An Analysis of the Implementation and Execution-Time Impact of Ada 9X Real-Time Features

Thomas J. Quiggle
Gary J. Dismukes

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121

quigglet@ajpo.sei.cmu.edu
gary@telesoft.com

1.0 Introduction

This document summarizes the results of a one-month special study of proposed Ada 9X real-time features, requested by the Ada 9X project office. The study was performed in conjunction with TeleSoft's Ada 9X User/Implementor contract.

Section 1 provides an overview of the document itself. Section 2 provides an overview of the TeleSoft compiler and run-time system. Section 3 discusses a base implementation of Protected Records that does not support Requeue, Selective Entry Call, or Asynchronous Transfer of Control. Section 4 discusses the incremental cost of providing Requeue. Section 5 discusses the costs associated with Selective Entry Call -- exclusive of the implementation of an *abortable final part.* Section 6 discusses the implementation of Asynchronous Transfer of Control. Section 7 summarizes our findings.

1.1 Purpose

This document serves to identify the costs associated with several proposed Ada 9X features that address requirements for real-time applications. Two distinct forms of cost are considered. First is the cost incurred by Ada vendors to implement the proposed features. This cost can be further broken down into the one-time cost associated with modifications made to components of the compilation and run-time environment that are common to the vendor's product line, and the recurring costs that are associated with modifications to components that are specific to each targeted platform. Second is the execution-time cost incurred by users of the proposed feature. This second cost is of critical importance to users building embedded real-time applications.

1.2 Intended Audience

This document is directed at members of the Ada 9X Project office, the Mapping Team, and reviewers of the proposed language features. Distribution need not be restricted, as no TeleSoft proprietary information has been included. It is assumed that the reader has some degree of familiarity with Ada implementation issues. No knowledge of TeleSoft's specific implementation is assumed.

1.3 Scope

This document provides an overview of the costs associated with Protected Records, Requeue Statements, Selective Entry Call (without an abortable final part), and Asynchronous Transfer of Control, as described in version 4.0 of the Mapping Document. Although reference is made to various design considerations associated with these features, this document is not intended to serve as a detailed design document for the implementation of the features discussed.

1.4 References

ACM, *Ada Performance Issues*, *a Special Edition of Ada Letters*, V 10, N. 3, Winter 1990.

IEEE, "Real-time Extension for Portable Operating Systems," P1003.4/D10, February 6, 1991.

IEEE, "Threads Extension for Portable Operating Systems," P1003.4a/D5, December, 1990.

Intel, "iRMK I.2 REAL-TIME Kernel Reference Manual," 1988.

Intermetrics, "Draft Ada 9X project Report - Ada 9X Mapping Document Volume II, Mapping Specification," December 1991.

Intermetrics, "User/Implementor Implementation Modules."

Ready Systems, "RTAda Real-Time Ada User's Guide," Document811112002, March 1990.

Software Components Group, "pSOS+/68K User's Manual," Document KX68K-MAN, 1989.

Software Components Group, "Timing Reference for pSOS+/68020"

TeleSoft, "TeleAda-Exec Users Manual Version 1.0 MC68020/30,"

1.5 Definitions

`RSP:` Run-time Support Packages - a collection of TeleSoft-provided packages that support the execution of Ada programs. Calls to the RSP are generated by the compiler, and are not present in the source representation of an application program.

`target`: The computer on which a compiled Ada program is intended to execute.

`target architecture:` An abstract target machine characterized by the instruction set it executes. For example, TeleSoft's Sun3-host compiler and Vax/E68 cross compiler both generate instructions for the Motorola MC68020 target architecture.

`target environment:` The collection of software components that execute on a target machine independent of the Ada program. For example on a Sun Host, the target environment consists of the SunOS Unix implementation and additional (optional) layered products.

2.0 TeleSoft Compilation and Run-Time System Overview

In order to evaluate the applicability of TeleSoft's cost estimates to other compilation and run-time systems, and to understand the distinction between one-time and recurrent costs for TeleSoft, we offer the following brief overview of TeleSoft's Compilation and Run-Time System.

2.1 Compiler Overview

The TeleSoft compiler is a single Ada program that operates in three phases: the Front End, the Middle Pass, and the Code Generator. The Front End accepts Ada source code as input and produces a high-level, machine-independent intermediate representation of the input text known as High Form. The Front End performs syntactic and semantic analysis and provides error diagnostics. The Middle Pass accepts the High Form input and produces a low-level, machine-independent graphical representation called Low Form. The Code Generator translates Low Form into object modules for a specific machine architecture.

The Front End and Middle Pass are common to all TeleSoft compilers (although each does contain a single parameterization package whose body is target-dependent). The Code Generator is unique to each targeted architecture. The cost of performing modifications to the Front End and Middle Pass are incurred once for the entire TeleSoft product line. Modifications to the Code Generator must be performed for each targeted architecture.

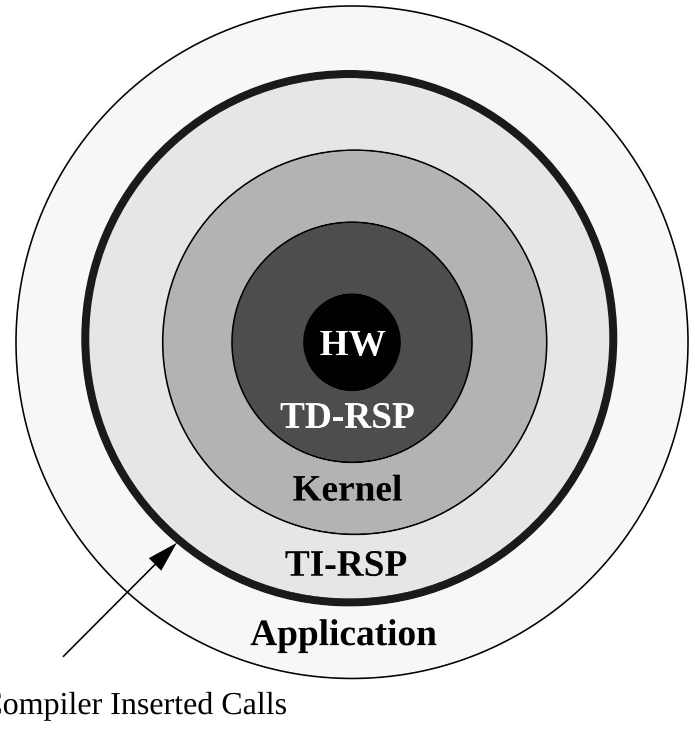
2.2 Run-Time System Overview

TeleSoft's Ada Run-time System is divided into three interdependent components:

