT parameter. A call to SEND has the following effects:

```
GET ("send semaphore ", PASSED, TIME_OUT);
if PASSED then
    add the message at the end of the buffer
    RELEASE ("get semaphore ");
end if
```

The procedure DESTROY "deletes" the buffer denoted by the accessor given as actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as actual parameter is set to NULL_ACCESSOR

c. if the waiting queue is not empty all the waiting tasks are resumed and the exception BUFFER_DESTROYED is propagated to them.

The function RECEIVE_COUNT returns the length of the queue of tasks waiting to receive a message from a buffer (i.e. "receive semaphore" count). Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function SEND_COUNT returns the length of the queue of tasks waiting to send a message into a buffer (i.e. "send semaphore" count). Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function MESSAGE_COUNT returns the current number of messages in the message queue. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function MESSAGE_VALUE returns the I-th message without removing it from the buffer. Using this function may be as unsafe as using the COUNT attribute on a task entry.A call to the function MESSAGE_VALUE has the following effects:

```
if either I > CAPACITY,or the I-th message does not exist then
    raise NON_EXISTENT_MESSAGE
end if
```

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

Discussion

Implementations may raise QUEUING_DISCIPLINE.UNSUPPORTED_DISCIPLINE when instantiated with values other than FIFO_QUEUING.

For the sake of simplicity, messages are always sent and received in the FIFO order. This message queuing discipline cannot be changed.

In some cases the implementation of the BUFFER operations may propagate an exception to the caller which the caller cannot handle (for example, the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a BUFFER accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_BUFFER when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated buffers handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Interactions With Other CIFO Entries

Queuing Discipline is referenced in the specification of Buffers to establish the behavior of the queues. The queueing discipline is thus defined to be static, but responds to global change requests, the semantics is exactly the same as for an Ada coded server. Suppliers must be cautious because providing this feature may defeat most known optimization strategies for such servers.

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes in exactly the same way as queues associated with Ada task entries.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This Entry is a new entry.

Blackboards

Issue

Realtime applications need some kind of communication object in order to efficiently make messages visible between tasks.

Proposal

The interface is a generic package defining the private BLACKBOARD type.

```
generic
    type MESSAGE is private;
    DEFAULT_MESSAGE : in MESSAGE;
package BLACKBOARDS is

    type BLACKBOARD is private;

    type BLACKBOARD_STATE is (VALID, INVALID);

    NULL_ACCESSOR: constant BLACKBOARD;
    IMMEDIATELY: constant DURATION := 0.0;

    NON_EXISTENT_BLACKBOARD: exception;
    INVALID_BLACKBOARD_NAME: exception;
    BLACKBOARD_DESTROYED: exception;

    MAX_WAITING_TASKS : constant POSITIVE := <implementation defined>;
    type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

    function CREATE (NAME: in STRING :="") return BLACKBOARD;
    function CREATE(  INITIAL: in MESSAGE;
                      NAME: in STRING := "") return BLACKBOARD;

    procedure READ(B: in BLACKBOARD; M: out MESSAGE);
    procedure READ(B: in BLACKBOARD;
                   M: out MESSAGE;
                   PASSED: out BOOLEAN;
                   TIME_OUT: in DURATION := IMMEDIATELY);

    procedure DISPLAY(B: in BLACKBOARD; M: in MESSAGE);
    procedure CLEAR (B: in BLACKBOARD);
    procedure DESTROY (B: in out BLACKBOARD);
    function COUNT (B: in BLACKBOARD) return WAITING_RANGE;
    function STATE (B: in BLACKBOARD) return BLACKBOARD_STATE;
private
    type BLACKBOARD is <implementation defined>;
    NULL_ACCESSOR: constant BLACKBOARD:= <implementation defined>;
end BLACKBOARDS;
```

Once created (CREATE) a blackboard may be written on (DISPLAY). The written typed information may be read (READ) by tasks until cleared (CLEAR). A blackboard may be observed (COUNT, STATE) and deleted (DESTROY).

Conceptually, a blackboard object is composed of:

a. a variable of type MESSAGE (to display the message),

b. a validity indicator for the message,

c. a set of waiting tasks (initially empty).

A blackboard object always satisfies the following invariants:

a. blackboard valid implies no waiting task

b. set of waiting tasks non empty implies blackboard invalid.

With regard to their semantics, the blackboard operations are defined as if executed by an Ada server run at the highest software priority.

The formal generic parameter MESSAGE type conveys the description of the data copied during the communication. The use of access types within such data is as unsafe as handling access types in subprogram or entry parameters.

The BLACKBOARD type, called an accessor, is implementation dependent. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side. The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the Blackboard declared operations except CREATE and DESTROY raise NON_EXISTENT_BLACKBOARD if the value used as an actual parameter for the accessor B is NULL_ACCESSOR.

The function CREATE with one parameter "creates" a blackboard. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the blackboard is initialized in the INVALID state

c. the accessor value that denotes the created buffer is returned.

The function CREATE (with two parameter) "creates" a blackboard. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the blackboard is initialized in the VALID state and with the message INITIAL

c. the accessor value that denotes the created buffer is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_BLACKBOARD_NAME.

The procedure READ (with two parameters) unconditionally "reads" a blackboard, while waiting indefinitely if the blackboard is "invalid". A call to READ has the following effects:

```
if blackboard invalid then
    the current task is blocked and added to the set of waiting tasks
    dispatching occurs
    this task may be resumed by the DISPLAY service (see below)
    the displayed message is returned in M
    thus finishing the READ procedure
else
    the currently displayed message is returned in M
end if
```

The procedure READ with three parameters "reads" a blackboard. It waits until the blackboard is "valid" or the delay expires, whichever occurs first. A "conditional read" (not waiting if the blackboard is invalid) is a read with a non-positive delay. IMMEDIATELY is a handy symbolic name for the "conditional read" TIME_OUT parameter value. A call to READ has the following effects:

```
if blackboard invalid then
    PASSED := FALSE
    if TIME_OUT > 0 then
        the current task is blocked and added to the set of waiting tasks
        a timer is initiated with duration TIME_OUT
        dispatching occurs
        this task may then be resumed:
            either by the expiration of the timer in which case
                it is removed from the set of waiting tasks
                and DEFAULT_MESSAGE is returned in M
            or by the DISPLAY service in which case
                the timer is cancelled and PASSED := TRUE
                and the displayed message is returned in M
            thus finishing the READ procedure.
    end if
else (blackboard is valid)
    PASSED := TRUE
    the currently displayed message is returned in M
end if
if PASSED is FALSE the out parameter M is set to DEFAULT_MESSAGE.
```

The procedure DISPLAY "displays" a message on a blackboard (possibly replacing a previously displayed message if blackboard was already VALID). A call to DISPLAY has the following effects:

```
blackboard := valid and displaying message M
if set of waiting tasks is not empty then
    for all the tasks of the set of waiting tasks do
        remove the task from the set and resume it
    end do
    dispatching occurs
end if
```

The set of tasks waiting on a blackboard are resumed in an order that is defined by the implementation. A program relying on the order is erroneous.

The message M is a copy transmitted to DISPLAY. After the call to DISPLAY, the value used as the actual parameter may be updated without any effect on the message copy.

The procedure CLEAR "clears" a blackboard. A call to CLEAR has the following effects:

```
        blackboard := invalid
```

The procedure DESTROY "deletes" the blackboard denoted by the accessor given as actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as actual parameter is set to NULL_ACCESSOR

c. if the set of waiting tasks is not empty all the waiting tasks are resumed and the exception BLACKBOARD_DESTROYED is propagated to them.

The function COUNT returns the number of tasks waiting on a blackboard. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function STATE returns the state of the blackboard. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

Discussion

In some cases the implementation of the BLACKBOARD operations may propagate an exception to the caller which the caller cannot handle (for example, the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a BLACKBOARD accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_BLACKBOARD when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated blackboards handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Interactions With Other CIFO Entries

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when blocking or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes exactly the same way as queued associated to Ada task entries do.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline. The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This is a new Entry.

Broadcasts

Issue

Realtime applications need some kind of communication object in order to efficiently broadcast messages to multiple tasks.

Proposal

The interface is a generic package defining the private BROADCAST type.

```
generic
    type MESSAGE is private;
    DEFAULT_MESSAGE : in MESSAGE;
package BROADCASTS is

    type BROADCAST is private;

    NULL_ACCESSOR: constant BROADCAST;

    NON_EXISTENT_BROADCAST: exception;
    INVALID_BROADCAST_NAME: exception;
    BROADCAST_DESTROYED: exception;

    MAX_WAITING_TASKS : constant POSITIVE := <implementation defined>;
    type WAITING_RANGE is range 0..MAX_WAITING_TASKS;

    function CREATE (NAME: in STRING := "") return BROADCAST;

    procedure RECEIVE(B: in BROADCAST; M: out MESSAGE);
    procedure RECEIVE(   B: in BROADCAST;
                         M: out MESSAGE;
                         PASSED: out BOOLEAN;
                         TIME_OUT: in DURATION );

    procedure SEND(B: in BROADCAST; M: in MESSAGE);

    procedure DESTROY (B: in out BROADCAST);
    function COUNT (B: in BROADCAST) return WAITING_RANGE;
private
    type BROADCAST is <implementation defined>;
    NULL_ACCESSOR: constant BROADCAST := <implementation defined>;
end BROADCASTS;
```

Once created (CREATE) a broadcast may be used to send (SEND) a typed message to all the tasks waiting for it (RECEIVE). This message is consumed by a RECEIVE operation. A broadcast may be observed (COUNT) and deleted (DESTROY).

Conceptually, a broadcast object is composed of a set of tasks waiting for a message(initially empty).

The formal generic parameter MESSAGE type conveys the description of the data copied during the communication. The use of access types within such data is as unsafe as handling access types in subprogram or entry parameters.

The BROADCAST type, called an accessor, is implementation dependent. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side.

The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the Broadcast declared operations except CREATE and DESTROY raise NON_EXISTENT_BROADCAST if the value used as an actual parameter for the accessor B is NULL_ACCESSOR.The function CREATE "creates" a broadcast. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the accessor value that denotes the created buffer is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_BROADCAST_NAME.

The procedure RECEIVE (with two parameters) unconditionally "receives" a message from a broadcast. A call to RECEIVE has the following effects:

```
the current task is blocked and added to the set of waiting tasks dispatching occurs
this task may be resumed by the SEND service (see below)
the supplied message (see below) is returned in M thus
finishing the RECEIVE procedure
```

The procedure RECEIVE with four parameters "receives" a message from a broadcast. It waits until a message is sent or the delay expires, whichever occurs first. A call to RECEIVE has the following effects:

```
PASSED := FALSE
if TIME_OUT > 0 then
    the current task is blocked and added to the set of waiting tasks
    a timer is initiated with duration TIME_OUT
    dispatching occurs
    this task may then be resumed:
        either by the expiration of the timer in which case
            it is removed from the set of waiting tasks
            DEFAULT_MESSAGE is returned in M
        or by the SEND service in which case
            the timer is cancelled
            PASSED := TRUE
            the supplied message is returned in M
        thus finishing the RECEIVE procedure.
end if
```

The procedure SEND "broadcasts" a message. A call to SEND has the following effects:

```
if the set of waiting tasks is not empty then
for all the tasks in the set of waiting tasks do
    remove the task from the set and resume it supplying a copy of M
end do
dispatching occurs
end if
```

The set of tasks waiting on a broadcast are resumed in an order that is defined by the implementation. A program relying on that an order is erroneous.

The procedure DESTROY "deletes" the broadcast denoted by the accessor given as actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as actual parameter is set to NULL_ACCESSOR

c. if the set of waiting tasks is not empty all the waiting tasks are resumed and the exception BROADCAST_DESTROYED is propagated to them.

The function COUNT returns the number of tasks waiting on a broadcast. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

With regard to their semantics, the broadcast operations are defined as if executed by an Ada server run at the highest software priority.

Discussion

In some cases the implementation of the BROADCAST operations may propagate an exception to the caller which the caller cannot handle (for example, the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a BROADCAST accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_BROADCAST when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated broadcasts handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Interactions With Other CIFO Entries

Time Critical Sections: Calls to primitives available on Asynchronous Cooperation Objects may be blocking When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes in exactly the same way as queues associated with Ada task entries.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration Of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration, if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This is a new Entry.

Barriers

Issue

Realtime applications need some kind of synchronization object in order to efficiently control the simultaneous resumption of some fixed number of waiting tasks.

Proposal

The interface is a package defining the private BARRIER type.

```
package BARRIERS is

    type BARRIER is private;

    NULL_ACCESSOR: constant BARRIER;

    NON_EXISTENT_BARRIER : exception;
    INVALID_BARRIER_NAME  : exception;
    BARRIER_DESTROYED     : exception;

    MAX_CAPACITY : constant POSITIVE := <implementation defined>;
    type CAPACITY_RANGE is range 1..MAX_CAPACITY;

    type WAITING_RANGE is range 0..MAX_CAPACITY;

    function CREATE(  CAPACITY: in CAPACITY_RANGE;
                      NAME: in STRING := "") return BARRIER;

    function CAPACITY (B: in BARRIER) return CAPACITY_RANGE;

    procedure WAIT (B: in BARRIER);

    procedure DESTROY (B: in out BARRIER);
    function COUNT (B: in BARRIER) return WAITING_RANGE;
    function VALUE (B: in BARRIER) return CAPACITY_RANGE;
private
    type BARRIER is <implementation defined>;
    NULL_ACCESSOR: constant BARRIER := <implementation defined>;
end BARRIERS;
```

Once created (CREATE) a barrier may be waited at (WAIT) until CAPACITY tasks are waiting. When this condition occurs the CAPACITY waiting tasks are resumed simultaneously. A barrier may be observed (COUNT, VALUE, CAPACITY) and deleted (DESTROY). Conceptually, a barrier object is composed of:

a. a non negative counter (initially equal to the capacity of the barrier)

b. a pulse to manage the tasks waiting at the barrier.

A Barrier object always satisfies the invariants number of waiting tasks = (capacity - counter).

The BARRIER type, called an accessor, is implementation defined. It is conceptually similar to an Ada access type (namely for :=, =, /= operators) but does not necessarily imply dynamic allocation and management of the accessed objects (see Asynchronous Cooperation Mechanisms Entry discussion and CREATE function).

The implicit assignment operation (:=) copies the accessor value given by the right hand side expression into the accessor denoted by the left hand side.

The implicit comparison operation (=) returns TRUE if both accessor values are equal (i.e. denote the same object).

All the Barrier declared operations except CREATE and DESTROY raise NON_EXISTENT_BARRIER if the value used as an actual parameter for the accessor B is NULL_ACCESSOR.

The function CREATE "creates" a barrier. A call to CREATE has the following effects:

a. the internal semantics of this creation are specific to each implementation

b. the initial value of the counter is given by the CAPACITY parameter

c. the accessor value that denotes the created barrier is returned.

The NAME parameter semantics are implementation defined (similar to the file creation FORM parameter in LRM 14.2.1.). Its use is never mandatory and the default value is the empty string. In some cases, invalid use of NAME may raise INVALID_BARRIER_NAME.

The function CAPACITY returns the capacity of the barrier.

The procedure WAIT unconditionally "waits" at a barrier. A call to WAIT has the following effects:

```
counter := counter - 1
if counter = 0 then (we are the last one, so open the barrier)    counter := capacity (for the
next time)
    set (pulse)
else (wait with the others)
    wait (pulse)
end if
```

No "timed" or "conditional wait" is proposed, since resuming a task waiting at a barrier while others are still waiting seems unsafe and may make the last ones wait forever.

The procedure DESTROY "deletes" the barrier denoted by the accessor given as actual parameter. A call to DESTROY has the following effects:

a. the internal semantics of this deletion are specific to each implementation

b. the accessor given as actual parameter is set to NULL_ACCESSOR

c. if the set of waiting tasks is not empty all the waiting tasks are resumed and the exception BARRIER_DESTROYED is propagated to them

The function COUNT returns the number of tasks waiting at the barrier. Using this function may be as unsafe as using the COUNT attribute on a task entry.

The function VALUE returns the value of the counter, that is number of waiting tasks needed before opening the barrier.

The effect of invoking this Entry from a Time Critical Section (seeTime Critical Sections CIFO Entry) is implementation-defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

With regard to their semantics, the barrier operations are defined as if executed by an Ada server run at the highest software priority.

Discussion

In some cases the implementation of the BARRIER operations may propagate to the caller an exception that they cannot handle (e.g. the CREATE function may raise STORAGE_ERROR in case of real dynamic allocation). These cases must be documented by the implementation, along with recommended handling.

Note that all the existing copies, if any, of a BARRIER accessor are not updated by the DESTROY operation. The effects of using such copies after destroying an accessor are unpredictable. An implementation may raise NON_EXISTENT_BARRIER when using such dangling accessors. This must be fully documented by each implementation.

In some implementations, the NAME parameter of the CREATE function may be used to denote statically pre-allocated barriers handled in a specific area configured externally (e.g. with a Runtime Executive configuration tool) or for debugging purpose. The semantics of the NAME parameter must be documented by each implementation.

Interactions With Other CIFO Entries

Time Critical Sections: calls to primitives available on Asynchronous Cooperation Objects may be blocking. When such a call blocks in a non-preemptible section, the semantics are exactly the same as for blocking on a rendezvous.

Task Suspension, Two Stage Task Suspension, and Asynchronous Task Suspension: In the same way, when suspending or placing a hold on a blocked task, the semantics are exactly the same as for blocking on a rendezvous.

Abortion Via Task Identifier: Aborting tasks which are waiting in a queue (or set of tasks) associated with an asynchronous cooperation object is similar to aborting tasks which are waiting in a queue associated with a task's entry.

Dynamic Priorities: The queues that are used in the asynchronous cooperation objects will respond to priority changes in exactly the same way as queues associated with Ada task entries.

Priority Inheritance Discipline: In the absence of any Priority Inheritance Discipline in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has the highest software priority of the system. In the presence of Priority Inheritance in an Ada program, the implementation of the current CIFO Entry operates as if its logical virtual agent has a priority ceiling equal to the highest software priority in the system. Consequently, in the presence of Priority Inheritance Discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the service of this CIFO Entry.

Pre-elaboration Of Program Units: The CREATE operations of the Asynchronous Cooperation Objects is allowable for pre-elaboration,if the parameters of the CREATE operation are allowable.

Priority Inheritance Discipline: The logical virtual agents that are hidden in the Asynchronous Cooperation Objects are to be executed at the highest software priority in the system.

Changes From The Previous Release

This is a new Entry.

Other Asynchronous Cooperation Mechanisms

The following CIFO Entries provide mechanisms for asynchronous communication and cooperation between Ada tasks. The mechanisms described in this section do formally define the memory management model used to implement the underlving resource (where appropriate). Some of the Entries in this section do not imply any underlying resource that must be concerned with such issues as dangling references (for instance, Signals).

Two of the Entries in this section, Shared Locks and Mutually Exclusive Access to Shared Data, may at first appear very similar. In fact, Shared Locks could be used to create mutually exclusive access to shared data. However, Shared Locks can be used to ensure mutually exclusive access to objects other than data.

Asynchronous Transfer Of Control

Issue

At the Second International Workshop on Real-Time Ada Issues, three requirements for Asynchronous Transfer of Control were identified. These requirements are restated here:

1) Fault Recovery - requires either stopping a task or altering its flow of control due to an occurrence of a fault. Specific examples include:

A timing fault in which a task fails to complete a time-critical computation on time. Recovery strategies include (1) for the offending task to abandon its ongoing computation and apply a short-cut algorithm and (2) for the offending task to stop executing immediately and restart with fresh data the next time it is scheduled to execute.

A (hardware or software) fault which affects other tasks on which a task is dependent in performing its execution. For example, in a distributed environment, an application may be distributed over multiple processors. When one of these processors has failed, recovery strategies include abandoning the execution of some portion of the application, restarting in a degraded mode, or performing a reconfiguration.

A fault which directly affects a task, yet the task is still able to execute. For example, a task may lose resources other than its memory or processing which prevent it from performing its mission. A recovery strategy might include announcing the task's inability to perform its mission.

2) Mode Changes - may require stopping an application or altering its flow of control. For example, aborting a mbing run to change to a defensive mode.

3) Partial Computations - A task may process an algorithm which gives a first approximation of a result (partial result) and then refines this result (giving progressively better refinements of the result as it is allowed more time to compute). When the result of such a computation is urgently requested, it is then necessary to stop the computing task and make its current partial result available.

Background

At the Third International Workshop on Real-Time Ada Issues, the Asynchronous Transfer of Control Working Group endorsed Tucker Taft's Ada Revision Request (0083) for implementing asynchronous transfers via a modified form of the select statement. The Joint Integrated Avionics Working Group - Common Ada Run Time Working Group (JIAWG/CART) subsequently endorsed this request as also fulfilling their requirements for asynchronous transfers.

The proposed modified form of select can not be implemented within the confines of the 1983 Ada language standard, as it requires a change in syntax. This Entry provides an approximation to the structure and semantics of the Taft proposal. The Taft proposal is reproduced here:

```
select
    select_alternative
{ or
    select_alternative }
and
    sequence_of_statements
end select;
```

The semantics of the above construct are as follows:

Normal processing is performed on select alternatives to determine which ones are open. Selection of one such open alternative takes place immediately if a rendezvous is possible. If no rendezvous is possible but a delay alternative of less than or equal to zero se nds is open, that alternative is executed. Otherwise, the sequence_of_statements following the "and" begins execution. If the sequence_of_statements completes execution, then the select statement as a whole is completed.

If prior to completion of the sequence_of_statements a delay alternative expires or a call is made to an open accept alternative, the sequence_of_statements is abandoned and control is transferred asynchronously to the appropriate open select alternative. This abandonment takes place no later than the next synchronization point, but it is the intent that any ongoing computation (outside of a rendezvous) be preempted, while following current Ada semantics.

For example, the transfer of control will await termination of tasks dependent on master constructs executed by the sequence_of_statements. If the task executing the sequence_of_statements is suspended at an entry call and the corresponding rendezvous has not yet commenced, the call is canceled and the task is removed from the corresponding entry queue. If the task executing the sequence_of_statements is suspended at an entry call and the corresponding rendezvous is in progress, the transfer of control will await completion of the rendezvous. If the task is executing the body of an .ccept statement, the transfer of control takes place and TASKING_ERROR is propagated to the caller.

The following example illustrates the usage of the modified select:

```
task TAFT_EXAMPLE is
    entry BREAK(...);
end TAFT_EXAMPLE;

task body TAFT_EXAMPLE is
begin
    loop
        select
            accept BREAK(...) do
                HANDLE_BREAK;
            end BREAK;
            RECOVER_AFTER_BREAK;
        or
            delay TIMEOUT;
            RECOVER_AFTER_TIMEOUT;
        and
            DO_WORK_ALLOWING_BREAK;
        end select;
    end loop;
end TAFT_EXAMPLE;
```

Proposal

This CIFO Entry provides a generic package to approximate the semantics of Taft's modified select statement within the confines of the 1983 language standard. This package can be thought of as creating an asynchronous agent to execute the sequence of statements following the AND in the Taft proposal, and exports a facility to asynchronously abandon that execution. The generic specification is as follows:

```
generic
    with procedure AGENT_ACTIONS;
    -- AGENT_ACTIONS encapsulates the sequence
    -- of statements following the AND in the
    -- modified select.
    with procedure AGENT_DONE;
    -- AGENT_DONE is intended to be an entry
    -- to be called to signal completion of
    -- AGENT_ACTIONS.
package ASYNCH_AGENT is
    procedure DESTROY_AGENT;
    -- DESTROY_AGENT causes the asynchronous
    -- abandonment of AGENT_ACTIONS.
end ASYNCH_AGENT;
```

The semantics of the body for Asynch_Agent are described by the following: (Note that the following is for illustrative purposes only. The actual implementation may be able to utilize a simplified method of concurrency in implementing the body of the package.)

```
package body ASYNCH_AGENT is

    task AGENT;

    task body AGENT is
    begin
        AGENT_ACTIONS;
        AGENT_DONE;
    end AGENT;

    procedure DESTROY_AGENT is
    begin
        abort AGENT;
    end DESTROY_AGENT;

end ASYNCH_AGENT;
```

Conceptually the agent task must execute at the priority of the instantiator of the generic package. The instantiator should be permitted to execute immediately following the completion of the agent task's activation, thereby permitting any queued entry call to be serviced (and the agent task aborted) prior to initiating the procedure call to Agent_Actions.

The intent of this CIFO Entry is to permit an implementation to exploit knowledge of the limited tasking features used by the agent task and utilize a simplified (and hence more efficient) method of concurrency in implementing the body of the package. The ideal case would be to have the compiler recognize the intended use of this generic package, and eliminate the separate thread of control entirely - as in the Taft solution.

Discussion

The following example illustrates the intended use of the generic package:

```
task TAFT_EXAMPLE_83 is
    entry BREAK(...);
    entry ASYNCH_DONE; -- new entry
end TAFT_EXAMPLE_83;

task body TAFT_EXAMPLE_83 is
begin
    loop
        declare
            procedure DO_WORK_ALLOWING_BREAK is
                ...
            end DO_WORK_ALLOWING_BREAK;
            package ASYNCH is
                new ASYNCH_AGENT (DO_WORK_ALLOWING_BREAK, ASYNCH_DONE);
        begin
            select
                accept BREAK(...) do
                    ASYNCH.DESTROY_AGENT;
                    HANDLE_BREAK;
                end BREAK;
                RECOVER_AFTER_BREAK;
            or
                delay TIMEOUT;
                ASYNCH.DESTROY_AGENT;
                RECOVER_AFTER_TIMEOUT;
            or
                accept ASYNCH_DONE; -- new entry
            end select;
        end;
    end loop;
end TAFT_EXAMPLE_83;
```

In the general case, the details of the transformation from Taft's syntax to the above usage are as follows. First, a new parameterless entry is declared for the task. Next, the select statement is surrounded by a block. In the declarative part of this block are two declarations: a procedure that encapsulates the sequence of statements in the original select statement, and an instantiation of Asynch_Agent. The instantiation uses the new procedure as the first parameter, and the new entry as the second parameter. Then several changes are made to the select statement itself. The "AND" and the following sequence of statements are changed to an "OR" followed by an accept of the new entry. All of the delay alternatives and accept alternatives in the original select statement are modified to add a call to Destroy_Agent in the new instance of Asynch_Agent; this call is the first thing to happen after the delay expires in a delay alternative, and the first thing to happen in the rendezvous in an accept alternative.

This generic package differs from the Taft proposal in the following ways:

The above formulation does not permit an accept statement within the procedure Agent_Actions. The Taft proposal may permit such an accept in the corresponding sequence of statements.

The above formulation recursively aborts any tasks dependent on Agent_Actions. The Taft proposal may permit tasks dependent on the corresponding sequence of statements to execute to their natural termination.

The above formulation may only occur where an accept statement is permitted. In the absence of an accept alternative, the Taft proposal may be permitted elsewhere.

If Asynch_Agent is instantiated with two procedures instead of a procedure and an entry, the effect is to asynchronously execute the two procedures, back-to-back.

Interactions with Other CIFO Entries

Dynamic Priorities: If a dynamic priority is specified for the master task, then a dynamic priority should also be specified for the agent task.

Interrupt Management: Due to the asynchronous nature of this CIFO Entry, care should be used when enabling or disabling interrupts in the AGENT_ACTIONS procedure.

Changes From The Previous Release

This is a new CIFO Entry.

Mutually Exclusive Access to Shared Data

Issue

In many applications the overhead of protecting shared data using a task is too great; furthermore it is not possible to use pragma SHARED for all data types. For portability, there should be a standard mechanism for mutually exclusive access to shared data.

Proposal

A generic package is proposed which when instantiated will allow mutually exclusive access to a shared data item with an appropriate queuing discipline.

```
with QUEUING_DISCIPLINE;
generic
    type ITEM is private;
    INITIAL_VALUE : ITEM;
    QUEUING : QUEUING_DISCIPLINE.DISCIPLINE;
package SHARED_DATA_TEMPLATE is

    type SHARED_DATA is limited private;

    procedure WRITE( TO_OBJECT : in out SHARED_DATA; NEW_VALUE : in ITEM );

    function VALUE_OF( OBJECT : in SHARED_DATA ) return ITEM;
private
    type SHARED_DATA is <implementation-defined>;
end SHARED_DATA_TEMPLATE;
```

After instantiating the generic with the required queuing discipline, a shared data object is created by declaring an object of type SHARED_DATA. The object will be automatically initialized to INITIAL_VALUE. The object can be given a new value by using the WRITE procedure. It can be read by using the VALUE_OF function. VALUE_OF and WRITE operations are mutually exclusive, successive read operations may occur in parallel but successive write operations are mutually exclusive. When tasks are blocked they are queued according to the selected discipline.

Discussion

Where tasks wish to communicate via accessing shared data items, it is important that those items are protected against uncontrolled concurrent access. In Ada this would be achieved by encapsulating the data in a package (possibly using private types) with a hidden manager task, and by providing visible procedures and functions to manipulate the data item. These subprograms in conjunction with the manager task can ensure that read and write operations do not interfere with each other. Without the proper optimizations or pragmas in the compiler, however, these packages would be extremely slow, compared to what is required. It is also beneficial to standardize the interface for such packages in order to facilitate portability. This feature might be used as shown below:

```
with QUEUING_DISCIPLINE;
with SHARED_DATA_TEMPLATE;
package MY_SHARED_DATA is

    type MY_TYPE is <application-defined>;

    package MY_SHARED_OBJECT is
        new SHARED_DATA_TEMPLATE
                (   ITEM => MY_TYPE,
                    INITIAL_VALUE => <application-defined>,
                    QUEUING => QUEUING_DISCIPLINE.SPINNING );

    use MY_SHARED_OBJECT;

    - One could declare the operations here (renaming the
    - packaged operations) in order for the operations to
    - be derivable.

    MY_OBJECT : MY_SHARED_OBJECT.SHARED_DATA;

end MY_SHARED_DATA;
```

Interactions With Other CIFO Entries

This Entry uses Queuing Discipline. The implementation of this package may use other CIFO Entries, such as Shared Locks. There is also an interaction with the Asynchronous Task Suspension Entry. If a task that is reading or writing the shared data becomes suspended then logically it is still reading or writing the data. If a task is queued waiting to read or write the data and it becomes suspended, then when its turn comes to access the data it becomes the logical reader or writer. This effectively blocks every task behind it in the queue.

Note that if a task is aborted while writing the shared data, then normal Ada semantics apply, that is the write completes because it is performed during a rendezvous.

If a time slice expires while writing then the write does not complete until the writing task is rescheduled. Clearly no other tasks can access the shared data during the intervening period.

If dynamic priorities or priority inheritance is available in a CIFO implementation then queues maintained by this Entry may need to be reordered when task priorities change.

If this Entry is implemented using Ada tasks and synchronization discipline is invoked with a conflicting queuing requirement, then the queuing discipline specified by the mutually exclusive access to shared data Entry is overruled.

If the subprograms defined by this Entry are called from within a Time Critical Section and this results in the calling task being blocked, then the processor is rescheduled. When the blocked task is eventually resumed, it once again runs in a Time Critical Section.

Changes From The Previous Release

In the previous release this Entry was called Generic Asynchronous Communication. Its name has been changed so that it more closely reflects the intent of the package. Some rewording has also occurred in the Issue and Discussion sections. The queuing of requests for any one service used to be first-in first-out. Now other queuing disciplines can be defined using the Queuing Discipline CIFO Entry. The function INITIALIZED has been deleted as the result could not be relied upon.

Shared Locks

Issue

Some applications cannot tolerate the overhead of Ada tasking and rendezvous for simple resource control, even when streamlined tasking paradigms such as "trivial entries" are provided.

Some applications require ordering of resource control requests by priority, rather than the FIFO ordering required by the Ada LRM for task entry queues.

Some applications require that certain resources be allocated conditionally, i.e., only if they are immediately available.

Some applications require that certain tasks ("readers") be granted concurrent shared access to a resource, whereas other tasks ("writers") be granted exclusive access to the resource.

Hardware and/or operating systems usually provide instructions (such as "compare and swap", "test and set", or "fetch and add") that can be used to build fast conditional locking mechanisms with appropriate queuing disciplines and levels of resource sharing. A common Ada interface to such capabilities should be provided to enhance portability.

Proposal

A generic package is proposed to provide combinations of the desired resource access (exclusive only versus shared and exclusive) plus the queuing discipline (if any) to be used when the lock is unavailable:

```
with QUEUING_DISCIPLINE;
generic
    QUEUING : in QUEUING_DISCIPLINE.DISCIPLINE;
    -- type of queuing to be used when the lock is unavailable
    ALLOW_SHARED_ACCESS : in BOOLEAN;
    -- true -> shared lock, false -> exclusive lock
package GENERIC_LOCK is

    type LOCK_STATUS_TYPE is ( NOT_LOCKED, SHARED, EXCLUSIVE );

    subtype LOCK_ACCESS_SUBTYPE is
        LOCK_STATUS_TYPE range SHARED .. EXCLUSIVE;

    type LOCK_TYPE is limited private;

    LOCK_ALREADY_OWNED_ERROR : exception;
        -- from LOCK or ATTEMPT_LOCK

    LOCK_NOT_OWNED_ERROR : exception;
        -- from UNLOCK

    LOCK_NOT_SHARABLE_ERROR : exception;
        -- from LOCK or ATTEMPT_LOCK

    LOCK_NOT_INITIALIZED_ERROR : exception;
        -- from any subprogram except INITIALIZE_LOCK
        procedure INITIALIZE_LOCK( LOCK : in out LOCK_TYPE );

        procedure CREATE( LOCK : in out LOCK_TYPE )
            renames INITIALIZE_LOCK;

        procedure FINALIZE_LOCK( LOCK : in out LOCK_TYPE );

        procedure DESTROY( LOCK : in out LOCK_TYPE )
            renames FINALIZE_LOCK;

        procedure LOCK(   LOCK : in out LOCK_TYPE;
                          FOR_USE : in LOCK_ACCESS_SUBTYPE := EXCLUSIVE );

        procedure UNLOCK( LOCK : in out LOCK_TYPE );

        procedure ATTEMPT_LOCK(    LOCK : in out LOCK_TYPE;
                                   FOR_USE : in LOCK_ACCESS_SUBTYPE := EXCLUSIVE;
                                   OBTAINED : out BOOLEAN );

        procedure ATTEMPT_UNLOCK( LOCK : in out LOCK_TYPE );

        procedure ATTEMPT_UNLOCK(    LOCK : in out LOCK_TYPE;
                                     RELEASED : out BOOLEAN );

        procedure BREAK_LOCK( LOCK : in out LOCK_TYPE );

        function GLOBAL_LOCK_STATUS( LOCK : in LOCK_TYPE )
            return LOCK_STATUS_TYPE;

        function LOCAL_LOCK_STATUS( LOCK : in LOCK_TYPE )
            return LOCK_STATUS_TYPE;
    private
        type LOCK_TYPE is <implementation defined>;
    end GENERIC_LOCK;
```

The semantics of this package are as follows:

1. For an INITIALIZE_LOCK request, which must be the first operation performed on a lock, implementation-defined initialization occurs.