- Target-Dependent RSP

- Kernel
- Target-Independent RSP

The relationship between these components follows the classic “onion model” as shown below:



Each of these layers is described briefly below:

- **`HW`**: The hardware (and software) environment targeted. This layer may consist of a bare board, an embedded target running a real-time executive, or a host computer running a full-featured operating system. The facilities available at this layer dictate the amount of work required to implement the next layer.
- **`TD-RSP:`** The Target-Dependent run-time Support Packages (TD-RSP) defines a common virtual machine that the surrounding layers interface to. The implementation of the TD-RSP is unique to each platform upon which Ada is targeted.
- **`Kernel:`** Provides operations on abstract threads of control. The operations provided include creation, destruction, suspension, resumption, priority manipulation, etc. The operations provided are the minimal set required for the implementation of Ada tasking, and are a subset of the facilities generally provided by a real-time executive or a general-purpose operating system.
- **`TI-RSP:`** The Target-Independent run-time Support Packages (TI-RSP) provide run-time support for implementing various Ada language constructs. Constructs implemented by the TI-RSP include various attributes (e.g. ’Image, ’Width), the NEW operator for heap allocation, and all tasking operations. The majority of the TI-RSP is dedicated to the implementation of Ada tasking. Calls into the TI-RSP are inserted by

the compiler into the Low-Form intermediate representation of the application program. The tasking portion of the TI-RSP interface is a high-level interface that is similar to the ARTEWG MRTSI interface.

- **`Application`**: This layer represents the user's application program.

The Target-Independent RSP constitutes the majority of the run-time system, and is further subdivided into tasking support and non-tasking support. The cost of performing modifications to the TI-RSP are incurred once for the entire TeleSoft product line. Modifications to the TD-RSP must be performed for each target environment.

The Kernel is neither a component of the TD-RSP nor TI-RSP. There exists a standard implementation of the Kernel interface that relies exclusively on the functionality provided by the TD-RSP, but adaptors are encouraged to provide alternate implementations for target systems where appropriate. Such adaptations support two distinct classes of target environments: those in which the Ada run-time system has exclusive control over the allocation and scheduling of processor resources, and those in which the Ada run-time system must coexist with a foreign scheduler such as a host operating system or real-time executive. The former class of systems represents a true bare target, in which only a single Ada program resides. The latter is often used for applications which require multiple Ada programs, or mixed-language programs. TeleSoft believes the majority of all current embedded Ada applications (for all vendors' compilers) utilize some form of foreign scheduler. Continued efficient support of this class of systems is of vital interest to TeleSoft and the Ada community.

The TeleSoft RSP was specifically designed to facilitate both classes of systems by isolating all scheduling operations into a Kernel package that is separate from the packages that implement the various Ada semantic operations. Implementations of the Kernel package exist for both bare targets and for targets running a foreign scheduler. A vital consideration for the design of Ada 9X real-time features is the ability to preserve the separation of Ada semantic operations from Kernel facilities. Only by doing so can we preserve the ability to adapt the Ada run-time system to a new scheduler in a cost effective manner.

3.0 Protected Records

Protected Records is the feature that TeleSoft has the greatest experience with. To date, we have implemented a subset of the protected records functionality that is consistent with the Implementation Module. Summarized below is the content of that implementation and the effort expended.

3.1 Initial Implementation

Each protected record type declaration defines a composite type and operations on that type. The compiler extends the protected record's component list to include a component of type Protected_Record_Control_Block (hereafter referred to as PRCB), or an access to the PRCB (see below). Operations on the protected record type are provided via the protected operations defined in the protected record type declaration. Protected operations may be functions, procedures, or entries.

3.1.1 In-line vs. Out-of-line Locking Operations

In the implementation of the protected operations, run-time actions must both precede and follow the user-provided sequence of statements. Two alternative implementation models were examined.

- **`Model 1:`** The body of a protected operation is generated as a directly-callable subprogram containing prologue and epilogue calls to run-time service routines that perform the necessary semantic actions for locking, barrier evaluation, etc.
- **`Model 2:`** Calls to a protected operation are implemented as calls to a run-time service routine. This run-time routine accepts as parameters a pointer to a parameter block for the current call and an access to a subprogram containing the actual protected operations. The run-time routine performs the necessary prologue (locking) operations, indirectly calls the subprogram containing the protected operations, and upon return, performs the necessary epilogue operations (barrier evaluation, queued caller processing, and unlocking).

Early experience with prototypes of the above models indicated that Model 1 significantly out performed Model 2 for protected subprogram calls and for protected entry calls that do not queue. Consequently Model 1 was adopted. A hybrid approach in which procedures and functions use Model 1 and entries use Model 2 was considered, but rejected in the interest of simplifying the implementation. This decision was made in conjunction with completing the Implementation Module for Protected Records, which did not include any form of Selective Entry Call (SEC). This decision was later revisited when evaluating the Implementation Module for SEC. The choice of directly-callable entry bodies with out-of-line RSP calls for the prologue and epilogue actions proved to hinder the implementation of SEC. Consequently protected record entries were re-implemented in conjunction with our run-time prototype of Selective Entry Call.