2. For a FINALIZE_LOCK request:

   a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

   b. Otherwise, implementation-defined finalization occurs. Once a FINALIZE_LOCK has been done, the lock may not be used again without another INITIALIZE_LOCK.

3. For a LOCK request:

   a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

   b. If the calling task requests shared access but the lock allows only exclusive access, then an exception

LOCK_NOT_SHARABLE_ERROR is raised.

c. Otherwise, if the lock is already owned by the calling task, then an exception LOCK_ALREADY_OWNED_ERROR is raised.

d. Otherwise, if the lock is not currently owned by another task, then the calling task becomes the/an owner of the lock and remains eligible for execution.

e. Otherwise, if the lock is already owned by another task and the lock allows exclusive access only, then the calling task is queued in the appropriate order (based on queuing discipline) and waits its turn at the lock.

When an UNLOCK is done and this calling task is logically first in the queue (based on queuing discipline), this calling task will become the owner of the lock and will again become eligible for execution.

f. Otherwise, the lock is already owned by another task and the lock allows both shared and exclusive access:

(1) If any of the following conditions exist:

(a) either the current owner of the lock or the calling task asked for EXCLUSIVE access,

(b) there is a higher priority task waiting in the queue (priority queuing only), or

(c) there is a task waiting in the queue (FIFO queuing only), then the calling task is queued in the appropriate order (based on queuing discipline) and waits its turn at the lock. When an UNLOCK is done and this calling task is logically first in the queue (based on queuing discipline), the calling task will become the owner of the lock and will again become eligible for execution.

(2) Otherwise, the calling task immediately becomes an owner of the lock and remains eligible for execution.

4. For an UNLOCK request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. If the lock is not currently owned by the calling task, then an exception LOCK_NOT_OWNED_ERROR is raised.

c. If the lock is currently owned by the calling task, then the lock is released and the calling task remains eligible for execution. In addition,

(1) If the lock allows exclusive access only, and if the queue is not empty, then the task that is logically first in the queue (based on queuing discipline) is removed from the queue, becomes the owner of the lock, and becomes eligible for execution.

(2) If the lock allows both shared and exclusive access, and if the calling task was the sole owner and the queue is not empty, then tasks are removed from the queue, become owners of the lock, and become eligible for execution, as follows:

(a) If the task that is logically first in the queue (based on queuing discipline) asked for EXCLUSIVE access, then it becomes the sole owner of the lock.

(b) If the task that is logically first in the queue (based on queuing discipline) asked for SHARED access, then that task becomes an owner of the lock. In addition, any other task asking for SHARED access and queued logically (based on queuing discipline) before the first task asking for

EXCLUSIVE access becomes an owner of the lock.

Note that these semantics provide for minimal priority inversion if priority queuing has been requested and also maximize the number of concurrent SHARED accesses.

5. For an ATTEMPT_LOCK request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. If the calling task requested shared access but the lock allows only exclusive access, then an exception LOCK_NOT_SHARABLE_ERROR is raised.

c. Otherwise, if the lock is already owned by the calling task, then an exception LOCK_ALREADY_OWNED_ERROR is raised.

d. Otherwise, if the lock is not currently owned by another task, then the calling task becomes the/an owner of the lock, OBTAINED is set to TRUE, and the calling task remains eligible for execution.

e. Otherwise, if the lock is currently owned by another task and the lock allows exclusive access only, then OBTAINED is set to FALSE and the calling task remains eligible for execution.

f. Otherwise, the lock is currently owned by another task and the lock allows both shared and exclusive access:

(1) If both the owner(s) and the calling task asked for SHARED access, and the calling task would be logically first if queued (based on queuing discipline), then the calling task becomes an owner of the lock, OBTAINED is set to TRUE, and the calling task remains eligible for execution.

Note that these semantics provide for bounded priority inversion if priority queuing has been requested, since a task asking for EXCLUSIVE access waits only for higher priority tasks and/or tasks already granted SHARED access.

(2) Otherwise, OBTAINED is set to FALSE and the calling task remains eligible for execution.

6. For an ATTEMPT_UNLOCK request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. If the lock is currently owned by the calling task, then the lock is released, RELEASED is set to TRUE (if requested), and the calling task remains eligible for execution. In addition,

(1) If the lock allows exclusive access only and if the queue is not empty, then the task that is logically first in the queue (based on queuing discipline) is removed from the queue, becomes the owner of the lock, and again becomes eligible for execution.

(2) If the lock allows both shared and exclusive access and if the calling task was the sole owner and the queue is not empty, then tasks are removed from the queue, become owners of the lock, and again become eligible for execution, as described in paragraphs 4c(2)(a) and 4c(2)(b) above.

c. If the lock is not currently owned by the calling task, then RELEASED is set to FALSE (if requested) and the calling task remains eligible for execution.

7. For a BREAK_LOCK request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. Otherwise, the lock is totally released (i.e. no task retains ownership of the lock) and the calling task remains eligible for execution. In addition, if the queue is not empty, tasks are removed from the queue, become owners of the lock, and again become eligible for execution, as described under UNLOCK above (paragraph 4c if lock allows exclusive access only, paragraphs 4d(1) and 4d(2) if lock allows both shared and exclusive access).

8. For a GLOBAL_LOCK_STATUS request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. If the lock is not currently owned by any task, then the function returns NOT_LOCKED.

c. If the lock is currently owned by a task that requested exclusive access, then the function returns EXCLUSIVE.

d. If the lock is currently owned by one or more tasks that requested shared access, then the function returns SHARED.

Note that the value returned by GLOBAL_LOCK_STATUS represents the status of the lock at the instant of the call, but this value may already be stale by the time the caller uses it.

9. For a LOCAL_LOCK_STATUS request:

a. If the lock has not yet been initialized, then an exception LOCK_NOT_INITIALIZED_ERROR is raised.

b. If the lock is not currently owned by the calling task, then the function returns NOT_LOCKED.

c. If the lock is currently owned by the calling task with exclusive access, then the function returns EXCLUSIVE.

d. If the lock is currently owned by the calling task with shared access, then the function returns SHARED.

Implementations of this package must ensure that LOCK_TYPE is defined such that arguments of that type are passed by reference, rather than by value, since there can be only one copy of each lock.Note that if this package is used in conjunction with the Priority Inheritance CIFO Entry, the locks should not violate the priority inheritance rules.

Discussion

The rendezvous is a good synchronous communication mechanism that may be used to implement a lock for resource serialization. However, some applications may not be able to tolerate the overhead associated with creating, maintaining, and performing rendezvous with an Ada task to accomplish this serialization. What is really required is a set of fast locking mechanisms that provide common Ada interfaces to machine-dependent implementations taking full advantage of hardware and/or operating system instructions specifically designed for this purpose.

Furthermore, in contrast to the Ada language requirement for FIFO entry queues, some applications require that all resource requests be honored in strict priority order.

Failure semantics for this package are implementation-defined and must be documented for each implementation. The preferred failure semantics would be to protect users by automatically unlocking any locks owned by a terminating task. However, in the absence of such protection, it becomes the user's responsibility to ensure that all locks are unlocked before terminating.

Likewise, deadlock protection mechanisms (prevention, detection, and/or correction) for this package are implementation-defined and must be documented for each implementation. In the absence of such protection, it becomes the user's responsibility to ensure that all locks are requested in an order appropriate for preventing deadlock.

Note that BREAK_LOCK is an extremely dangerous operation, but it may be needed to recover from certain failure situations. Since owner(s) of the lock are not notified of the fact that they have lost control of the lock, the integrity of any resource being controlled by this lock is jeopardized.

Early drafts of this proposal contained four separate packages, corresponding to the four possible combinations of values of the generic parameters. The current alternative was chosen because it seemed simpler to have one unified interface rather than four subtly different ones. Moreover, use of a generic can allow some decisions (such as queuing discipline) to be deferred to run time.

The simplest use of an exclusive lock is as a fast method of doing pure resource serialization. In this case, the application imports package QUEUING_DISCIPLINE (not shown), instantiates GENERIC_LOCK with the appropriate parameters, declares a lock (of type LOCK_TYPE) that is visible to all the tasks wishing to access the resource, and takes care of the initialization and finalization of the lock (not shown). Each task surrounds its access to the resource with a LOCK and UNLOCK as shown in the following code fragment:

```
-- Instantiate lock for exclusive access and priority queuing
package LOCK_INSTANCE is
    new GENERIC_LOCK( QUEUING => QUEUING_DISCIPLINE.PRIORITY_QUEUING,
                      ALLOW_SHARED_ACCESS => TRUE );
use LOCK_INSTANCE;
-- Lock for controlling access to resource
-- (initialization and finalization not shown in this example)

MY_LOCK : LOCK_TYPE;

task body T1 is -- a task needing access to the resource
begin
    -- Perform operations (if any) not using the resource
    -- Obtain exclusive access to resource controlled by MY_LOCK
    LOCK (MY_LOCK);
    -- Perform operations using the resource
    -- Release exclusive access to resource controlled by MY_LOCK
    UNLOCK (MY_LOCK);
    -- Perform operations (if any) not using the resource
end T1;

task body T2 is -- another task needing access to the resource
begin
    -- Perform operations (if any) not using the resource
    -- Obtain exclusive access to resource controlled by MY_LOCK
    LOCK (MY_LOCK);
    -- Perform operations using the resource
    -- Release exclusive access to resource controlled by MY_LOCK
    UNLOCK (MY_LOCK);
    -- Perform operations (if any) not using the resource
end T2;
```

The simplest use of a shared lock is for fast resource control among tasks needing differing types of access (SHARED for "readers" versus EXCLUSIVE for "writers"). In this case, the application imports package QUEUING_DISCIPLINE (not shown), instantiates GENERIC_LOCK with the appropriate parameters, declares a lock (of type LOCK_TYPE) that is visible to all the tasks wishing to access the resource, and takes care of the initialization and finalization of the lock (not shown). Each task surrounds its access to the resource with a LOCK (SHARED or EXCLUSIVE, as desired) and UNLOCK as shown in the following code fragment:

```
-- Instantiate lock for shared access and priority queuing
package LOCK_INSTANCE is
   new GENERIC_LOCK(QUEUING => QUEUING_DISCIPLINE.PRIORITY_QUEUING;
   ALLOW_SHARED_ACCESS => TRUE);
use LOCK_INSTANCE;

-- Lock for controlling access to resource
-- (initialization and finalization not shown in this example)

MY_LOCK : LOCK_TYPE;

task body T1 is -- a task needing shared access to the resource
begin
   -- Perform operations (if any) not using the resource
   -- Obtain shared access to resource controlled by MY_LOCK
   LOCK (MY_LOCK, SHARED);
   -- Perform operations using the resource
   -- Release shared access to resource controlled by MY_LOCK
   UNLOCK (MY_LOCK);
   -- Perform operations (if any) not using the resource
end T1;

task body T2 is -- another task needing shared access to the resource
begin
   -- Perform operations (if any) not using the resource
   -- Obtain shared access to resource controlled by MY_LOCK
   LOCK (MY_LOCK, SHARED);
   -- Perform operations using the resource
   -- Release shared access to resource controlled by MY_LOCK
   UNLOCK (MY_LOCK);
   -- Perform operations (if any) not using the resource
end T2;

task body T3 is -- a task needing exclusive access to the resource
begin
   -- Perform operations (if any) not using the resource
   -- Obtain exclusive access to resource controlled by MY_LOCK
   LOCK (MY_LOCK, EXCLUSIVE);
   -- Perform operations using the resource
   -- Release exclusive access to resource controlled by MY_LOCK
   UNLOCK (MY_LOCK);
   -- Perform operations (if any) not using the resource
end T3;
```

Another common use of shared locks would be within service packages. In this case, the lock itself and associated subprogram calls would be hidden inside the service package and invisible to the application using the service. More sophisticated uses of shared locks can easily be imagined. For example, conditional locking might be used and different alternatives selected based on whether or not the lock is immediately available.

Interactions With Other CIFO Entries

Queuing Discipline: The subject Entry contains a definition of the possible queuing disciplines that can be used when the lock is unavailable.

Dynamic Priorities: Changing of dynamic priorities will affect the ordering of tasks waiting for locks and may cause priority inversion.

Time Critical Sections: The use of Time Critical Sections to enclose lock operations is dangerous and should be avoided.

Abort by Task Identifier: The abortion of a task holding a lock may not release the lock correctly and tasks waiting for the lock may never be dequeued.

Task Suspension Entries: If a task is suspended while holding a lock, deadlock can occur.

Priority Inheritance Entries: If locks are implemented using Ada tasks and/or rendezvous, then the priority inheritance disciplines and the synchronization disciplines apply to these task entry queues. The locks supplied here must obey the priority inheritance rules if this CIFO Entry is used in conjunction with the Priority Inheritance CIFO Entry.

Changes From The Previous Release

This is a new Entry.

Signals

Issue

Real-time applications need a means to asynchronously signal an entry point or procedure upon the occurrence of a specific condition. Such a notification may be accompanied by the communication of a message associated with the signal.

Asynchronous activations can be implemented in Ada, but at the cost of agent tasks.

Proposal

The interface consists of two generic packages :

```
generic
    type PARAMETER is private;
    with procedure TO_BE_CALLED( PARAM : in PARAMETER );
package SIGNAL_WITH_PARAMETER is
    procedure NON_WAITING( PARAM : in PARAMETER );
end SIGNAL_WITH_PARAMETER;

generic
    with procedure TO_BE_CALLED;
package SIGNAL is
    procedure NON_WAITING;
end SIGNAL;
```

The generic formal type PARAMETER should include all data to be communicated, in a single message type.

The effect of calling the procedure NON_WAITING, which is exported by an instance of the SIGNAL_WITH_PARAMETER generic package, is the same as creating a new thread (agent task) which makes a local copy of the parameter value, and then calls TO_BE_CALLED.

The message PARAM is a copy transmitted to TO_BE_CALLED. After the call to NON_WAITING, the value used as actual parameter may be updated without any effect on the message copy.

The Signals Entry behaves as if an agent task were used to transfer the signal and associated message to the target procedure or task entry, even though the implementation may not actually use an Ada task. The semantics of the agent task are that, upon creation, it has the priority of the invoking (initiating) thread of control, if dynamic priorities are available to the implementer of the SIGNALS capability; otherwise the agent task has the highest software priority in the system. Using the priority of the signaling thread of control is the preferred implementation. The existing Ada rules for priorities between tasks within rendezvous apply to the interactions between the agent task and the invoking thread. On the receiving end, the asynchronous signal has the same semantics as a normal call (to either a procedure or a task entry). Similarly, the effect of all other CIFO Entries, such as Priority Inheritance Discipline, are as if an Ada task were being used to transmit the signal.

The generic package SIGNAL is a simplification of SIGNAL_WITH_PARAMETER which corresponds to the case where no data is sent with the signal. The effect of calling the procedure, which is exported by an instance, is the same as for SIGNAL_WITH_PARAMETER except that no parameters are copied.

A NON_WAITING caller never waits. If the callee has disappeared or is not callable at NON-WAITING calling time, then the call is lost. The asynchronous property does not belong to the callee, it belongs to the caller.

The agent task depends on the execution frame of the scope enclosing the instantiation. Accordingly, the appropriate frame may not be left until the agent task completes, specifically the frame may not be left until any existing asynchronous call is finished. This prevents such a call from possibly accessing data that no longer exists.

Any exception possibly raised in the call to the generic actual parameter associated with the procedure TO_BE_CALLED is not propagated to the caller; instead, the agent task completes. In this sense, the agent task acts like any other task that gets an unhandled exception.

On the other hand, any exception that can be raised in the call to the procedures NON_WAITING must be documented by the implementation, along with its recommended handling. In particular, the effects of the policy used to manage the agent tasks and the parameter buffering must be documented.

The effect of invoking this Entry from a Time Critical Section (see Time Critical Sections CIFO Entry) is implementation defined. Specifically, the effect of invoking an operation which could block the current thread of execution from within a Time Critical Section is implementation-defined.

Discussion

Asynchronous activations can be implemented in Ada, but at the cost of agent tasks. An optimized implementation of these packages could bypass the creation of the agent task and would instead use some more efficient way to achieve the same effect.

The existence of two different packages (with and without a parameter) makes a more efficient implementation possible when the need is just to activate asynchronously without sending any data.

The specifications of these two packages allow for using task entries, as well as procedures, as generic actual parameters for TO_BE_CALLED. In the case of procedures, the possible optimization schemes seem to be different from those for task entries. Therefore, if these generic packages are instantiated with an actual parameter that is a procedure, the effect is implementation-defined.