3.1.2 Allocation of Protected Record Control Blocks

Some systems require that PRCBs be allocated disjoint from the user-declared components of a protected object. At least three circumstances result in such a requirement: 1) segmented architectures in which the run-time system resides in a segment distinct from the application program due to segment-size limitations, 2) systems with security requirements mandating that all run-time system data structures be protected from inadvertent or malicious modification by applications programs, and 3) multiprocessor systems in which the memory architecture requires that “test and set” cells used for multiprocessor locks must be allocated from specific regions of memory (for example, specific pages in which cache write-through or cache snooping is guaranteed).

In such disjoint-memory systems, the allocation and initialization of the PRCB must be performed by the run-time system. An access value is returned from the run-time system to the application program for subsequent use in requesting operations on the protected record object. This access value may not be meaningfully dereferenced from the context in which the application code executes. Such access values can be controlled by the run-time system; specifically, the access value can be checked for validity and access rights prior to operating on the designated PRCB.

For systems where the application and run-time system share data visibility, the actual PRCB could be declared as a component of the protected object. In this case, the space for the PRCB can be allocated contiguous with the user-declared components of the protected object, and the compiler can perform the initialization of the PRCB in the same manner that it would perform default initialization of the user-declared components. Access values can still be used for passing PRCB parameters to the run-time system; thus providing a common compiler/run-time interface for both uniform- and disjoint-memory targets.

For the purposes of the initial implementation, the PRCBs were dynamically allocated and initialized by the run-time system. This corresponds to the more restrictive case wherein the application code and run-time system do not share data addressability. The motivations for selecting this allocation model are two-fold: first to insure that the design takes into consideration disjoint memory architectures; and second, to simplify the compiler implementation by eliminating the need to understand and initialize the PRCB structure.

3.1.3 Call-Outs

TeleSoft’s 68020 Code Generator utilizes a local display to reference up-level data objects. A local display is only built for subprograms that make up-level references and/or call subprograms that make up-level references. If a display is required, it is copied from the caller’s local display and extended to include an entry for the caller’s local frame. The subprogram’s lexical level determines the size of the display copied. Elements of the local display are treated like any other local data item, and may be allocated to registers. A call to a lex-level zero subprogram will never result in the copying of the caller’s local display. Consequently, an indirect call to an entry body or barrier evaluation subprogram via a lex-level zero routine must pass an additional parameter from which a local display can be built. The call-out mechanism will vary from code-generator to code-generator.

To handle call-outs from the TI-RSP (an interface consisting of lex-level zero subprograms) to application subprograms, a new target-dependent interface was created. This interface provides the necessary indirect-call capability to call barrier evaluation functions and entry bodies. The implementation for the 68020 required slightly less than a day of effort.

3.2 Implementation Costs

The following sections summarize the effort expended to implement protected records exclusive of selective entry call and asynchronous transfer of control.

3.2.1 Compiler Implementation

The Front End changes to support Protected Records consisted primarily of changes to the high-level symbol table to represent a protected unit and its corresponding protected type. Due to the similarity with record types and task types, we were able to reuse a significant amount of existing code to support the creation of the new symbols. There were also changes to support calls to protected operations and references to protected components, which required some differences in representation relative to task entry calls and normal record component references.

The modifications to the Middle Pass were more extensive, but it was possible to encapsulate the majority of new code within two new packages. One package performs all symbol table and low-level intermediate code generation specific to protected records and operations. The other package provides functions for generating the run-time calls required to support PRCB initialization and processing of protected operations (prologue, epilogue, etc.). As in the Front End, a high degree of reuse of existing code was possible by building on the existing support for packages, subprograms, and records. A protected unit is represented essentially as a package with an associated record type where the package provides the protected operations and the protected components are associated with the record type.

The combined Front End and Middle Pass design and implementation effort for the protected records prototype took approximately 20 person-weeks. Note that no changes were required to the back-end (code generation) phase of compilation. We estimate that the effort to transition the prototype to a complete implementation is moderate. One set of changes we anticipate making in this transition would be to convert the support for protected entry calls to use the out-of-line call model described earlier.

3.2.2 Run-time Implementation

The implementation of protected records within the run-time system primarily consisted of defining the Protected Record Control Block (PRCB) and the prologue and epilogue operations that act upon it. This work was minimally impactive of the existing run-time system code. The areas in which the existing run-time code was impacted are:

- The addition of checks to all potentially-blocking run-time service calls to determine if the calling task is executing a protected operation
- Abort deferral for tasks executing protected operations
- Addition of several components to the Task Control Block

The design and implementation of protected records took approximately10 person-weeks of effort, and increased the overall size of the target-independent tasking run-time by approximately 20% (based on a raw SLOC count).

3.3 Execution-Time Costs

The initial implementation of Protected Records was performed in the environment that traditionally has yielded the highest performance for Ada 83 tasking. Specifically, the implementation was built using TeleSoft's standard Ada Kernel with the application running in supervisor mode. In this mode of operation, modifications to a task's Kernel state (such as the task's priority or interrupts enabled state) can be executed without change in processor mode, and can be in-lined into the appropriate TI-RSP routine by the global optimizer. Following this approach, we attempted to evaluate the best attainable performance for protected records. Table 1.0 summarizes the performance of various constructs when executed on a Motorola MVME135-1 embedded target (20 MHz 68020, zero wait-state).

Test Name	Operation Tested	uSec
pr000001	PR Function Call	13.6
pr000002	PR Procedure Call - PR has no entries	14.9
pr000003	Simple PR Entry Call - PR has only one entry	33.9
pr000004	Simple PR Entry Call - PR has two entries	39.8
pr000005	Simple PR Entry Call - PR has ten entries	40.7
t000001	Task Entry Call and return. 1 task, 1 entry, no select	107.2
t000002	Task Entry Call and return. 1 task in pkg., 1 entry, no select	108.8
t000003	Task Entry Call and return. 2 tasks, 1 entry each, no select	109.8
t000005	Task Entry Call and return. 10 tasks, 1 entry each, no select	108.2
t000007	Task Entry Call and return. 1 task, 1 entry, null accept body	72.8

TABLE 1. Task and Protected Record Execution Times

The above results indicate that Protected Records significantly outperform active tasks in this environment. However, we anticipate significant performance degradation when moving to other environments if the same ceiling priority mechanism is used for mutual exclusion. On certain target systems in which scheduling actions are performed by a foreign kernel, the implementation of ceiling priority locks for protected records will require calls into the kernel to modify the task's priority and time-slicing status.