Interactions With Other CIFO Entries

Task Identifiers: If this package is implemented with Ada tasks, a call to TASK_IDS.CALLER from a signaled entry will return the task id of the agent task, and not the originating task. If the package is implemented without agent tasks, the call will return task id for a null task.

The decision to use Signals, rather than a normal Ada rendezvous, is made by the calling task and is not visible to the called task; if the task being signaled is within a reused package, that task may have been designed to use TASK_IDS.CALLER without considering agent tasks. Thus, it is inadvisable to use Signals to call a task entry unless the task entry was designed to be signaled. Any reusable package that can accept Signal entries should document this. Otherwise, users are cautioned against using Signals to interface with reused packages.

Changes From The Previous Release

This Entry supersedes the "Asynchronous Entry Calls" Entry.

Interrupt Support

This section provides facilities for the control of interrupt- oriented hardware (as opposed to control of preemption) and for the specification of additional information to the runtime system in order to significantly enhance performance of interrupt handling.

The provision of interrupt hardware control is in the form of a package, which exports various operations and types. In contrast, in order to tailor the runtime system, pragmas are used to inform the runtime system that certain design considerations are in force. This tailoring allows the runtime system to make use of otherwise unassumable information, resulting in significant performance enhancements. Specifically, pragmas are defined which specify that a given entry declaration will have no sequence of statements for the corresponding accept body, allowing optimizations in the scheduling of calling tasks (given other information concerning priorities). In the same vein, other pragmas specify that certain reasonable restrictions have been followed in the use of the language when implementing interrupt handler tasks, such that traditional amounts of interrupt latency can be achieved.

Interrupt Management

Issue

The LRM (13.5.1) does not define the manner of managing interrupts on the target, since they are highly implementation dependent. Typical architectures incorporate either a level oriented mechanism, in which interrupts are specified by number, or a named mechanism, in which interrupts are individually named. A common format for controlling and interrogating either individually-named or level-oriented interrupts is thus desirable. Clearly, this sort of utility is very machine dependent, and will no doubt be bound differently for each machine. Furthermore, managing interrupts should not be confused with the issues of preemption. This proposal is concerned with the management of very low-level hardware, whereas preemption is concerned with software scheduling and dispatching (see Time Critical Sections CIFO entry).

Proposal

```
package INTERRUPT_MANAGEMENT is

    type INTERRUPT_ID is <implementation-defined discrete type>;

    type INTERRUPT_LIST is array( INTERRUPT_ID ) of BOOLEAN;

    ENABLE_FAILURE : exception;

    procedure ENABLE( INTERRUPT : in INTERRUPT_ID );

    procedure DISABLE( INTERRUPT : in INTERRUPT_ID );

    function ENABLED return INTERRUPT_LIST;

    procedure MASK( INTERRUPTS : in INTERRUPT_LIST );

    procedure UNMASK( INTERRUPTS : in INTERRUPT_LIST );

    function MASKED return INTERRUPT_LIST;

end INTERRUPT_MANAGEMENT;
```

The actual implementation used for type INTERRUPT_ID must be a discrete type.

Procedure ENABLE will make it possible for the specified interrupt to occur (in contrast to allowing the interrupt arbitration system to pass it on once it actually has occurred, which is accomplished by MASK). The procedure DISABLE is called in the same manner as procedure ENABLE. It will disable the interrupt specified from occurring. Calls to ENABLE which fail will raise ENABLE_FAILURE. Calls to DISABLE which fail have no effects.

However, procedures ENABLE and DISABLE have an effect that is implementation-dependent with respect to their effect on any interrupts other that those specified in the call. For example, the hardware may be such that a call to DISABLE will disable the specified interrupt and all "lower-level" interrupts as well. The implementation is expected to document this kind of effect.

The package defines a type INTERRUPT_LIST, which is an array of Boolean status flags. Each flag indicates the state of a particular interrupt. An individual interrupt status may be accessed via the array index, which is based on the names of the interrupts implemented. (The array is constrained for the sake of efficient implementation. An unconstrained array type supporting aggregates containing just those interrupts of interest was considered, but was believed too costly.)

Function ENABLED returns a value of the type INTERRUPT_LIST. The function will return values of "true" for those interrupts that are allowed to occur. Since the function returns an array indicating the status of all interrupts, a snapshot can be taken, as follows:

```
declare
    SNAPSHOT : INTERRUPT_LIST;
begin
    ...
    SNAPSHOT := ENABLED;
    ...
end;
```

The status of several interrupts, at the time the snapshot was taken, can be assessed by indexing the array object:

```
if SNAPSHOT( CLK ) and SNAPSHOT( TIMER1 ) then ...
```

To access the current state of an individual interrupt, the identifier for that interrupt could be used as an index into the result of the function call, as follows:

```
if ENABLED( CLK ) then ...
```

Procedure MASK takes a list of interrupts, treated in the sense that those interrupts specified as "true" will not be accepted if they occur. Those that are set to "false" will not be affected in any way. Thus, procedure ENABLE is used to control whether those interrupts are possible, in a hardware/device context, and procedure MASK controls whether an interrupt is recognized once it does occur. Calls to MASK and UNMASK interrupts which are not enabled have no effects.

Procedure UNMASK takes a list of interrupts to treat in the logical sense that those interrupts specified to be "true" will be accepted if they occur. Those that are set to "false" will not be affected in any way.

Function MASKED returns an array of type INTERRUPT_LIST. Those elements in the array that are "true" are masked in the sense that procedure MASK has been called for them. Those that are "false" are "unmasked" in the same sense that procedure UNMASK has been called for them. Individual interrupts may be interrogated via indexing the function call, or a logical "mask" (snapshot) may be taken at once. As such, this function may be used in a similar manner as that of function ENABLED.

Discussion

The manipulation of interrupts by the application developer is inherently a complex and dangerous activity. The potential for corrupting the integrity of the runtime system is great. However, it is an activity that is commonly considered an absolute requirement in highly-constrained applications. As such, programmers will get the desired effect by whatever manner available, resorting to assembly language if necessary. Therefore, for the sake of portability and consistency of interface, this package is provided, but the use of this package is to be left to highly experienced personnel.

Use of this package is highly dependent upon proper documentation of the runtime environment requirements regarding interrupts, and a thorough knowledge of the supporting interrupt hardware. Extreme care must be taken to avoid corrupting the runtime environment internals. There may exist interrupts that are critical to the implementation and for which the implementation must maintain strict control. The application may therefore be disallowed from enabling/disabling or masking/unmasking these critical interrupts. These interrupts are obviously implementation-dependent and target-dependent and should be specified by the implementation documentation. Additionally, some interrupts may not be maskable, and should also be noted as such by the implementation.

Interactions With Other CIFO Entries

Depending upon details of implementation, there are potential scheduling interactions with four other CIFO entries that must be carefully considered by the user. They are : Time-Critical Sections, Task Suspend and Resume, Two Stage Task Suspension, Asynchronous Task Suspension, Time-slicing, and Synchronous and Asynchronous Task Scheduling.

Time-Critical Sections : the designer of these sections has an application context which typically calls for careful consideration of timing and of interrupts that may occur within the section. Not all hardware interrupts are maskable. Their effects may change the timing and the context envisioned by the designer of the section. In particular, the interrupts that remained enabled in a non-preemptible section may be expressed as an INTERRUPT_LIST array. An implementation can define the interrupts it expects to be enabled as INTERRUPT_LIST. An application can change this configuration of enabled interrupts by enabling/disabling interrupts prior to entering a nonpreemptible section and disabling/enabling them after exiting the non-preemptible section.

Task Suspend and Resume: The inherent dangers of the suspend and resume features are typically applied for managing very scarce resources such as time and processor cycles. Care must be taken in the interactions with interrupts that defeat the intended resource management and with routines designed to manage a certain class of interrupts via enable/disable and mask/unmask operations. Also if suspend operations are not synchronization points, designer attempts to bracket suspends following a sequence of "disable interrupt, update value, suspend, enable" may result in a deferred update that was not intended by the designer but was permitted by an optimizing compiler.

Time Slicing and Asynchronous Task Scheduling: These scheduling entries are subject to many of the same concerns identified above. In addition there are potential interactions with eight other entries that should be understood by the user concerned with predictability of timing, behavior and effects. These interactions may exist because of implementation dependencies or because of the requirements of the application. They are : Events, Pulses, Fast Interrupt Pragmas, Blackboards, Broadcast, Resource Control, Buffers and Asynchronous Transfer of Control.

Note that pre-elaboration of objects of type INTERRUPT_LIST could be beneficial (see Pre-elaboration CIFO entry).

Changes From The Previous Release

To clarify and improve semantics, the exception ENABLE_FAILURE was introduced in the proposal. In the discussion, three sentences were added. These describe the intended semantics when calls to ENABLE or DISABLE fail, and when calls to MASK and UNMASK fail.

Trivial Entries

Issue

An important special case of Ada rendezvous is the utilization of a rendezvous with an empty accept body for the sole purpose of optimizing the performance of task synchronization.

Proposal

The pragma

```
pragma TRIVIAL_ENTRY( NAME : <entry_simple_name> );
```

may appear within a task specification after an entry declaration. Its argument must be the simple name of an entry. This pragma conveys to the implementation the information that the entry at hand has only "trivial" accept statements. A trivial accept statement is one that does not have a sequence of statements, i.e. a trivial accept statement is of the form:

```
accept <entry_simple_name>;
```

If the information conveyed by the pragma is not correct, the Ada program is erroneous. In this case, the pragma should be ignored and a warning message issued by the compiler.

Discussion

An implementation can take advantage of this pragma in the following ways:

If the accepting task is already blocked in the accept statement at the time of the entry call, then the called task will be changed from the blocked to the ready state. The state of the caller will remain unchanged. Transitions to or from the executing state by either task as a result of this entry call is determined by the scheduling policy and processor configuration (uniprocessor or multiprocessor).

If the accepting task is not blocked in the accept statement at the time of the entry call, the pragma will have no effect.

If a task executes a conditional entry call to a trivial entry of another task whose current priority is lower than that of the caller, then it is guaranteed that the caller will not lose control as a result of the entry call. This property is important for "signalling interrupt tasks" as discussed in the proposal on fast interrupt pragmas.

Interactions With Other CIFO Entries

Callers queued at a trivial entry are queued according to the queuing discipline in effect as specified by the Queuing Discipline entry. If the Queuing Discipline entry is not implemented, calls are queued as specified in the LRM.

It is intended that fast interrupt entries should be able to issue conditional entry calls to trivial entries (see the entry on Fast Interrupt Pragmas).

The presence of a pragma Trivial_Entry does not preclude asynchronous entry calls to the named entry, nor does it preclude use of the entry in a selective wait statement. Such entries may be used as actual subprogram parameters for instantiations of packages SIGNAL and SIGNAL_WITH_PARAMETER (refer to SIGNALS CIFO entry).

As a consequence of the Ada language rules for address clauses for subprograms, an entry (and hence a trivial entry) can not be called via the mechanism defined in the CIFO entry on UNCHECKED_SUBPROGRAM_INVOCATION.

Changes From The Previous Release

None.

Fast Interrupt Pragmas

Issue

Transfer of control to an accept statement for an interrupt entry is slower than transfer of control to a traditional interrupt routine (whose address has been loaded into a hardware interrupt vector). This is because in the case of the Ada interrupt rendezvous, various tasking data structures in the runtime environment have to be read and updated. The difference in speed is likely to prohibit the use of Ada for many real-time applications.

Proposal

This proposal consists of two pragmas, INTERRUPT_TASK and MEDIUM_FAST_INTERRUPT_ENTRY, and an enumeration type INTERRUPT_TASK_KIND. The enumeration type declaration appears in package SYSTEM and is defined as follows:

```
type INTERRUPT_TASK_KIND is ( SIMPLE, SIGNALLING );
```

Note that a given implementation may support additional enumeration values.

The first pragma

```
pragma INTERRUPT_TASK( KIND : INTERRUPT_TASK_KIND )
```

is only allowed within a task specification for a single task (c.f. LRM 9.1 (2)). It must precede any entry declaration. This pragma conveys to the implementation the information that the task at hand satisfies a set of restrictions, and that the Ada program to which this task belongs satisfies certain restrictions as well. If any of the restrictions are violated, the Ada program is erroneous.

The parameter KIND indicates the exact set of restrictions that are satisfied. The possible values include SIMPLE and SIGNALLING. A particular implementation may support additional values for KIND.

The following restrictions apply to all interrupt tasks:

1. An interrupt task has exactly one entry, and that entry is an interrupt entry. An interrupt task body has no declarative part. Its sequence of statements consists of a single loop statement that has no iteration scheme. The sequence of statements of this loop statement, in turn consists of a single accept statement for the entry of the task. (Note that there is no restriction, however, on how many statements there may be in the "sequence of statements" of this accept statement.)

2. The Ada program to which this task belongs does not contain any entry calls to the interrupt task.

3. The sequence of statements inside the accept statement does not contain or invoke code that changes the interrupt state of the hardware. In particular, for architectures where interrupts are disabled upon occurrence of an interrupt, interrupts are not re-enabled during the interrupt rendezvous (associated with the accept statement).

4. The execution of the interrupt rendezvous does not, at any one time, occupy more than a given, implementation-dependent amount of stack space. If the interrupt rendezvous invokes subprograms, the stack space occupied by those subprograms is included in the count. If the interrupt rendezvous causes subsequent interrupts, the stack space required for the handling of these interrupts is also included in the count.

5. The sequence of statements inside the accept statement, as well as all the code that is invoked by this sequence of statements, is restricted to what might intuitively be described as "straightforward code". The definition of "straightforward code" varies with the KIND of the interrupt task.

"Straightforward code" for a SIMPLE interrupt task does not contain:

i. Delay statements, entry calls, accept statements, declarations of tasks, allocators on access types for task types, program asynchronous I/O operations, abort statements

ii. References to task attributes of the interrupt task at hand, or references to the attribute COUNT of the interrupt entry at hand.

iii. Any checks for the exception STORAGE_ERROR in the generated code.

iv. Any code that leads to the raising of Ada exceptions. (This means, e.g. that it is the responsibility of the Ada programmer to ensure that no CONSTRAINT_ERROR exception will be raised during an interrupt rendezvous.)v. References to types or objects that are declared in scopes that enclose the interrupt task, except for types and objects that are declared in library packages.

vi. Calls to the services of other CIFO entries.

The definition of "straightforward code" for SIGNALLING interrupt tasks is identical to "straightforward code" for simple interrupt tasks, with one exception: the sequence of statements inside the accept statement, as well as the code that is invoked by this sequence of statements, may include a conditional entry call to a "trivial entry" in a different task. (See the interface proposal "Trivial Entries".)

The second pragma

pragma MEDIUM_FAST_INTERRUPT_ENTRY(NAME : <entry_simple_name>);

may appear within a task specification after an entry declaration. Its argument must be the simple name of an entry, and that entry must be an interrupt entry. This pragma conveys to the implementation the information that the entry at hand satisfies all those restrictions that are satisfied by the entry in a signalling interrupt task, except for the restriction concerning references to non-local types and objects. The accept statement of a medium fast interrupt entry, as well as the code invoked by it, are permitted to reference all those types and objects that they may reference according to the scope and visibility rules of the Ada language.

In addition to these semantics for each pragma, the following apply. The runtime environment does not change the interrupt masking state of the hardware between the occurrence of the interrupt and the end of the interrupt rendezvous. In particular, it does not re-enable interrupts before the end of the interrupt rendezvous.

At the end of the interrupt rendezvous, the runtime environment resets the interrupt masking state of the hardware to what it was before the interrupt occurred. In other words, the runtime environment cancels the changes that were effected by the hardware when the interrupt occurred. The time needed to transfer control to the accept statement in an interrupt task or in a signalling interrupt task is of the same order of magnitude as the time required by the state saving portion of a traditional interrupt routine. The exact time is implementation-dependent.

The time needed to transfer control to the accept statement for a medium fast interrupt entry lies between the time required in the case of an interrupt task, and the time needed for an ordinary interrupt entry. The exact time is implementation-dependent.

Discussion

An interrupt task can be implemented in the following way:

First, augment the sequence of statements of the accept statement, by adding code to save and restore the state of the interrupted computation before and after the interrupt rendezvous.

Then, load the starting address of the thus augmented sequence of statements into the hardware interrupt vector for the interrupt at hand. Depending on the hardware architecture, this operation may be part of a more comprehensive interrupt setup operation, through which an interrupt mask is specified along with the address. It is assumed that whatever needs to be loaded along with the address is known, either by convention in the runtime environment, or because it has been specified by the user in some implementation-dependent fashion.

The fact that the runtime environment does not re-enable any interrupts before the end of the interrupt rendezvous, along with the restrictions 1 through 3 above, makes this implementation equivalent to the loop in the source text of the interrupt task.

In the runtime model for Ada, the execution of all Ada code is viewed as taking place under the control of tasks. If a program segment is not invoked by an explicitly declared task, it is viewed as executing under the control of "some environment task" (c.f. LRM 10.1(8)). The runtime environment therefore maintains information on:

1. which task is currently executing

2. which tasks are currently eligible to execute

Whenever control passes from one task to another, this information has to be updated. The restrictions for interrupt tasks are such that this information can be allowed to be out of date for the duration of an interrupt rendezvous. In the case of a simple interrupt task, the information will automatically be correct again after the interrupt rendezvous is over. In the case of a signalling interrupt task, the information may have to be updated, but the update can be postponed until the end of the interrupt rendezvous.