As a case in point, if implemented in conjunction with the Software Components Group's pSOS+ real-time executive, each protected operation would require a minimum of 4 Kernel calls:

1	Raise the caller's priority to the ceiling	(T_SETPRI = 44.4 uSec)
2	Disable time-slicing for the task	(T_MODE = 19.0 uSec)

3	Lower the caller's priority to its previous value	(T_SETPRI = 44.4 uSec)
4	Re-enable time-slicing for the task	(T_MODE - 19.0 uSec)

This represents an additional cost of 126.8 uSec for the required Kernel operations when executing on a 20Mhz 68020 processor. The resulting execution times for protected operations are expected to approach the execution time for a rendezvous with an active task, thereby obviating much of the advantage of protected records. Similar results are expected for implementations built with other popular real-time executives.

For such target environments, we propose to use a locking mechanism built on a potentially blocking lock object whose state is maintained in user space (and therefore efficiently accessible). We are encouraged by the recent move of the specific ceiling semantics for mutual exclusion of protected operations to the real-time annex, thus permitting the use of an efficient locking mechanism in an environment where priority adjustments are prohibitively expensive.

4.0 Requeue

4.1 Implementation

Our existing implementation only addresses requeue for protected records. Requeue statements from within accept statements have not been implemented. Based on a limited evaluation, we anticipate that the required changes for requeue on a task entry will be small.

4.1.1 Compiler Implementation

Given an existing implementation of the core protected records features, the incremental cost of adding compiler support for requeue is quite small. In our prototype implementation, both features were implemented together. We have no data on the cost of implementing requeue separate from the core protected records features. We believe that the requeue component of the implementation constituted at most 10 percent of the overall implementation cost. For the compiler, requeue is implemented as a call to a run-time service routine that closely parallels the implementation of an entry call.

4.1.2 Run-time Implementation

Requeue is implemented entirely within the Target-Independent RSP, and as such, its implementation is a one-time cost. The implementation of Requeue is very similar to that of a normal protected record entry call. First the requeue operation checks to see if the destination entry belongs to the same protected record as the surrounding entry body. If so, the necessary prologue code has already been executed, otherwise the prologue operations for the destination entry are performed.

Next the entry barrier is evaluated. If the barrier is open, the epilogue code for the outer entry call is performed (in case the task blocks on a requeue for the destination entry) and the entry body is executed. Upon return from the entry body, the epilogue code for the destination entry is performed.

The requeue operation sets a flag in the caller's TCB indicating that the entry call was requeued. This flag is checked by the epilogue code for the original entry call to determine if the epilogue for the outer entry has already been performed by an inner requeue.

4.2 Execution-Time Costs

The need to check for a requeue when returning from an entry body introduces a distributed cost to the implementation of protected record entry calls. This overhead is on the order of a single test and branch, plus the cost of maintaining the caller information in the PRCB. We do not yet have performance figures for the execution of a requeue statement as implemented in our prototype. Based on examination of the prototype code, we anticipate that the execution-time for a requeue operation will be slightly greater than the cost for a normal entry call. The variation is not expected to exceed 10 per-cent.

5.0 Selective Entry Call

This section describes the implementation of Selective Entry Call in the absence of an abortable final part. Abortable final parts are discussed separately in section 6.0.

5.1 Implementation

The implementation of Selective Entry Call is very similar to that of Selective Waits in Ada 83. The compiler generates code that initializes a data structure describing the alternatives to the select statement and passes the data structure to a run-time service routine for further evaluation. The run-time routine evaluates the alternatives relative to the current state of the called entries and returns an integer value indicating which alternative was selected for execution. If no alternative can be immediately selected, the run-time service routine will block the executing task until one of the alternatives can be selected. The value returned is used in a case statement to select the appropriate statement sequence to execute following the entry call. For example, the select statement:

```
select
 when condition2 =>
    server1.entry1(...);
    statement_sequence1;
or
 when condition2 =>
    server2.entry2(...);
    statement_sequence2;
or
 delay 1.0;
 statement_sequence3;
end select;
```

would be translated into something of the form:

```
declare
 Header : Selective_Call_Header(Num_Alternatives => 3);
begin
 <initialize components of the selective call header>
 case RTS.Evaluate_SEC(Header) is
    when 1 => statement_sequence_1;
    when 2 => statement_sequence_2;
    when 3 => statement_sequence_3;
 end case;
  RTS.Complete_Select(Header);
exception
 when others => RTS.Complete_Select(Header);
                 raise;
end;
```

5.1.1 Compiler Implementation

For purposes of this study, a prototype implementation of selective entry calls was undertaken that involved changes only to the run-time support packages. No compiler modifications to support this feature have been made to date. The compiler implementation is not believed to be difficult, however, since the model for selective entry calls is so closely analogous to that for Ada 83 selective wait statements.

The Front End support for selective wait statements would be generalized to allow entry call alternatives and abortable final parts, which will also subsume the Ada 83 conditional and timed entry call constructs. It also appears that the Middle Pass implementation will be relatively straightforward to extend to support selective entry calls due to the strong similarities with selective wait statements. It should be possible to parameterize the existing Middle Pass support for selective wait to allow for the new forms of alternatives and to generate the run-time calls appropriate for the selective entry call case. Alternatively, the selective wait generation code could be duplicated and specialized to selective entry calls, including reusing the existing Middle Pass support for the special cases of Ada 83 conditional and timed entry calls. In any event, we estimate the compiler adaptation cost to be easy or moderate, probably on the order four person-weeks of effort. Note that this does not include the Middle Pass effort to support abortable final parts.