The significance of the individual restrictions is this:

1 through 3: Make it possible to load the address of the (augmented) sequence of statements for the interrupt rendezvous directly into the interrupt vector.

4: The runtime environment allocates (statically) stack space for interrupt rendezvous. This space must not be exceeded.

5: "Straight-forward" code restrictions vary with the type of interrupt task (as defined by the enumeration type INTERRUPT_TASK), and are discussed in the following:

i: The implementation of the abort statement may or may not involve task switches. Since it is unlikely that there will be a strong requirement for an abort statement in an interrupt rendezvous, it is prohibited. All other constructs listed here will lead, at least potentially, to task switches. This must be prohibited if the lists used by the task switching mechanism are out of date.

The weaker version of restriction (i) that is postulated for signalling interrupt tasks is still sufficient to avoid task switches during the interrupt rendezvous: the updating of the appropriate lists can be postponed until after the interrupt rendezvous.

ii: These operations make use of a variable indicating which task is currently executing.

iii: Depending on the particular implementation, these operations may make use of a variable indicating which task is currently executing. This is most likely the case if there is no hardware mechanism for detecting stack overflow, or if the initialization of that mechanism is too slow.

iv: An Ada exception can be re-raised from within an exception handler, and this can be done without specifying the exception again. It is therefore likely that implementations will save the exception in a task-specific location when the exception is raised for the first time. The pointer to this location may be out of date, and moreover, an implementation may not provide such a location for an interrupt task.

v: Non-local variables will be accessed through task specific pointers. For interrupt tasks, these pointers may either not be implemented or not be up to date. Note that this restriction does not exist for medium fast interrupt rendezvous. The updating of these pointers is the reason why a medium fast interrupt rendezvous is slower than both simple and signalling interrupt tasks. (Note that accessing of non-local variables from an interrupt handler may not be feasible in all architectures.)

With the possible exception of (iv), the above restrictions are met naturally in many real-time applications. The proposed pragmas can therefore be viewed as a mechanism informing the runtime environments of the application. Once the runtime environment has knowledge of these circumstances, it can take advantage of them and provide a more efficient implementation.

Below are three examples of the use of fast interrupt pragmas. The situation assumed for all these examples is the following: whenever the temperature measured by a hardware probe exceeds a given threshold value, the probe generates an interrupt request to the CPU. As soon as the CPU recognizes this interrupt, it transfers control to a routine that computes new settings for a certain group of valves, and issues commands to adjust the valves accordingly.

Note that the three examples behave differently with respect to "losing" interrupts:

In Example 1, it is guaranteed that no occurrence of the interrupt in question will be lost.

In Example 2, no occurrence of the interrupt will be lost, but the conditional entry call to the trivial entry SECONDARY.SYNCH may be lost. This will happen if the conditional entry is issued at a time where the task SECONDARY is not suspended in the matching accept statement.

In Example 3, an occurrence of the interrupt will be serviced if the task named PARTLY_INTERRUPT_DRIVEN is suspended in the matching accept statement when the interrupt occurs. Otherwise, the interrupt will be lost (assuming the interrupt is implemented as a conditional entry call).

Example 1: "Simple" Interrupt Tasks

```
task INTERRUPT_DRIVEN is
    pragma INTERRUPT_TASK( KIND => SIMPLE );
    entry HOT;
    for HOT use at HOT_ADDRESS;
end INTERRUPT_DRIVEN;


task body INTERRUPT_DRIVEN is
begin
    loop
        accept HOT do
            -- compute new settings for valves;
            -- issue commands to change valve settings;
        end HOT;
    end loop;
end INTERRUPT_DRIVEN;
```

Example 2: "Signalling" Interrupt Tasks

```
task INTERRUPT_DRIVEN is
    pragma INTERRUPT_TASK( KIND => SIGNALLING );
    entry HOT;
    for HOT use at HOT_ADDRESS;
end INTERRUPT_DRIVEN;

task SECONDARY is
    entry SYNCH;
    pragma TRIVIAL_ENTRY( NAME => SYNCH );
end SECONDARY;

task body INTERRUPT_DRIVEN is
begin
    loop
        accept HOT do
            compute new settings for valves;
                -- issue commands to change valve settings;
                -- schedule the second task:
            select
                SECONDARY.SYNCH;
            else
                null;
            end select;
        end HOT;
    end loop;
end INTERRUPT_DRIVEN;


task body SECONDARY is
begin
    loop
        accept SYNCH;
        -- perform elaborate and time consuming operations which are
        -- needed because interrupt occurred;
    end loop;
end SECONDARY;
```

Example 3: "Medium Fast Interrupt Entries"

```
task PARTLY_INTERRUPT_DRIVEN is
    entry HOT;
    for HOT use at HOT_ADDRESS;
    pragma MEDIUM_FAST_INTERRUPT_ENTRY( NAME => HOT );
end INTERRUPT_DRIVEN;

task body PARTLY_INTERRUPT_DRIVEN is

    type SNAPSHOT is
        record
            - various data on the temperature, the time,
            - and other values at the time
            - when the interrupt occurred
        end record;

    HISTORY : array(1 .. n) of SNAPSHOT; - used as a cyclic buffer
                                         - to reco, d the last n SNAPSHOTS

    SUMMARY :     - variable of some suitab;y chosen type that
                  - summarizes the recent history

    COEFFICIENT :- variable to contain a coefficient that is
                  - used in the computation of the new valve
                  - settings

begin
    - obtain a value for COEFFICIENT;
    loop
        accept HOT do
            -record the current S, JAPSHOT into the current element of HISTORY;
            -based on the current SNAPSHOT, COEFFICIENT, and SUMMARY,
            -compute new settings for valves;
            -issue commands to change valve settings;
        end HOT;
        - update SUMMARY;
    end loop;
end PARTLY_INTERRUPT_DRIVEN;
```

Interactions With Other CIFO Entries

Trivial Entries: Interrupt tasks of type SIGNALLING may include a condit... entry call to a "trivial entry" in a different task.

Interrupt Management: By disabling an interrupt execution of a Fast Interrupt Rendezvous may be disabled. Apart from this, there is no interaction between Fast Interrupts and Interrupt Management.

Time-Critical Sections: Depending on the implementation, preemption control may use interrupt management services to disable the interrupt, thereby disabling the execution of the Fast Interrupt rendezvous.

Synchronization Discipline, Priority Inheritance: These disciplines have no impact on Fast Interrupts, since in this context an occurrence of an interrupt has the semantics of a conditional entry call.

Time Slicing, Asynchronous Transfer of Control, Synchronous/Asynchronous Task Scheduling, Dynamic Priorities: A task enclosing a Medium Fast Interrupt Entry may be affected by these services in the usual way. Such a task may also call these services in the usual way, but not from within the interrupt rendezvous. These services affect the enclosing task, but not the Fast Interrupt Rendezvous.

No CIFO entries, except those allowed for Trivial Entries, may be used from a Fast Interrupt Rendezvous. Note that this implies that TASK_IDS.SELF can not be called from within an interrupt task. Therefore, an interrupt task can not be the object of CIFO services which use task ids.

Changes From The Previous Release

In the description of "straightforward code", the restriction was added that calls to other CIFO entries are not permitted.

Compiler Directives

This section provides facilities which furnish the compiler with additional information about the application in order to increase performance, reduce executable image size, support shared data, and support additional functionality (such as referencing a static object via an access value).

In some cases pragmas alone are sufficient, and in others pragmas and subprograms are supplied. When combined, the pragmas supply information to the runtime system so that potential implementation problems can be avoided with respect to the subprograms' interaction with the rest of the runtime system.

This section thus defines facilities for:

a) requesting the elaboration of given items prior to execution,

b) designating statically-allocated objects via access variables,

c) indicating when tasks should or should not be implemented as a thread of control,

d) calling subprograms via their address, and

e) specifying data objects which are shared between tasks (beyond the functionality of the predefined pragma SHARED).

Pre-elaboration of Program Units

Issue

Real-time applications frequently require that programs (and tasks) be able to start up more quickly than if all the work of elaboration is done at execution time. A well-defined class of constructs, defined similarly to static expressions but larger in scope, is needed such that a programmer can rely on the elaboration of these constructs not taking any execution time. We call constructs whose elaboration does not take any execution time "pre-elaborated".

Runtime elaboration of constant data structures and a priori known tasks is not consistent with many embedded systems' power-up and restart requirements.

Proposal

```
pragma PRE_ELABORATE( [ <identifier_list> ] );
```

The pragma shall be allowed in the positions allowed for a declaration. It will request the compilation system (i.e., compiler/linker) to pre-elaborate the indicated list of data structures and program units. If the optional identifier list is omitted, all possible entities will be pre-elaborated, in the program unit where the pragma appears, and a list of those entities that cannot be pre-elaborated will be produced by the compilation system.

Pre-elaborated objects should be ROM-able.

An implementation supporting this feature shall include the following constant declaration in package SYSTEM:

```
SUPPORTS_PREELABORATION : constant BOOLEAN := TRUE;
```

Support of this feature requires that at least the following "allowable" constructs be pre-elaborated:

A static expression is allowable.

A scalar type with a static constraint is allowable.

A subtype is allowable if its base type is allowable, the constraint is static, and elaboration of the subtype raises no exceptions.

An array type is allowable if the component and index subtypes are allowable.

A record type is allowable if every component is of an allowable subtype.

An access type is allowable if the designated subtype is allowable.

An allocator is allowable if its subtype and initial value (if any) are allowable.

An aggregate is allowable if its subtype, component values, and any expressions used as array index choices are allowable.

An object declaration is allowable if its subtype is allowable and not constrained, and either it is a variable declaration and has no initial value, or it is a constant declaration and the value is specified by an allowable expression.

A type or subtype declaration is allowable if the type or subtype is allowable.

A subprogram declaration or body is always allowable (this does not include generic units and generic instantiations).

A generic instantiation is allowable if the generic unit, its body, and all generic parameters are allowable, and the instantiation does not raise any exceptions.

A package is allowable if all its component declarations are allowable and its body has no sequence of statements.

A task, and a task object which is declared using a task type, is allowable if all the component declarations of its body are allowable and it does not occur within another task or subprogram body.

A generic declaration is allowable if all expressions and types in its formal part are allowable, and its body is a subprogram or allowable package body.

Note in particular that expressions containing allocators are allowable in constant declarations.

Discussion

This is a proposal for a kind of optimization --- one that is sufficiently critical for real-time applications that some standard support seems necessary. In an application that is driven by timing constraints it is necessary to know a priori which constructs are "safe" with regard to not imposing any significant run-time costs. It is typical of many real-time applications that the entire application or certain tasks within it must be able to be started (and restarted) instantly, more or less. This implies that constructs which impose run-time elaboration costs be avoided. Also, constant data bases are currently loaded into read-only memory, so that runtime initialization of these entities is impossible. It is therefore important that application programmers and real-time Ada compilers agree upon a set of constraints, so that if a programmer respects these constraints and requests pre-elaboration, the compiler can be relied upon not to generate any significant amount of run-time elaboration code.

While it is clear that each compiler must set a limit on how far to go in compile-time elaboration, in the absence of a standard this limit will vary widely. It would then be impractical to develop any common style of real-time programming and difficult to re-use real-time software components across compilers. Any application concerned about start-up time would need to define its own set of usage restrictions to fit a particular compiler, and might even require a custom compiler.

The present Ada standard sets a precedent by defining a class of "static" expressions, which a compiler is supposed to be able to evaluate at compile-time. This class is clearly more narrowly defined than necessary, but it is sufficient to be useful. The class of allowable constructs for pre-elaboration defined here is a generalization of this idea to more complex constructs including certain simple forms of aggregate objects, subtypes, procedures, packages, and tasks.

By defining a standard class of constructs that can be elaborated we hope to give real-time programmers a framework within which they know it is "safe" to operate, given they use a compiler that supports this feature. By requiring compilers that do not support this feature to reject programs that request it, programmers would know that programs that compile are "safe".

We feel the class of allowable constructs defined here is sufficiently narrow to insure the constructs can be elaborated at compile time, but sufficiently large to permit programming a useful range of real-time applications. The requirements above are a compromise between ease of implementation and generality. They could be somewhat relaxed, at a price. For example, initial values could be allowed in variable declarations, but then restarting a program quickly might be difficult. Similarly, expressions involving selection of components and slices of allowable array and record constants might be allowed, but the benefit was felt to be outweighed by the extra cost. Even though certain kinds of nested tasks could conceivably be pre-elaborated, we have ruled them out for similar reasons. We intend to have ruled out all constructs whose elaboration might require the execution of a statement. In contrast, allocators are permitted in expressions defining the values of constants, even though supporting this may not be easy. This is permitted because access types are considered a necessity for real-time programs.

Interactions With Other CIFO Entries

Asynchronous Cooperation Mechanism object "accessors", as implemented by the reference memory model, are allowable if the parameters (if any) are allowable (see Asynchronous Cooperation Mechanisms CIFO entry).

Mutually Exclusive Access to Shared Data is allowable if the data which is shared is allowable.

Shared Locks objects are allowable.

Note that various other CIFO entries could benefit from preelaboration, such as interrupt handler tasks (see Fast Interrupt Pragmas CIFO entry).

Changes From The Previous Release

The definition of the pre-elaboration concept has been reworded.

The definition of the allowability concept is new.

A non-limited list of allowable entities (entities to be preelaborated in an application) is provided.

Access Values That Designate Static Objects

Issue

In many embedded systems, it is necessary to manipulate references to static objects.

An object may need to be static because it is located in read-only memory, or it is accessed by an I/O processor directly (DMA), or because it is a pre-elaborated constant, or for other reasons.

It may be necessary to refer to such an object indirectly. For example, a reference to the object may appear as an actual parameter of a procedure or as the value of a function. Also, groups of static objects are sometimes logically viewed as arrays, while they do not have all the physical properties of arrays, such as contiguity in memory. In these cases, the logical view can be implemented as an array of references.

Proposal

This proposal consists of a generic function, a pragma, and a Boolean constant as defined in the following:

```
generic
    type OBJECT is limited private;
    type ACCESS_TYPE is access OBJECT;
function MAKE_ACCESS_VALUE( STATIC : OBJECT ) return ACCESS_TYPE;
```

Assume a type OT and a corresponding access type AOT, and assume the generic instantiation:

```
function ACCESS_TO is new MAKE_ACCESS_VALUE( OT, AOT );
```

If OBJ is of type OT, the value ACCESS_TO(OBJ) will be an access value that designates OBJ in the same way in which the access value returned by an allocator designates the corresponding dynamically allocated object.

If an access value obtained in this way is used after the designated object has ceased to exist, the effect is undefined.

Because this feature may require special compiler support, an implementation shall also include an implementation-defined pragma:

```
pragma MAY_MAKE_ACCESS_VALUE( <type_mark> );
```

This pragma is to be specified for every type or subtype for which a programmer intends to instantiate MAKE_ACCESS_VALUE. This pragma must appear in the same declarative part as the named type, and must appear before any forcing occurrence (see LRM 13.1) of the named type.

The implementation shall also supply a boolean constant in package SYSTEM named:

```
MAKE_ACCESS_SUPPORTED : constant BOOLEAN := <boolean value>;
```

which indicates whether or not the feature is supported.

An implementation of this feature must assume that the access method for the type at hand is known at compile time. If this or other conditions make a safe implementation for a particular type impossible, the compiler shall generate a warning. If executable code is generated at all, the exception PROGRAM_ERROR shall be raised as soon as possible. This may be at the point of an instantiation of MAKE_ACCESS_VALUE, or when such an instantiation is called.

If UNCHECKED_DEALLOCATION is instantiated for a (sub)type for which pragma MAY_MAKE_ACCESS_VALUE has been specified, the compiler shall generate a warning. If UNCHECKED_DEALLOCATION is instantiated and applied to a static object of such a (sub)type, the effect is undefined and may in fact result in global corruption of the memory management mechanism.

Discussion

Intuitively, the effect of an instance of function MAKE_ACCESS_VALUE is equivalent to first obtaining the address of the object in question through the ADDRESS attribute, and then converting that address through a suitable instance of UNCHECKED_CONVERSION into the desired access value. This method will indeed work in many cases. However, it relies on internal implementation details, both of the representation of the object in question, and of the implementation of access values.

The proposed feature removes these implicit dependencies.

Interactions With Other CIFO Entries

There are no interactions between this entry and other CIFO entries. The proposed generic function does not constitute a change to the Ada language. The last sentence of 3.8 (8) of the LRM will no longer be correct, but 3.8 (8) is a "Note" and as such not part of the Ada language definition.

Changes From The Previous Release

This is a new CIFO entry.

Passive Task Pragma

Issue

Applications programmers require greater control over the manner in which tasks are implemented by an Ada compilation system.

Many tasks can be implemented as either active or passive entities. Active tasks require separate threads of control that can be independently scheduled by the run-time system. Passive tasks are those that do not require threads of control. For passive tasks, entry calls can be transformed into procedure calls, avoiding the cost of task switching. The memory required for the implementation of passive tasks is negligible, roughly equivalent to simple objects such as semaphores.

Numerous Ada abstractions are represented using intermediate tasks such as server tasks or agent tasks which are inherently passive. If the implementation cannot detect and optimize such tasks, their cost can be prohibitive for many real-time applications.