5.1.2 Run-time Implementation

In our prototype implementation, the existing data structures for describing alternatives to Ada 83 selective wait statements have been generalized to include alternative forms for entry call alternatives. The previous selective wait header has been reformulated in terms of the new more general form. An additional header type has been introduced for selective entry calls. Some of the original type names have been modified to more closely parallel the nomenclature used in the Mapping Document.

The implementation of selective accept statements was derived from the existing support for Ada 83 selective wait statements. The implementation was modified to examine alternatives in their lexical order, instead of the “round-robin” order previously used. This change represents a minor simplification.

The existing support for conditional and timed entry calls will be retained. The Middle Pass will recognize selective entry calls containing a single entry call alternative and one delay alternative or an else part, and implement such calls exactly as in their Ada 83 counterpart. We believe that the more restrictive implementation of the Ada 83 constructs will significantly out-perform an equivalent implementation of the more general selective entry call mechanism. We wish to prevent an unexpected degradation in performance for syntactically identical constructs when user’s upgrade from Ada 83 to Ada 9X.

5.1.2.1 Allocation of Queue Elements

For the prototype implementation, we adopted the “agent task” approach proposed by Tucker Taft in electronic mail to the User/Implementor teams. This mechanism avoids the need for a client task to reside on more than one entry queue when executing a selective entry call. Instead, a separate “agent” waits on an entry queue on behalf of the task executing the selective entry call. The agent task consists of a minimal TCB that does not correspond to a thread of control, but does contain linkage and state information used in processing a rendezvous. The agent task replaces the QEL construct proposed in the Implementation Module and has the advantage of eliminating an intermediate queue element for normal entry calls (i.e., those not part of a selective entry call).

To preserve the ability to execute in disjoint memory architectures, the allocation and management of agent TCBs is performed by the run-time system, not the compiler. In our prototype implementation, we pre-allocate a fixed number of agent TCBs at elaboration time and place them on a free list. As agent TCBs are needed, the free list is examined for available agents. If none are available, a new agent is allocated from the global heap. When an agent is no longer needed, it is returned to the free list. The maximum size of the free list is bounded by a run-time constant. When an agent task is returned to the system and the free list contains the maximum number of reserved agents, space for the returned agent is simply returned to the global heap. In a production compiler, the initial and maximum free list sizes would be governed by configuration parameters, and the allocation of additional agents from the global heap could be optionally replaced by a raise of STORAGE_ERROR.

5.1.2.2 Impact on Accept Statements and Entry Bodies

The previously existing support for accept statements and entry bodies contained checks to determine if an enqueued caller was executing a timed entry call that had expired. Upon encountering a caller that has timed out, the acceptor simply ignores the entry call and re-examines the queue. This check has been compounded with a check to determine if the caller is an agent enqueued on behalf of a selective entry call, and if so, execute the appropriate code to claim the select header prior to initiating the rendezvous. Similar to the processing of an expired timed entry call, if the select header cannot be claimed the caller is disregarded and the entry queue re-examined. This check for a selective entry call need not introduce a distributed cost for normal entry calls beyond what already exists for timed entry calls.

In the Ada 83 implementation, upon conclusion of an accept statement, any exception raised within the accept body is latched in the caller's TCB. The calling task is then made eligible for execution. This exit code for a rendezvous has been modified to determine if the caller is an agent executing on behalf of a selective entry call. If so, additional operations are required to resolve potential race conditions with the task executing the selective entry call, and to insure that the true caller (as opposed to the agent) is made eligible to execute. This additional check adds a small distributed cost to all rendezvous.

5.1.2.3 Resolution of Race Conditions

Since selective entry calls are permitted to contain entry call alternatives for protected record entries, the protection of the header components cannot use the same (potentially blocking) mutex mechanism currently used for TCBs. To do so would violate the non-suspension rule for execution within a protected record when servicing a queued entry call for a caller executing a selective entry call. The Implementation Module suggests the use of an atomic test-and-set cell to resolve race conditions between multiple servers attempting to provide service for the same selective entry call, coupled with a signal/wait mechanism to resolve race conditions between the winner of such a race and the task executing the selective entry call. A new target-dependent package was created to provide access to an architecture-specific atomic test and set instruction. TeleSoft's kernel interface does not provide an event mechanism. Instead, we utilized interrupt lockout to protect several critical sections in which the winners of race conditions are determined. A better long-term solution is desirable.

5.1.2.4 Re-Implementation of Protected Record Entries

As discussed in section 3.1.1, the initial implementation of protected records entries generated a directly callable subprogram that contained out-of-line calls to RSP prologue and epilogue routines. This model does not scale to selective entry calls in which a portion of the prologue routine must be performed prior to committing to the entry call, and the parameters to the entry call must be evaluated prior to the execution of a run-time service routine that selects the alternative to execute. Consequently, the implementation of protected record entries was modified to use a model in which the entry body is not directly called in response to an entry call statement, but is called indirectly by the run-time system after executing the appropriate prologue code. The prologue code for a simple entry call differs from that of a selective entry call in that there is no need to claim an intermediate select header for a simple call. Furthermore, if a simple entry call must queue, the executing task waits directly on the entry queue, whereas for a selective entry call a separately allocated agent waits on the entry queue.

5.1.2.5 Run-Time Implementation Cost

The prototype implementation took approximately 2 person weeks of effort plus an additional week to re-implement protected record entries. As a rapid prototype, the implementation has not undergone the normal process of design review and code inspection. It is likely that the resulting code contains defects not uncovered by unit test, and that performance improvements are possible. We would expect a more formal implementation to take an additional 6-8 weeks of effort, including documentation and reviews. This is a one-time cost for modification to the target-independent RSP.

The introduction of new target-dependent functionality to implement the claim mechanism results in a recurring cost that will be incurred for each target. This cost should be quite small. It should be possible to implement the necessary functionality in only a few days.

5.2 Execution-Time Costs

Although the implementation of Selective Entry Call closely resembles that of the Ada 83 Selective Wait, the introduction of intermediate queueing agents, coupled with the additional logic to resolve races that do not exist for Selective Wait, introduces significant additional overhead. Table 2.0 summarizes the execution times for various forms of selective entry calls and selective accepts.

Test Name	Operation Tested	uSec
pr000006	Selective Entry Call - PR has two entries, alternately open	249.2
pr000007	Selective Entry Call - PR has ten entries, first is open	294.5
pr000008	Selective Entry Call - PR has ten entries, last is open	1723.4
t000004	Task Entry Call and return. 1 task, 2 entries in select	148.2
t000006	Task Entry Call and return. 1 task, 10 entries in select	275.1

TABLE 2. Selective Entry Call Execution Times

Test Name	Operation Tested	uSec
t000009	Selective Entry Call. two alternatives, alternately open	207.1
t000010	Selective Entry Call, ten alternatives, first always taken	307.5
t000011	Selective Entry Call, ten alternatives, last always taken	1348.2

TABLE 2. Selective Entry Call Execution Times

The large differential between tests pr000007 and pr0000008, and between t000010 and t000011 indicate that the overhead associated with adding and removing agents from entry queues is significant.

6.0 Asynchronous Transfer of Control

6.1 Implementation

The implementation of an abortable final part adds complexity to the selective entry call implementation by adding the requirement that the case statement generated for controlling selection among alternatives may need to be re-executed when an abortable final part is initially selected and then aborted.

We have examined two possible mechanisms for aborting a sequence of statements and returning control to the appropriate event alternative. The first involves propagating the abort event in a manner similar to the existing support for exception handling. The second involves saving the state of task prior to executing the generated case statement, and directly restoring that state when the sequence of statements is aborted. Hereafter we will refer to the first model as the *propagation* model, and the second as the *long-jump* model. Each of these models is detailed below.

Consider a select statement of the form:

```
select
 when condition2 =>
    server1.entry1(...);
    statement_sequence1;
or
 when condition2 =>
    server2.entry2(...);
    statement_sequence2;
then abort
 statement_sequence3;
end select;
```

6.1.1 The Propagation Model

For the propagation model, we would introduce additional control flow in the form of a loop statement that is exited when the abortable final part completes successfully or one of the entry call alternatives is executed. The loop is repeated if an abort propagates out of the abortable final part. This additional control flow is shown by the following pseudo-code:

```
declare
 SEC_Header : Header_Type;
begin
  <initialize components of selective entry call header>
  loop
    case RTS.Evaluate_SEC(Header) is
      when 1 => statement_sequence_1; exit;
      when 2 => statement_sequence_2; exit;
      when 3 =>
        begin
          statement_sequence_3;
          exit;
        exception
          when abort  => null;
          when others => raise;
        end;
    end case;
  end loop;
  RTS.Complete_Select(Header);
exception
  when others => RTS.Complete_Select(Header);
                 raise;
end;
```

In this model, the abandonment of the abortable final part is expressed in terms of exception propagation. It is already the case that, when an exception propagates through a scope containing dependent tasks, handlers are inserted by the compiler to call the necessary run-time routines to await dependent task termination. Other forms of finalization are also handled in this manner. The same mechanism can be used for the propagation of abort events. However, the implementation cannot use a normal Ada exception for performing the abort propagation, as such an exception could be caught and handled by an others clause in an exception handler contained within the abortable final part. Instead, we propose that the compiler-inserted handlers be specially marked to distinguish them from user-supplied handlers. The propagation of an abort event would only execute handlers that are so marked. The propagation of exceptions would continue to execute both classes of handlers.

The Middle Pass of the compiler already classifies exception handlers as being user-supplied or compiler-generated and includes this classification in the Low-Form intermediate representation. However the existing 68020 code-generator does not make use of this classification information. The Code Generator would need to be modified to include the classification information in the exception tables produced in the generated object code. We believe this to require only a small amount of effort. A special abort propagation routine would be needed. This routine could be readily derived from the existing exception propagation code. The effort required for other code generators may vary.

In this model, all finalization operations (including awaiting dependent task termination) will be automatically performed as a consequence of the propagation of the abort event. The initiation of asynchronous transfer need only invoke the abort propagation routine. Details of how abort initiation could be performed by the run-time system are described later.

6.1.2 The Long-Jump Model

The Long-Jump abandonment mechanism does not require additional control constructs, but does require an additional run-time service call to save the state of the executing task prior to executing the selective entry call. The code required would resemble the following:

```
declare
 SEC_Header : Header_Type;
begin
  <initialize components of selective entry call header>
  setjmp(Header.Saved_State);
  case RTS.Evaluate_SEC(Header) is
    when 1 => statement_sequence_1;
    when 2 => statement_sequence_2;
    when 3 => statement_sequence_3;
  end case;
  RTS.Complete_Select(Header);
exception
  when others => RTS.Complete_Select(Header);
                 raise;
end;
```

In the above pseudo-code, the state-saving operation is defined in terms of the common Unix setjmp primitive, and for Unix targets the available setjmp operation is sufficient. For non-Unix targets, the equivalent mechanism will need to be created from scratch. Note that the saved state is stored in the select header so it can be made available to subsequent run-time routines that are used in implementing asynchronous transfer of control.

The longjmp operation that corresponds to the setjmp would be performed by the run-time system in implementing the abandonment of an abortable final part. Since the stack cannot be carved back prior to finalization, this longjmp cannot be issued until all necessary finalization has been performed on interior scopes.

A final selection of the appropriate mechanism awaits the completion of the design for finalization.

6.1.3 Run-time Implementation

The run-time implementation of asynchronous transfer of control can be divided into four distinct operations:

1. The processing performed by the task executing the selective entry call to evaluate the select alternatives and initiate the abortable final part.
2. The processing performed by a server accepting an open entry call alternative (or delay alternative expiring) for a selective entry call with abortable final part.
3. The mechanism used by a server to effect the abandonment of the abortable final part.
4. The processing performed by the task executing the abortable final part to effect the asynchronous transfer.