Many of the operations defined by CIFO entries are applicable only to active tasks (e.g. suspend/resume, set_priority, etc). If an Ada compilation system implements a task that is intended to be manipulated by CIFO operations as a passive entity, that task can not be correctly used by calls to CIFO entries.

The application must have control over which tasks are passive and which are active.

Proposal

An implementation-defined pragma:

```
        pragma THREAD_OF_CONTROL( DESIRED : BOOLEAN );
```

The pragma THREAD_OF_CONTROL is allowed within a task specification both for a single task and a task type. This pragma conveys to the implementation whether the application intends for the immediately enclosing task to be implemented as an active or passive task.

Conforming implementations must issue a warning message if the intent of the THREAD_OF_CONTROL pragma is not (or can not) be obeyed. If the task contains a pragma THREAD_OF_CONTROL(FALSE) and the implementation can not implement the corresponding task body as a passive task, a warning message must be generated. Similarly, if the application contains a pragma THREAD_OF_CONTROL(TRUE) and the implementation does not implement the corresponding task body as an active entity, a warning message must be generated.

At a minimum, an implementation that supports passive tasks should allow passive tasks of either of the following two forms:

1) Simple accept statements

```
task body T is
    <restricted_declarative_part>
begin
    loop -- optional loop
        accept E1(...) do
            ...                              - optional accept body
        end E1;
        .
        .
        accept En(...) do
            ...                              - optional accept body
        end;
    end loop;
end T;
```

2) Selective wait statement

```
task body T is
    <restricted_declarative_part>
begin
    loop                                     - optional loop
        select
            when <condition> =>              - optional guard
                accept E1(...) do
                    ...                      - optional accept body
                end E1;
        or
            .
            .
        or  when <condition> =>              - optional guard
                accept En(...) do
                    ...                      - optional accept body
                end En;
        or  when <condition> =>
                terminate;                   - optional
        end select;
    end loop;
end T;
```

<restricted_declarative_part> consists of any declaration except declarations of dependent tasks, inner packages, objects of dynamic size or access types. Additionally, the default initialization for any declaration is restricted from calling user-defined functions.

In both forms the following restrictions apply :

1) In the selective wait, no delay alternative or else part is allowed.

2) Nested accept statements are not allowed.

3) Entry families are not supported.

4) In the task body there is one and only one accept statement for each entry declared in the task specification.

5) The specification of storage size for a task activation is not allowed for passive task and is ignored by the implementation if present.

6) Passive tasks may not have dependent tasks.

7) No exception handler is allowed for the task body.

8) The guard expressions (<condition>) must not contain calls to user-defined functions.

9) The only statement allowed outside of select and accept bodies is an unconditional loop.

An implementation is free to provide additional forms of passive tasks, or to eliminate some of the above restrictions. An implementation must clearly define the circumstances under which a task is eligible for implementation as a passive entity.

Discussion

A passive task can be implemented in the following way :

The task body elaboration is done by the activator task on behalf of the passive task. The restriction that passive task declarative parts must not contain function calls prevents incorrect deadlock due to an entry call from one passive task to another during its activation (since the activations are actually serialized). The activator elaborates data declared in the passive task body, the size of this data area is static since dynamic-sized objects are not allowed. However objects of dynamic size could be supported without using the heap as long as a static storage_size clause is used. Indeed, a static-value storage size clause could be used to specify statically the task body data size. Then the data would be "allocated" in this area during elaboration (including dynamic-size). STORAGE_ERROR would be raised if the storage size clause value was too small during elaboration. This requires the use of a storage size clause when declaring dynamic-sized objects and introduces an additional complexity for the programmer. For this reason dynamic-sized objects are not allowed.

Each accept body is executed by the entry caller on behalf of the passive task as if the accept body was a simple procedure. (It is interesting to note that the two forms of passive tasks can be regarded in the same manner by the implementation. Indeed the first form can be transformed by the implementation into an infinite loop on a selective wait composed of guarded accept alternatives plus a guarded terminate alternative so that only one alternative is open at a given time.) The selective wait can be viewed as a set of procedures protected by a lock. When a task calls an entry of a passive task it attempts to get the lock (Test_And_Set operation). If the lock is not available the task is suspended and inserted in the corresponding entry queue of the passive task. If the lock is free the caller task executes the accept body statements and then re-evaluates the guards of the selective wait if any. If no task is suspended on any open alternative the caller task releases the lock; otherwise it executes the implementation-defined algorithm for selecting the next caller task, which is awakened and inherits possession of the lock.

Termination of the passive task is performed either by its master at the point of master completion and termination of all other dependents (via selection of the terminate alternative in the select statement), or by an entry caller getting back an unhandled exception terminating the task, or by an explicit abort of the passive task. Clean-up actions taking place upon task termination can be time-bounded since no dependent tasks, no access types, and no objects of dynamic size are allowed in the declarative part of the task body.

An exception handler is not allowed for the task body since all the statements excluding the outer loop statement are encapsulated into accept statements. Consequently any exception occurring outside an accept statement terminates the passive task. This applies to exceptions raised during task body elaboration, exceptions raised but not handled within accept statements, and exceptions raised during guard evaluation. When the exception is raised during elaboration, the activator must raise TASKING_ERROR after terminating the passive task. If the exception is raised during guard evaluation, the passive task is terminated with no impact on the caller. Any unhandled exception occurring in an accept procedure terminates the passive task and is propagated to the caller.

Justification of the programming restrictions :

Allowing a delay alternative or else part in select statements, or nested accept statements within accept procedures, would require a thread of control for the task which is contrary to the passive task concept.

The prohibition of entry families, and the requirement for a one-to-one correspondence between an entry and an accept statement, allow a static association between each entry call and its accept subprogram which enables the subprogram-like call optimization.

Though the storage size clause might be a solution for supporting declarations of dynamic-sized objects, it is traditionally used to control stack size, which is meaningless for a task which does not have a stack. For this reason this feature is not allowed.

Statements outside accept procedures except for the outer infinite loop are not allowed because they would have to be executed by the caller task, and consequently the semantics of the task would differ from those when no pragma PASSIVE is applied.It is important to note that passive tasks can be assigned priority via the pragma PRIORITY. They may have local data and entries mapped onto interrupts. There are no restrictions on the content of the accept bodies. They may be null procedures (trivial entries), and any statements are allowed, in particular entry calls and abort statements. The following examples show the utility of passive tasks.

Example 1

The following example, derived from that given in LRM 9.12, illustrates the need to allow guards and local data in the passive task model.

```
task BUFFER is
    entry READ (C : out CHARACTER);
    entry WRITE (C : in CHARACTER);
    pragma THREAD_OF_CONTROL(FALSE);
end BUFFER;

task body BUFFER is
    POOL_SIZE : constant INTEGER := 100;
    POOL   : array(1 .. POOL_SIZE) of CHARACTER;
    COUNT: INTEGER range 0 .. POOL_SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
begin
    loop
        select
            when COUNT < POOL_SIZE =>
                accept WRITE(C : in CHARACTER) do
                    POOL(IN_INDEX) := C;
                    IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
                    COUNT:= COUNT + 1;
                end WRITE;
        or when COUNT > 0 =>
            accept READ(C : out CHARACTER) do
                C := POOL(OUT_INDEX);
                OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
                COUNT:= COUNT - 1;
            end READ;
        or
            terminate;
        end select;
    end loop;
end BUFFER;
```

Task switches to and from the task BUFFER are completely avoided, so the producer and the consumer just alternate execution. If they have the same priority, they may alternate only when the buffer becomes full or empty, instead of at each character, thus saving even more task switches. Another important benefit is that no thread of control is given to the buffer task, saving the need to allocate a stack.

Example 2

The following example shows how the passive task optimization can solve efficiently the asynchronous communication issue in Ada. This example re-uses the agent task model given in the Rationale to illustrate how it is possible to "decouple" a client task from a server task. This allows the customer to proceed with further actions after requesting a service and before receiving the results, and allows the server to return the results of the service without fear of the customer not being ready to receive them, by passing them instead to an agent which is guaranteed to be waiting. The pragma PASSIVE allows elimination of threads of control both for the server task and the agent task. This is possible because passive task entries can be mapped onto interrupts and entry calls are supported within accept bodies.

```
task type MESSENGER is
    entry DEPOSIT(X : in ITEM);
    entry COLLECT(X : out ITEM);
    pragma THREAD_OF_CONTROL(FALSE);
end MESSENGER;

task body MESSENGER is
    STORE : ITEM;
begin
    accept DEPOSIT(X : in ITEM) do
        STORE := X;
    end DEPOSIT;
    accept COLLECT(X : out ITEM) do
        X := STORE;
    end COLLECT;
end MESSENGER;


type AGENT is access MESSENGER;


task SERVER is
    entry REPAIR(X : ITEM; A : AGENT);
    -- The following entry is mapped onto an interrupt raised by an external device
    entry INTERRUPT;
    for INTERRUPT use at EXTERNAL_DEVICE_ADDRESS;
    pragma THREAD_OF_CONTROL(FALSE);
end SERVER;

task body SERVER is
    JOB       : ITEM;
    THE_AGENT : AGENT;
begin
    loop
        accept REPAIR(X : ITEM; A : AGENT) do
            JOB       := X;
            THE_AGENT := A;
            -- Initiate the repair in item JOB :
            -- INTERRUPT is called when JOB is complete.
        end REPAIR;
        accept INTERRUPT do
            -- the following rendezvous is never blocking.
            -- this is a feature of agent tasks
            THE_AGENT.DEPOSIT(JOB);
        end INTERRUPT;
    end loop;
end SERVER;
```

```
task CUSTOMER;

task body CUSTOMER is
    MY_ITEM : ITEM;
    MY_AGENT : AGENT := new MESSENGER;
begin
    ...
    SERVER.REPAIR(MY_ITEM, MY_AGENT);
    ...
    MY_AGENT.COLLECT(MY_ITEM);
    ...
end CUSTOMER;
```

Example 3

The following example shows how the passive task optimization can be applied to some instances of asynchronous transfer of control This example deals with the time-out issue in Ada. In particular it shows how an asynchronous transfer mechanism can be achieved with passive task model supporting features such as abort statement and exception propagation in accept procedures. The example assumes that the effect aborting a passive takes place immediately.

```
--
-- watch-dog via passive task optimization
--
task type SUB_CONTRACTOR is
    entry DO_THE_JOB;
    pragma THREAD_OF_CONTROL(FALSE);
end SUB_CONTRACTOR;

type AGENT is access SUB_CONTRACTOR;

task body SUB_CONTRACTOR is
begin
    accept DO_THE_JOB do
        -- This calculation should finish in 5.0 seconds
        -- if not, it is assumed to diverge
        HORRIBLY_COMPLICATED_RECURSIVE_FUNCTION(X,Y);
    end DO_THE_JOB;
end SUB_CONTRACTOR;


task WATCH_DOG is
    entry ARM(X : in DURATION; Y : in AGENT);
    entry DISARM;
    -- the following entry is mapped onto a timer interrupt
    entry ELAPSED;
    for ELAPSED use at TIMER_INTERRUPT_ADDRESS;
    pragma THREAD_OF_CONTROL(FALSE);
end WATCH_DOG;
```

```
task body WATCH_DOG is
    THE_AGENT : AGENT;
    PENDING_TIME_OUT : BOOLEAN := FALSE;
begin
    loop
        select
            accept ARM(X : in DURATION; Y : in AGENT) do
                THE_AGENT := Y;
                PENDING_TIME_OUT := TRUE;
                -- Perform necessary actions to cause ELAPSED
                -- to be triggered X seconds later
                ...
            end ARM;
        or when PENDING_TIME_OUT =>
            accept DISARM do
                PENDING_TIME_OUT := FALSE;
                -- Clear the pending time-out
                ...
            end DISARM;
        or when PENDING_TIME_OUT =>
            accept ELAPSED do
                PENDING_TIME_OUT := FALSE;
                -- The time-out has elapsed : cancel the calculation
                -- It is assumed that the effect of the abort takes place
                -- immediately.
                abort THE_AGENT;
            end ELAPSED;
        end select;
    end loop;
end WATCH_DOG;


-- Perform time-limited calculation
-- This calculation should finish in 5.0 seconds
-- If not, it is assumed to diverge
declare
    MY_AGENT : AGENT := new SUB_CONTRACTOR;
begin
    -- The effect of the passive tasks optimization is the
    -- three following entry calls are executed as procedure
    -- calls without blocking
    WATCH_DOG.ARM(5.0, MY_AGENT);
    MY_AGENT.DO_THE_JOB;
    WATCH_DOG.DISARM;
exception
    when TASKING_ERROR =>
        PUT_LINE("Calculation does not converge");
end;
```

Interactions With Other CIFO Entries

Trivial_Entries

The pragma TRIVIAL_ENTRY does not conflict with the pragma PASSIVE. The pragma THREAD_OF_CONTROL(FALSE) and TRIVIAL_ENTRY have the same effect in term of rendezvous optimization, but in addition the pragma THREAD_OF_CONTROL(FALSE) conveys to the implementation that the task does not need a thread of control.

Task_Ids

It is not possible to obtain the TASK_IDENTIFIER of a passive task. Calls to TASK_IDS.SELF, TASK_IDS.CALLER, TASK_IDS.MASTER that would indicate a passive task raise TASK_IDS.PASSIVE_TASK_ERROR. As a consequence, any CIFO entry that operates on tasks via their TASK_IDENTIFIER can not operate on a passive task. This includes:

Abortion by task identifier

Task Suspend and Resume

Two Stage Task Suspension

Asynchronous Task Suspension

Time Slicing

Synchronous and Asynchronous Task Scheduling.

Task Suspend and Resume and Two Stage Task Suspension

The effect of calling Suspend_Self from within an accept statement of a passive task is undefined, and should be avoided. Asynchronous Task Suspension

The effect of calling HOLD_TASK specifying a task that is currently executing an entry call to a passive task has the effect of holding the passive task. Subsequent calls to other entries of the passive task can not be accepted until the held caller is released by a call to RELEASE_TASK, and has completed the rendezvous. Note that this situation can lead to excessive blocking times for other clients of the passive task, and should be avoided.

Time Slicing

If the time quanta expires for a task that is currently executing an entry call to a passive task, the effect is to suspend the passive task. Subsequent calls to other entries of the passive task can not be accepted until the original caller is rescheduled and completes the rendezvous. Note that this situation can lead to excessive blocking times for other clients of the passive task, and should be avoided.

Trivial Entries

Entries of passive tasks can be designated as trivial entries. A call to such an entry must perform any subsequent guard evaluation and reevaluate the status of callers queued at other entries.

Changes From The Previous Release

This is a new Entry.

Unchecked Subprogram Invocation

Issue

Ada provides the capability to take the address of any program unit (LRM 13.5,13.7.2) but the language does not guarantee that all implementations provide the corresponding capability to invoke that unit by its address.

Proposal

An implementation-defined pragma shall be supported, of the following form:

```
pragma INVOKE_BY_ADDRESS( <subprogram_name> );
```

An Ada language implementation supporting this feature shall permit the use of a subprogram address clause as a means for calling a subprogram via an address that is determined at run time. Support for this feature shall be indicated by inclusion of the following constant declaration in package SYSTEM:

```
SUPPORTS_INVOCATION_BY_ADDRESS : constant BOOLEAN := TRUE;
```

Implementations supporting this feature shall recognize a pragma INVOKE_BY_ADDRESS, whose semantics are defined below. This pragma takes a subprogram name as its only argument.

The code sketch below illustrates how this feature could be used to call subprogram SOME_PROCEDURE with arguments Z and W.

```
CALL_TABLE : array ( 1 .. N ) of SYSTEM.ADDRESS;
...
CALL_TABLE(I) := SOME_PROCEDURE'ADDRESS;
...
declare
   procedure P( X : in INTEGER; Y : out INTEGER );
   pragma INTERFACE(SOME_LANGUAGE,P);
   pragma INVOKE_BY_ADDRESS(P);
   for P use at CALL_TABLE(I);
begin
   ...
   P( Z, W );
   ...
end;
```

According to the Ada language standard, the effect of an address clause is to specify the address of the body of a subprogram. This proposal defines the effect of an address clause more precisely:

1) The address specified in an address clause is the address to be used in calling the corresponding subprogram.

2) The address provided by the attribute 'ADDRESS can be used in calling the corresponding subprogram, as required by (1) above.

3) For a subprogram to be invoked by address (as specified by an INTERFACE pragma and an INVOKE_BY_ADDRESS pragma), an arbitrary address expression shall be permitted in the address clause. In particular, the expression shall not be required to be static.

4) For a subprogram to be invoked by address, the subprogram declaration is presumed to be an alias for another subprogram, which will be specified at run time. The compiler and linker shall check that there is no body corresponding to the subprogram mentioned in the two pragmas (as for any other subprogram for which an INTERFACE pragma is specified). For instance, the address provided at run time (the "actual subprogram address") may be that of a subprogram whose declaration and body appear elsewhere in the same Ada program. Another example would be the address of a built-in test subprogram supplied by a hardware vendor; such a routine might be provided in ROM and have no source code available.

For an INVOKE_BY_ADDRESS pragma to take effect, all of the following must occur:

1) The pragma must occur at the place of a declarative item, and must apply to a subprogram declared by an earlier declarative item of the same declarative part or package specification.

2) An INTERFACE pragma must be specified for the same subprogram. (Otherwise, Ada would require that a body be given for the subprogram.)