Each of these operations is described below.

6.1.3.1 Processing of the Selective Entry Call

The processing of the selective entry call containing an abortable final part proceeds exactly like the processing of a selective entry call that does not contain an abortable final part. The alternatives are processed in the order given. If any event alternative can be immediately selected, it is selected and the abortable final part is not executed. If no alternative can be immediately selected, the select header is updated to indicate that the calling task is executing an abortable sequence of statements, and the value returned by the run-time service routine indicates that the abortable final part is to be executed. We estimate that the additional cost to implement the processing of an alternative for an abortable final part will not exceed one person-week.

6.1.3.2 Selection of an Event Alternative

When a queued entry call is accepted, the server executing the accept statement (or entry body) must check the claimed select header to determine if the calling task is executing an abortable final part. If so, the server must initiate the abandonment of the abortable final part by the caller. The mechanism used to perform this initiation is discussed in the following section.

TeleSoft's Kernel interface does not contain provisions for modifying non-Kernel data structures upon expiration of a delay request. The existing facilities only provide for the timed suspension of a task requesting a delay and the canceling of a timed suspension by another task. We have limited our kernel interface to these operations because we believe that these facilities can be supported by a wide variety of schedulers (our experience to date supports this belief).

On many systems, it is impossible for a task to be both executing and receive asynchronous notification of delay expiration. For such systems, the presence of a delay alternative for a selective entry call will either require the creation a high-priority agent task to delay on behalf of the task executing the selective entry call, or the implementation will have to resort to a polling scheme. Neither alternative is very attractive.

The necessary modifications to the run-time system to initiate abort upon selecting an entry call alternative are expected to incur a one-time cost of less than one person-month to implement. The modifications to support initiation of abort upon expiration of a delay alternative involve a fundamental change to the Kernel interface to permit a kernel operation to invoke an Ada semantic operation. This change would require several person-months.

6.1.3.3 Notification of Asynchronous Abort

The mechanism used to notify a task of a pending abort depends heavily on the environment in which the application will execute. As discussed in section 2.2.1, TeleSoft supports target environments in which the Ada run-time system has complete control over task scheduling, as well as environments in which task scheduling is performed by a foreign scheduler. In the former environment, it is possible for the implementation of

Asynchronous Transfer of Control to access the saved state of a suspended task and to force the task to resume operation at a location other than that at which the task was last executing. In the second class of systems, the saved execution state of a task is generally not available to the Ada run-time system, so some alternative mechanism for initiating ATC must be developed.

For systems in which the Ada run-time system has control over task context switching, the saved state of an aborted task can be modified such that when the task resumes execution it will begin executing at a routine that initiates the asynchronous transfer.

For systems in which scheduling is performed by an operating system or executive that supports inter-task signalling (e.g. POSIX), the propagation of an abort will be initiated by sending the aborted task a signal that is reserved for this purpose. The signal handler will initiate the asynchronous transfer.

Many existing real-time executives do not provide a mechanism for inter-task signaling. Examples of such executives include Intel’s iRMK, Ready System’s VRTX/ARTX, and TeleSoft’s TeleAda-Exec. Software Components Group’s pSOS+ does provide inter-task signaling, but upon completion of a signal handler there is no facility to resume execution of the signaled task at other than the location where it was executing when the signal was delivered. For all of these systems, and other similar systems, the only feasible approach to initiating an ATC is to poll at specific points within the run-time system for pending ATC requests. This effectively renders the ATC facility useless for such systems, since there is no guarantee that the abortable final part will ever reach a run-time call. TeleSoft believes that these systems represent an important class of targets for Ada execution.

Given the variations in the available notification mechanisms, the implementation will require the introduction of a new target-dependent interface to permit task-to-task signaling of pending aborts. Additionally, polling points must be optionally included in to the run-time system to facilitate those targets that can not support inter-task signaling. The one-time cost of defining the target-dependent notification mechanism, and adding the necessary notification to the processing of all event selection routines, is estimated to take approximately one person-month. This estimate takes into consideration the considerable documentation requirement for introduction of new TD-RSP interfaces. The recurring cost of implementing a notification mechanism will vary widely by target.

6.1.3.4 Abandonment of the Abortable Final Part

Once the task executing the abortable final part receives notification of a pending abort action (via one of the mechanisms described in section 6.2.3.3) it must initiate the abandonment of the abortable sequence of statements. The exact mechanism used depends on the model selected for finalization.

If the propagation model is used, control is transferred to the abort propagation routine as soon as the pending abort operation is detected. All necessary cleanup is performed automatically as the abort propagates through any and all scopes that contain finalization handlers. The abort propagates back to the handler inserted by the compiler for the abortable sequence of statements, and the task resumes execution at the point of the handler. The loop statement is not exited, so the select statement will be executed a second time. The select header will have been updated to reflect that one of the event alternatives has already been selected, so the second execution of the run-time service routine that evaluates the select alternatives will immediately return the number of the selected alternative. The execution-time cost of propagating the abort can be estimated by measuring the cost of exception propagation.

If the long-jump model is used, the finalization handlers for any enclosed scopes must be executed, followed by the execution of a longjmp back to the setjmp location that preceded the case statement for the selective entry call (see section 6.2.2). The mechanism used by the run-time routine that detects a pending abort to locate the appropriate finalization routines remains as an open issue for the design of finalization. Assuming this issue can be resolved, we anticipate that the long-jump model will execute faster than the propagation model by eliminating the need to unwind the stack as scopes are exited. The cost of saving and restoring the general purpose registers on a 20 Mhz 68020 processor is approximately 10 uSec, plus another 30 uSec if the floating point processor registers have been used.

The implementation cost for abandonment depends on the model selected for finalization. The propagation model is estimated to require one person-month to implement. The long-jump model is easily implemented for targets that already support setjmp/longjmp, but will require an estimated 2 person-weeks for other targets. Since both mechanisms are target-dependent, this is a recurring cost.

6.2 Execution-Time Costs

The total execution time for effecting an asynchronous transfer of control will include the execution time for: a selective entry call, the selection of one of the event alternatives, the signaling of the aborted task, the execution of the appropriate finalization operations (if any), and the return to the original execution point for the select (either via propagation or longjmp). We do not yet have measurements for all but the first of these costs; however we anticipate that the total execution time will be large.

7.0 Conclusions

7.1 Protected Records

We believe that protected records, although moderately costly to implement within the compiler, provide significant functionality with reasonable performance, at a reasonable implementation expense. Performance on some systems would be greatly enhanced by replacing the non-blocking, priority-ceiling, exclusion mechanism with a different mechanism. We are pleased that the semantic requirement for a non-blocking exclusion mechanism has been moved from the core language to an annex.

7.2 Requeue

In our initial implementation of protected records (in which entry bodies were directly executed in response to an entry call) Requeue proved very cumbersome to implement. Our subsequent redesign of protected records entry calls has greatly simplified the implementation of Requeue. We now believe that the implementation cost for requeue of a protected record entry is small, and that the added expressive power of the feature justifies its inclusion. To date we have not evaluated the cost of implementing requeue for task entries.

7.3 Selective Entry Call

The effort to implement selective entry call is comparable to that of Ada 83 selective wait. However, the run-time execution costs appear to be higher. We are concerned that the overhead associated with this feature may be a limiting factor in its acceptance by the real-time community.

7.4 Asynchronous Transfer of Control

The implementation of asynchronous transfer of control within the run-time system appears expensive, particularly if the construct is allowed to be nested. Furthermore, for a large class of systems, the only implementation option is to poll for pending aborts - thus rendering the feature largely useless. Given that the feature is tightly integrated with selective entry call, the run-time cost of performing an asynchronous transfer will be high. We do not believe that the implementation cost is justified given the limited domain in which the feature can be used. We would encourage the development of some alternate mechanism to achieve the same benefit.

A.Test Execution Logs

All tests were compiled with checks on and with the maximum optimization option selected. The test programs were executed on a Motorola MVEM135 single board computer consisting of a 20 Mhz 68020 processor with one megabyte of zero wait-state RAM.

```
135Bug>go 5000
Effective address: 00005000

Test Name:   PR000001                     Class Name:  Protected Records
CPU Time:       14.9  microseconds
Wall Time:      14.9  microseconds.         Iteration Count:   2048
Test Description:
 Minimum protected record procedure call.
 Protected Record inside procedure
 no PR entries


135Bug>go 5000
Effective address: 00005000

Test Name:   PR000002                     Class Name:  Protected Records
CPU Time:       13.6  microseconds
Wall Time:      13.6  microseconds.         Iteration Count:   2048
Test Description:
 Minimum protected record function call.
 Protected Record inside procedure
 no PR entries


135Bug>go 5000
Effective address: 00005000

Test Name:   PR000003                     Class Name:  Protected Records
CPU Time:       33.9  microseconds
Wall Time:      33.9  microseconds.         Iteration Count:   1024
Test Description:
 Minimum protected record entry call.
 Protected Record inside procedure
 PR has only one entry, always open


135Bug>go 5000
Effective address: 00005000

Test Name:   PR000004                     Class Name:  Protected Records
CPU Time:       39.8  microseconds
Wall Time:      39.8  microseconds.         Iteration Count:   256
Test Description:
 Minimum protected record entry call.
 Protected Record inside procedure
 PR has two entries, always open
```

```
135Bug>go 5000
Effective address: 00005000

Test Name:   PR000005                     Class Name:  Protected Records
CPU Time:       40.7  microseconds
Wall Time:      40.7  microseconds.         Iteration Count:   64
Test Description:
 Protected record entry call.
 Protected Record inside procedure
 PR has ten entries, always open


135Bug>go 5000
Effective address: 00005000

Test Name:   PR000006                     Class Name:  Protected Records
CPU Time:      249.2  microseconds
Wall Time:     249.2  microseconds.         Iteration Count:   128
Test Description:
 Selective protected record entry call.
 Protected Record inside procedure
 PR has two entries, alternately open.


135Bug>go 5000
Effective address: 00005000

Test Name:   PR000007                     Class Name:  Protected Records
CPU Time:      294.5  microseconds
Wall Time:     294.5  microseconds.         Iteration Count:   128
Test Description:
 Selective protected record entry call.
 Protected Record inside procedure
 PR has ten entries, first entry always open.


135Bug>g 5000
Effective address: 00005000

Test Name:   PR000008                     Class Name:  Protected Records
CPU Time:     1723.4  microseconds
Wall Time:    1723.4  microseconds.         Iteration Count:   16
Test Description:
 Selective protected record entry call.
 Protected Record inside procedure
 PR has ten entries, Last entry always open.


135Bug>go 5000
Effective address: 00005000

Test Name:   T000009                        Class Name:  Tasking
CPU Time:      207.1  microseconds
```

```
Wall Time:     207.1  microseconds.        Iteration Count:   128
Test Description:
 Task selective entry call and return time
 Two tasks active, one entry per task
 selective entry call statement


135Bug>go 5000
Effective address: 00005000

Test Name:   T000010                       Class Name:  Tasking
CPU Time:      307.5  microseconds
Wall Time:     307.5  microseconds.        Iteration Count:   128
Test Description:
 Task selective entry call and return time
 Ten tasks active, one entry per task
 First task will accept calls


135Bug>go 5000
Effective address: 00005000

Test Name:   T000011                       Class Name:  Tasking
CPU Time:     1348.2  microseconds
Wall Time:    1348.2  microseconds.        Iteration Count:   16
Test Description:
 Task selective entry call and return time
 Ten tasks active, one entry per task
 Last task will accept calls
```

B.Test Sources

The tests used to evaluate the performance of protected records and selective entry call were derived from the PIWG90 T (tasking) tests. The source for the modified tests is included here for reference.