3) An address clause must be given for the same subprogram. (Otherwise, the meaning is unclear.)

4) The address clause must occur after both the INTERFACE pragma and the INVOKE_BY_ADDRESS pragma (to simplify compiler implementations).

The following run-time restrictions apply to subprograms that are invoked by address. The execution of a program is erroneous if any of these restrictions are violated.

1) If the actual subprogram address refers to an Ada subprogram, the subprogram must not be declared within a block statement or the body of a subprogram or task. The purpose of this restriction is to avoid problems associated with nested execution environments. Specifically, this eliminates the need for specifying environment information (e.g., a "static link") along with the subprogram address, and eliminates the possibility of calling a subprogram from outside its scope.

2) Similarly, this feature is not required to be supported for actual subprogram addresses that refer to generic instantiations, or to subprograms declared within packages that are generic instantiations. The reason for this restriction is that code-sharing schemes for such a subprogram may require additional environment information for calling it, so that the address is not sufficient.

@indent1 = 3) Similarly, this feature is not required to be supported for actual subprogram addresses corresponding to Ada attributes, enumeration literals, or entries. While Ada sometimes treats these entities as if they were subprograms at the user level, they are treated in diverse ways by the lower levels of Ada implementations.

4) If the actual subprogram address refers to an Ada subprogram, then that subprogram and the subprogram to be invoked by address must have the same parameter and result profile. Furthermore, the subtypes of corresponding formal parameters must be the same.

5) Similarly, the result subtype of any function results must be the same.

6) The language specified for the subprogram to be invoked by address must specify the proper calling conventions for the actual subprogram address. Similar restrictions to those in 1 through 5 above may be required for interfaced subprograms written in other languages. The exact nature of the restrictions will depend on the language.

This feature shall be supported for interfaced subprograms in at least the following languages:

a) Ada -- meaning subprograms following the calling conventions of the same Ada compiler, whether compiled from Ada source code or written in assembly language.

b) ASSEMBLER -- meaning subprograms following the basic calling conventions established by the processor instruction set.

Discussion

The Ada language definition leaves considerable freedom to language implementors with regard to the specific effect of a subprogram address clause. It is therefore an option for the implementation to permit an address clause to be used, in combination with an INTERFACE pragma, to call a subprogram whose address is not specified until run time.

The effect of this proposal is to introduce a limited form of dynamic binding into Ada. However, this proposal uses the term "invocation by address" rather than "dynamic binding" because the latter term can be much broader (as in LISP, for example).

This proposal seems to meet the functional requirements of the JIAWG Ada 9X revision request entitled "execution of a program unit by its address," without changing the Ada language; similarly, it responds to the JIAWG Common Ada Runtime Requirement for "support for separately compiled ROM-based procedures," which was derived from that revision request. The proposal's main weakness is that there is no way of automatically checking that the various run-time restrictions are actually observed. Another weakness is that some external routines may not be callable with this proposal, due to the wide variety of possible calling conventions; for example, it may not be possible to tell the Ada compiler in which registers a ROM routine expects to find its parameters.

Early versions of this proposal omitted the pragma INVOKE_BY_ADDRESS; instead, the effect of this pragma could have been assumed whenever the address in a subprogram's address clause was nonstatic. This could have caused problems, however, for compilers for which the type SYSTEM.ADDRESS is not a scalar type, and hence has no static values (LRM 4.9(1)). Moreover, this could lead to confusion in cases like the following:

```
-- Assume System.Address is an integer type ...
procedure SYSTEM_ROUTINE;
pragma INTERFACE( C, SYSTEM_ROUTINE );
-- for SYSTEM_ROUTINE'ADDRESS use SYSTEM.ADDRESS(3000); -- not static
-- for SYSTEM_ROUTINE'ADDRESS use SYSTEM.ADDRESS'(3000); -- static
```

If the interpretation of address clauses were to depend on staticness, then the two similar-looking address clauses above could have very different meanings. The first would mean that invocation by address is used, so the linker need not find a body named SYSTEM_ROUTINE; instead, whatever is at address 3000 is used. The meaning of the second would be system dependent, but one possible interpretation is that the linker is to find a C procedure named SYSTEM_ROUTINE ... and place it at address 3000. To avoid confusion in cases like this, the pragma INVOKE_BY_ADDRESS was introduced.

Some ARTEWG members have complained that invocation by address is dangerous. The entire idea seems somewhat contrary to the "spirit" of Ada; the language design team went to great lengths to make sure that all binding of Ada routines is done at compile time or link time. One reason for this was to support Ada's strong typing. As can be seen, this proposal circumvents Ada typing: many run-time requirements, such as parameter matching, are not checked. Accordingly, it would be wise for application programmers to avoid using this proposal except in the rare cases where it is truly necessary.

Interactions With Other CIFO Entries

Trivial Entries, Signals and Passive Tasks: It is a consequence of the semantics of task entries that address clauses cannot be applied to entries of tasks that have been renamed as procedures; thus no interaction is possible.

All entries: It is not recommended that users attempt to replace individual CIFO subprograms using this facility.

Changes From The Previous Release

This is a new Entry.

Data Synchronization Pragma

Issue

The use of CIFO entries for Ada task synchronization will lead to errors if the user assumes that these entries are synchronization points. Some CIFO users, unaware of the risk involved, will naturally want to use these entries to implement protocols for sharing data among Ada tasks. If the Ada compiler being used has optimizations suppressed or does not perform a high degree of optimization, this approach to data sharing may work, accidentally. Unfortunately, with an Ada compiler that does perform a high degree of optimization, this approach to data sharing has a high probability of failure. As highly optimizing compilers become more common, this problem will occur more often.

The problem arises because an Ada compiler will not recognize the invocation of CIFO entries as a synchronization point as it would a task entry call. To solve the problem, the synchronization points and the data shared must be identified for the compiler. If this is done, the CIFO task synchronization entries can be used to implement data sharing. The pragma SHARED is limited to scalars and access types, a small subset of the variables types that need to be shared. The proposed pragma SHARED_DATA eliminates these restrictions.

Proposal

The proposed pragma designates a variable that is permitted to be a scalar, access type, or an aggregate type as shared and requires every read or update of the variable or its components within the scope of its declaration to be treated as a synchronization point by the compiler.

```
pragma SHARED_DATA ( <variable_name> );
```

The shared variable is designated by <variable_name>, a variable of any type. The <variable_name> may be restricted to a <variable_simple_name> as it is in pragma SHARED or extended to record components, array elements, and slices at the discretion of the compiler implementor. The pragma must occur only in the declarative part that contains the declaration of the variable.

The effect of pragma SHARED_DATA is to force the compiler to allocate storage for the shared variable, to read a value from storage when the variable or any of its components is referenced, and to write a value to storage when the variable or any of its components is updated. With shared cached memory, the cache entry should be written to memory or flushed as appropriate. The effect of pragma SHARED_DATA is to maintain data coherency.

Discussion

The use of the SHARED_DATA pragma together with task synchronization CIFO entries or any other task synchronization support provided by the vendor will permit the implementation of reliable data sharing between tasks. Since the effect of the pragma is to inhibit optimizations involving the shared variable in the scope of its declaration, a large number of references to and updates of the variable can result in performance problems. Shared variables will not be maintained in registers across multiple references or updates. The effect on performance due to cache manipulations could also be significant. The recommended approach is that if a shared variable is referenced more than once in a critical region that it should be copied to a local variable. For the compiler implementor, pragma SHARED_DATA should be straightforward to implement since it maps into the machinery that most compilers possess for tracking the run-time state.

Interactions With Other CIFO Entries

This pragma ensures that it is possible to use successfully CIFO synchronization entries such as locks to control access to shared data objects.

Changes From The Previous Release

This is a new Entry.

Memory Management

This section of the CIFO contains an Entry to allow applications more control over the management of memory allocation and deallocation.

Dynamic Storage Management

Issue

The freedom which the Ada standard leaves to the language implementer with regard to dynamic storage management makes it difficult for application programmers to program time and memory-constrained applications. Two particular manifestations of this problem are: (1) Ada Runtime Environment routines for storage management may use up significant amounts of processor time at times not predictable by the application programmer; (2) dynamic storage may be exhausted without any warning to the application program, and the STORAGE_ERROR exception may be raised too late or in a section of the program where recovery is not possible.

This proposal is intended to provide low-cost implementations of dynamic storage management for those resource-constrained applications that require dynamic storage allocation and recovery, but where the more general schemes would be too time-consuming.

Proposal

Two packages are proposed: (1) POOLS to define storage pools, and (2) GIVE_AND_TAKE to assign designated user data types to be allocated from these pools:

```
package POOLS is

    type POOL_ID is private;

    type BLOCK_POINTER is <implementation defined>;

    NULL_BLOCK_POINTER : constant BLOCK_POINTER := <implementation defined>;

    type NUMBER_OF_FREE_BLOCKS is range 0 .. <implementation defined>;

    subtype NUMBER_OF_BLOCKS is
        NUMBER_OF_FREE_BLOCKS range 1 .. <implementation defined>;

    type NUMBER_OF_STORAGE_UNITS_PER_BLOCK is range <implementation defined>;

    POOL_STORAGE_ERROR          : exception;
    POOL_BLOCKS_STILL_IN_USE    : exception;
    POOL_ID_INVALID             : exception;
    POOL_BLOCK_UNAVAILABLE      : exception;
    POOL_BLOCK_SIZE_TOO_SMALL : exception;

    function NEW_POOL(  POOL_SIZE : NUMBER_OF_BLOCKS;
                        BLOCK_SIZE : NUMBER_OF_STORAGE_UNITS_PER_BLOCK )
        return POOL_ID;

    function CREATE_POOL( POOL_SIZE : NUMBER_OF_BLOCKS;
                          BLOCK_SIZE : NUMBER_OF_STORAGE_UNITS_PER_BLOCK )
        return POOL_ID renames NEW_POOL;
    procedure RELEASE_POOL( POOL : in POOL_ID );

    procedure CONDITIONAL_DESTROY_POOL( POOL : in POOL_ID )
        renames RELEASE_POOL;

    procedure UNCONDITIONAL_RELEASE_POOL( POOL : in POOL_ID );

    procedure UNCONDITIONAL_DESTROY_POOL( POOL : in POOL_ID )
        renames UNCONDITIONAL_RELEASE_POOL;

    procedure EXTEND_POOL( POOL : in POOL_ID;
                           EXTENSION_SIZE : in NUMBER_OF_BLOCKS );

    function GET_BLOCK( POOL : POOL_ID ) return BLOCK_POINTER;

    procedure FREE_BLOCK( POOL : POOL_ID; BLOCK : in out BLOCK_POINTER );

    function POOL_SIZE( POOL : POOL_ID ) return NUMBER_OF_BLOCKS;

    function BLOCK_SIZE( POOL : POOL_ID ) return
        NUMBER_OF_STORAGE_UNITS_PER_BLOCK;

    function FREE_STORAGE_BLOCKS( POOL : POOL_ID )
        return NUMBER_OF_FREE_BLOCKS;

private
    type POOL_ID is <implementation defined>;
end POOLS;


with POOLS;
generic
    POOL : POOLS.POOL_ID;
    type DESIGNATED_SUBTYPE is limited private;
package GIVE_AND_TAKE is

    - *** WARNING: This access type may NOT be used with "new",
    - *** "UNCHECKED_DEALLOCATION", or "pragma CONTROLLED".
    type POINTER is access DESIGNATED_SUBTYPE;

    function NEW_BLOCK return POINTER;

    procedure RELEASE_BLOCK( BLOCK : in out POINTER );

end GIVE_AND_TAKE;
```

The semantics of these packages are as follows:

1. Package POOLS contains basic storage pool operations. Note that the allowed ranges for the number of free blocks in a pool (NUMBER_OF_FREE_BLOCKS) must include 0.

a. A call to function POOLS.NEW_POOL creates a new storage pool with the specified number of blocks and storage units per block and then returns a pool identifier for the new storage pool. If there is not enough free storage available, exception POOL_STORAGE_ERROR is raised and no storage at all is allocated to the pool.

NOTE: The number of storage units per block specified by the caller does not include the amount of storage that may be needed by the implementation for internal bookkeeping purposes. In addition, a storage pool is not required to occupy one contiguous area of memory.

b. A call to procedure POOLS.UNCONDITIONAL_RELEASE_POOL or procedure POOLS.RELEASE_POOL deallocates all storage associated with the specified storage pool and invalidates its pool identifier. Once a pool is released, any attempt to use its invalidated pool identifier will cause exception POOL_ID_INVALID to be raised.

NOTE: Since pool ids will be checked for validity, this implies that pool ids must be unique for the life of the program.

NOTE: If any blocks in the pool are still in use when RELEASE_POOL is called, exception POOL_BLOCKS_STILL_IN_USE is raised and the pool is not released. However, if any blocks in the pool are still in use when UNCONDITIONAL_RELEASE_POOL is called, no exception is raised and it is the caller's responsibility to ensure that nobody else is dependent upon the storage blocks in the pool being released.

c. A call to procedure POOLS.EXTEND_POOL adds the specified number of blocks to the specified storage pool. If there is not enough free storage available, exception POOL_STORAGE_ERROR is raised and no storage at all is added to the pool. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

d. A call to function POOLS.GET_BLOCK allocates a block from the specified storage pool and returns its pointer. If there are no blocks available, exception POOL_BLOCK_UNAVAILABLE is raised. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

e. A call to procedure POOLS.FREE_BLOCK deallocates the specified block from the specified storage pool and sets the pointer to null. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

f. A call to function POOLS.POOL_SIZE returns the total number of blocks currently in the specified pool. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

g. A call to function POOLS.BLOCK_SIZE returns the actual number of storage units per block in the specified pool. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

NOTE: The block size returned by this function may be larger than the number of storage units per block specified when the pool was created. For example, the implementation may enforce a minimum block size, or may round the block size up to maintain boundary alignment. The implementation must document the approach used.

h. A call to function POOLS.FREE_STORAGE_BLOCKS returns the number of blocks currently available in the specified pool. If the specified pool identifier is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID is raised.

2. Generic package GIVE_AND_TAKE may be instantiated any number of times to associate particular user data types with certain pools. The generic parameter POOL is the pool from which the designated subtype is to be allocated. If the number of storage units per block for the specified pool is not large enough to hold the specified user data, the exception POOL_BLOCK_SIZE_TOO_SMALL in package POOLS is raised. Note that objects of the type GIVE_AND_TAKE.POINTER should never be used with "new", UNCHECKED_DEALLOCATION, and pragma CONTROLLED.

a. A call to function NEW_BLOCK in an instantiation of generic package GIVE_AND_TAKE allocates a block from the appropriate pool and returns an appropriate access type for the user data. If there are no blocks available, exception POOL_BLOCK_UNAVAILABLE in package POOLS is raised. If the pool identifier specified at generic instantiation is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID in package POOLS is raised.

b. A call to procedure RELEASE_BLOCK in an instantiation of generic package GIVE_AND_TAKE deallocates the block associated with the specified access value from the appropriate pool and sets the pointer to null. If the pool identifier specified at generic instantiation is not (currently) associated with a valid storage pool, exception POOL_ID_INVALID in package POOLS is raised.

WARNING: The access type defined by GIVE_AND_TAKE may NOT be used in conjunction with "new", "UNCHECKED_DEALLOCATION", or "pragma CONTROLLED". Otherwise, the integrity of the heap might be compromised.

Any implementation of these packages must guarantee that:

1. all subprograms in these packages execute safely in a multitasking environment

NOTE: This requirement may necessitate the use of storage synchronization mechanisms (e.g. locks) that in turn may result in tasks being blocked. The implementation must document any calls which are blocking.

2. the execution time of allocation of a block from a storage pool is bounded by a (small) constant

3. the execution time of returning of a block to a storage pool is bounded by a (small) constant

The actual execution times are implementation defined and must be documented for each implementation.

Discussion

It should be noted that the solution that is being proposed here provides upper bounds for the time for allocation and deallocation of blocks only for pools of the kind introduced here. No such bounds are guaranteed for "sized collections" or for the "global heap". It is not guaranteed that an operation on a "sized collection" or on the "global heap" does not trigger a long internal storage reorganization in the runtime environment. Moreover, it is also not guaranteed that the runtime environment does not use the dynamic storage administration system for internal purposes, such that storage reorganizations may be triggered by events that have no obvious connection to dynamic storage administration.

Interactions With Other CIFO Entries

If the implementation of these packages can cause blocking to occur, the subprograms should not be invoked in Time Critical Sections, while holding shared locks, or from interrupt handlers. An aborted task holding a pool block may not release that block and the storage associated with that block may be lost.

Changes From The Previous Release

This is a new CIFO Entry.

Rationale for Deleted Entries

The *Catalogue of Interface Features and Options for the Ada Runtime Environment* is intended to be a living document. As a working group we have always encouraged comments from the public. The results of the review process, as explained in the preface, are added incrementally to the catalog.

Special Delays

As a consequence of the review process several criticisms of the Entry "Special Delays" were brought to our attention, summarized as follows:

The characteristics of CPU instructions, especially timing, are very target dependent.

It is unrealistic for a vendor to ensure that this feature will work accurately for a family of processors.

To provide accurate timing, knowledge of external hardware support may be required.

If provided by the vendor, this feature may give the user false expectations in anticipating precise time delays.

This feature is more suitably supplied by the end user having knowledge of the exact hardware characteristics

These comments have convinced Subgroup III (Interfaces Subgroup) to delete the Special Delays Entry.

Transmitting Task Identifiers Between Tasks

The function TRANSLATE made an implementation detail of the type TASK_ID explicit. It had been thought that, in distributed systems, a TASK_ID value on one computer could very well be implemented as the address of a task control block (TCB). This address would not be correct, however, if it were used by another task on a different computer.

After further consideration, it became clear that many other types of data (including access types) would suffer the same difficulties, and that a distributed Ada system should have general solutions, not specific, uncoordinated ones. For example, an implementation could make location completely transparent, such that access variables (for example) would be automatically converted when transmitted to another computer. An implementation could, on the other hand, allow an application to explicitly control location, and provide only the lowest-level communication services. In the latter case, implementation details would need to be provided for all local types, so that application designers could create their own translation operations.

Within this release of the Catalogue, distribution has explicitly been deferred. A task force within ARTEWG is examining this issue in general, among others. Since this entry provided a specific solution which was not part of a more coordinated and consistent long-term approach, it was removed.

Interactions Between CIFO Entries

The CIFO interactions matrix describes required and known interactions between CIFO Entries. Each cell of the matrix indicates how two CIFO Entries interact. Additional information is provided in the interactions subsections for the affected Entries.

The table is a non-symmetric square, with each cell answering the question: does the row CIFO Entry depend upon the definition of or operation of the column Entry?

There are two categories of interactions considered:

1. Referential dependency -- the row Entry uses the column Entry by means of an Ada context clause. In this case, there is a CIFO requirement for the dependency.

2. Meaning dependency -- the row Entry depends for it's effect on the effect of the column Entry. In this case, the sense of the dependencies is related to the nature of the row and column Entries.

The CIFO Entries are classed according to their effect:

Actor CIFO Entries have their effect directly and immediately by being invoked. Broadcasts are good examples.

Environment CIFO Entries have their effect indirectly by changing the Ada or RTS environment so that other modules behave differently. Priority changing is an example. The specific change of priority value has no effect, but the different priority values do cause modules that enforce scheduling decisions to behave differently.

The distinction is important because the scope of action of the environment variable depends upon that to which it is applied. For example, the decision of whether or not the scheduling Entries (actor class) are compatible with the passive task Entry (an environment class) depends upon whether or not the passive task Entry applies to the specific task that the scheduling Entry applies to.

This means that the questions that the matrix answers are slightly different depending upon whether the row and column Entries are actors or environment classes. In particular, if:

Row = actor; Column = actor:

Can the Row Entry be performed while the Column Entry is active? Does the Row Entry have a different meaning if the Column Entry has been or is being used?

Row = actor; Column = environment:

Can the Row Entry be used while the Column Entry is acting on the execution environment applicable to the Row Entry?

Row = environment; Column = actor:

Can the Row Entry change the environment while the Column Entry is acting? Does the Row environment affect the action being performed by the Column Entry?

Row = environment; Column = environment:

Will the Row Entry change the environment on which the Column Entry is applied (i.e. the environments overlap)? Will the Row Entry constrain actors (in the environment affected by the Row Entry) which might reference the environment affected by the Column Entry (i.e. the environments do not overlap but they "communicate" by actors in the environment affected by the Row Entry)?

The interaction matrix is constructed for users of CIFO Entries, not for implementers. It is not intended as a guide for implementers, but rather to constrain or define the effects of using CIFO Entries.

Interactions in the table are classed according to these notations:

"blank entry"	The row Entry does not interact with the column Entry.
"P"	The row Entry pot:ntially interacts with the column.
"I"	The row Entry definitely interacts with the column Entry, but in a manner intended by the designers of the respective Entries.
"X"	the row Entry and the column Entry interact in a competing manner or are incompatible. The actual interaction is either forbidden or discouraged.

If the interaction is noted as "I", then a second letter of "R" indicates that the interaction is because of an Ada context clause and is required.

CIFO Interactions Matrix

		TI	QD	SAT	PI	DP	TCS
TI	Task Identifiers						
QD	Queuing Disciplines				I	I	
SAT	Synchronous and AsynchronousTask Scheduling	IR			I	IR	P
PI	Priority Inheritance		I	P			
DP	Dynamic Priorities	IR	I	I	I		I
TCS	Time Critical Sections			P		I	
ATI	Abortion via Task Identifiers	IR		P			I
TSR	Task Suspension (and Resumption)	IR		X		I	I
TSTS	Two Stage Task Suspension (and Resumption)	IR		X		I	X
ATSR	Asynchronous Task Suspension(and Resumption)	IR		X		I	X
TS	Time Slicing	IR		X	X	X	I
SD	Sychronization Discipline	IR	IR		I	I	
R	Resources		IR	P	I	I	I
E	Events			I	I	I	I
P	Pulses				I	I	I
BUF	Buffers		IR		I	I	I
BLK	Blackboards				I	I	I
BRD	Broadcasts				I	I	I
BAR	Barriers				I	I	I
ATC	Asynchronous Transfer of Control	P	P	P	P	I	P
MEA	Mutually Exclusive Access to Shared Data		IR		I	I	I
SL	Shared Locks		IR	P	I	X	X
S	Signals	I					P
IM	Interrupt Management					P	P
TE	Trivial Entries		I				P
FIP	Fast Interrupt Pragmas			I	I	I	I
PE	Pre-Elaboration of Program Units						
AVS	Access Values That Designate Static Objects						
PTP	Passive Task Pragmas	X		X		P	P
USI	Unchecked Subprogram Invocation						P
DS	Data Synchronization						
DSM	Dynamic Storage Management						P

CIFO Interactions Matrix

		ATI	TSR	TSTS	ATSR	TS	SD
TI	Task Identifiers						
QD	Queuing Disciplines						
SAT	Synchronous and Asynchronous Task Scheduling	P	X	X	X	X	
PI	Priority Inheritance					X	I
DP	Dynamic Priorities		I			P	I
TCS	Time Critical Sections	I	X	X	X	I	
ATI	Abortion via Task Identifiers		I	I	I		
TSR	Task Suspension (and Resumption)	I		I	I	I	
TSTS	*Two Stage Task Suspension (and Resumption)*	I	I		I	I	
ATSR	Asynchronous Task Suspension(and Resumption)	I	I	I		I	
TS	Time Slicing		I	I	I		
SD	*Sychronization Discipline*						
R	Resources	I	I	I	I	P	P
E	Events	I	I	I	I		
P	Pulses	I	I	I	I		
BUF	Buffers	I	I	I	I	P	P
BLK	Blackboards	I	I	I	I		
BRD	Broadcasts	I	I	I	I		
BAR	Barriers	I	I	I	I		
ATC	Asynchronous Transfer of Control	P	P	P	P	P	P
MEA	Mutually Exclusive Access to Shared Data	I			I	I	I
SL	Shared Locks	X	P	P	P	P	P
S	Signals	P	P	P	P		
IM	Interrupt Management	P	P		P	P	
TE	Trivial Entries						
FIP	Fast Interrupt Pragmas		P	P	P	I	I
PE	Pre-Elaboration of Program Units						
AVS	Access Values That Designate Static Objects						
PTP	Passive Task Pragmas	X	X	X	X	X	
USI	Unchecked Subprogram Invocation						
DS	Data Synchronization						
DSM	Dynamic Storage Management	P	P	P	P		

CIFO Interactions Matrix

		R	E	P	BUF	BLK	BRD
TI	Task Identifiers						
QD	Queuing Disciplines						
SAT	Synchronous and Asynchronous Task Scheduling	P	IR				
PI	Priority Inheritance	I			I		
DP	Dynamic Priorities						
TCS	Time Critical Sections	X	X	X	X	X	X
ATI	Abortion via Task Identifiers	P	P	P	P	P	P
TSR	Task Suspension (and Resumption)						
TSTS	Two Stage Task Suspension (and Resumption)						
ATSR	Asynchronous Task Suspension (and Resumption)						
TS	Time Slicing	P			P		
SD	Sychronization Discipline	P			P		
R	Resources						
E	Events						
P	Pulses						
BUF	Buffers						
BLK	Blackboards						
BRD	Broadcasts						
BAR	Barriers						
ATC	Asynchronous Transfer of Control	P	P	P	P	P	P
MEA	Mutually Exclusive Access to Shared Data	P					
SL	Shared Locks	P					
S	Signals	P	P	P	P	P	P
IM	Interrupt Management	P	P	P	P	P	P
TE	Trivial Entries						
FIP	Fast Interrupt Pragmas	P	P	P	P	P	P
PE	Pre-Elaboration of Program Units	I	I	I	I	I	I
AVS	Access Values That Designate Static Objects						
PTP	Passive Task Pragmas	P	P	P	P	P	P
USI	Unchecked Subprogram Invocation	P			P	P	P
DS	Data Synchronization						
DSM	Dynamic Storage Management						

CIFO Interactions Matrix

		BAR	ATC	MEA	SL	S	IM
TI	Task Identifiers		P				
Q D	Queuing Disciplines		P				
SAT	Synchronous and Asynchronous Task Scheduling		P		P		
PI	Priority Inheritance		P	I	I		
D P	Dynamic Priorities				I		P
T C S	Time Critical Sections	X	X	X	X	I	I
ATI	Abortion via Task Identifiers	P	P	P	P	P	X
T S R	Task Suspension (and Resumption)		I		P	P	P
TSTS	Two Stage Task Suspension (and Resumption)		I		P	P	
ATSR	Asynchronous Task Suspension (and Resumption)		I		P	P	P
T S	Time Slicing		I	I	P		X
S D	Sychronization Discipline		P	P	P		
R	Resources		P	P	P	P	P
E	Events		P			P	P
P	Pulses		P			P	P
B U F	Buffers		P			P	P
B L K	Blackboards		P			P	P
B R D	Broadcasts		P			P	P
B A R	Barriers		P			P	P
A T C	Asynchronous Transfer of Control	P	P	P	P	P	P
M E A	Mutually Exclusive Access to Shared Data		P		I		P
S L	Shared Locks		P	P			P
S	Signals	P	P				P
I M	Interrupt Management	P	P	P	P	I	
T E	Trivial Entries		P				
FIP	Fast Interrupt Pragmas	P	I	P	I	P	I
P E	Pre-Elaboration of Program Units	I	P	I	I		P
A V S	Access Values That Designate Static Objects		P				
PTP	Passive Task Pragmas	P	P	P	P	P	P
U S I	Unchecked Subprogram Invocation		P	P	P	X	
D S	Data Synchronization		P				
D S M	Dynamic Storage Management				P		P

CIFO Interactions Matrix

		TE	FIP	PE	AVS	PTP	USI
TI	Task Identifiers					X	
Q D	Queuing Disciplines						
SAT	Synchronous and AsynchronousTask Scheduling		I			X	
PI	Priority Inheritance		P				
D P	Dynamic Priorities		I			X	
T C S	Time Critical Sections	X	X			X	P
ATI	Abortion via Task Identifiers		I			X	
TSR	Task Suspension (and Resumption)		X			X	
TSTS	Two Stage Task Suspension (and Resumption)		X			X	
ATSR	Asynchronous Task Suspension(and Resumption)		X			X	
T S	Time Slicing		X			X	
S D	Sychronization Discipline		P				
R	Resources		P	I		P	P
E	Events		P	I		P	
P	Pulses		P	I		P	
B U F	Buffers		P	I		P	P
B L K	Blackboards		P	I		P	P
B R D	Broadcasts		P	I		P	P
B A R	Barriers		P	I		P	
A T C	Asynchronous Transfer of Control	P	P	P	P	P	P
M E A	Mutually Exclusive Access to Shared Data		P			P	P
S L	Shared Locks		P			P	P
S	Signals		P			P	P
I M	Interrupt Management		P	P		P	
T E	Trivial Entries		I				X
FIP	Fast Interrupt Pragmas	I	I	P			P
P E	Pre-Elaboration of Program Units		I			P	
A V S	Access Values That Designate Static Objects					P	
P T P	Passive Task Pragmas	I		P	P		P
U S I	Unchecked Subprogram Invocation	X	P			X	
D S	Data Synchronization						
D S M	Dynamic Storage Management		P				P

CIFO Interactions Matrix

		DS	DSM
TI	Task Identifiers		
QD	Queuing Disciplines		
SAT	Synchronous and Asynchronous Task Scheduling		
PI	Priority Inheritance		
DP	Dynamic Priorities		
TCS	Time Critical Sections		X
ATI	Abortion via Task Identifiers		P
TSR	Task Suspension (and Resumption)		X
TSTS	Two Stage Task Suspension (and Resumption)		X
ATSR	Asynchronous Task Suspension(and Resumption)		X
TS	Time Slicing		
SD	Sychronization Discipline		
R	Resources		
E	Events		
P	Pulses		
BUF	Buffers		
BLK	Blackboards		
BRD	Broadcasts		
BAR	Barriers		
ATC	Asynchronous Transfer of Control		
MEA	Mutually Exclusive Access to Shared Data		
SL	Shared Locks		P
S	Signals		
IM	Interrupt Management		P
TE	*Trivial Entries*		
FIP	Fast Interrupt Pragmas		P
PE	Pre-Elaboration of Program Units		
AVS	Access Values That Designate Static Objects		
PTP	Passive Task Pragmas		
USI	Unchecked Subprogram Invocation		P
DS	Data Synchronization		
DSM	Dynamic Storage Management		

Deferred Items

The following is a partial list of those subjects not addressed by this document. They are either now under consideration but not yet ready for inclusion, or are topics for future work. This is a partial list in the sense that other topics will emerge as work continues.

Run Time Kernel

Exception Identifiers

Generic I/O

Multiprogramming

Atomic Actions

Nested Transactions

Checkpointing

Distributed Communications

Abortion in a Multiprogramming Environment

Acknowledgement of Members

The following list acknowledges both current members of ARTEWG and those members who served previously for at least a year. Any omissions are unintentional. We apologize in advance and will be happy to correct this list in a future release.

Dock Allen, Control Data Corporation

Neal Altman, Software Engineering Institute

Ted Baker, Florida State University

Mary Bender, CECOM

Eric Beser, Westinghouse Electric Corporation

Mark Borger, Software Engineering Institute

Alan Burns, University of York

Scott Carter, McDonnell Douglas Electronics Systems Company

Dominique Chandesris, CR2A

Janice Chelini, Delco Electronics

Jim Chelini, Raytheon Company

Midge Contreras, Hughes Aircraft Company

Donna Forinash, Texas Instruments

Mary Forthofer, IBM Federal Systems Division

Bob Gable, Lear Siegler

Kathleen Gilroy, Software Productivity Solutions

Steven Goldstein, IITRI

Tom Griest, LabTek Corporation

Mark Hall, Lockheed Aeronautical Systems

Sam Harbaugh, Integrated Software Incorporated

Rakesh Jha, Honeywell Systems Research Center

Bruce Jones, Verdix Corporation

Juern Juergens, Encore Computer Corporation

Mike Kamrad, Unisys

Peter Krupp, MITRE

Joan Kundig, Allied Signal Aerospace Company

Larry Lehman, Integrated Systems Inc.

John Litke, Grumman Corporate Research Center

Carl Malec, Boeing Aerospace and Electronics

Fred Maymir-Ducharme, Grumman

Charles McKay, University of Houston-Clear Lake

Glen McMillen, Boeing Computer Services

Kevin McQuown, General Dynamics

Mike Mills, United States Air Force

Offer Pazy, Intermetrics

Gilles Pitette, CR2A

Richard Powers, Texas Instruments

Tom Quiggle, Telesoft

Moshe Rabinowitz, InterAct Corporation

Roger Racine, Charles Stark Draper Lab

Marc Richard-Foy, Alsys

David Ripps, Industrial Programming, Incorporated

Pat Rogers, Software Arts and Sciences/SBS Engineering Incorporated

Dan Stock, RR Software Incorporated

Daryl Winters, Sanders Associates

Mike Victor, Raytheon Company

Dave Vogel, IBM

Barry Watson, IITRI

Andy Wellings, University of York

Russ Wilson, Boeing Aerospace and Electronics

Submission of Comments

For submission of comments on this *Catalogue of Interface Features and Options*, we would appreciate them being sent via ARPANET/MILNET to the following address:

artewg-interface@ajpo.sei.cmu.edu

If you do not have network access, please send the comments by mail to:

Dr. Charles W. McKay
University of Houston at Clear Lake
2700 Bay Area Blvd.
Box 447
Houston, TX 77058
Attn : ARTEWG comments

For mail comments, it will assist us if you are able to send them on a 360K, 5-1/4 inch floppy diskette, under DOS (or MS-DOS) compatible format. But even if you are able to manage this, please do send us a paper copy to ensure their evaluation, in case of problems in reading the diskette.

All comments will be sorted and processed electronically (of course) in order to speed their analysis. To aid this process you are requested to precede each comment with a three-line header, as follows:

!section ...
!version **Release 3.0**
!topic ...

The section line includes the section and paragraph number, your name or affiliation (or both), and the date in ISO standard format (yy-mm-dd).

The version line should contain the current release identifier (for example, "Release 3.0") in order to distinguish comments from future releases.

The topic line should contain a one-line summary of the comment. This line is essential, and you are kindly asked to avoid such topics as "Typo" or "Editorial comment" which will not convey any information when viewed individually (such as in a table of contents).

Thank you for your work and support.