PART/GNARL Interface Definition

Version 1.28

E.W. Giering, III
T.P. Baker
Florida State University
Tallahassee, FL 32306

1993/12/11 00:40:11

Copyright ©1993 by the Florida State University. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage and credit to the source is given. Abstracting with credit is permitted. Photocopying and electronic data transmission of this document are permitted for private use. Unrestricted use of the Ada source code of the interface specifications, including but not restricted to their implementation as commercial products, is permitted. All other rights are reserved.

This document describes the interface between the code generated by the Gnu NYU Ada Translator (GNAT) and the tasking support portion of the Gnu Ada Runtime Library (GNARL). The present version is a preliminary draft, for internal use of the GNAT and GNARL working groups, and for review by interested parties. We expect to make significant changes in both the content and scope of the document. Therefore, we warn against basing any product development on this version of this document.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Contents

PAGE

Contents ii

Section 1: Introduction 1
1.1 Purpose 1
1.2 Scope 1
1.3 Documentation Conventions 2
1.4 Principles 3
1.5 Dependences of the TRTS on the Compiler 4
1.6 Dependences of the Compiler on the TRTS 5
1.7 Dependences of the TRTS on the Target Machine 5
1.7.1 Initial Task State 5
1.7.2 Pre-Call State 7
1.7.3 Task Storage Size Type 7
1.7.4 Machine Exceptions Type 7
1.7.5 Interrupt ID Type 7
1.7.6 Interrupt Info Type 7
1.7.7 Error Information Type 7
1.8 RTS Service Call Conventions 7
1.9 Uninterpreted Data in the TRTS 8
1.9.1 RTS Responsibilities 8
1.9.2 RTS Notes 8
1.9.3 Compiler Responsibilities 8
1.9.4 Compiler Notes 9

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

25 1.10 Executing Compiler-Generated Code 9
26 1.11 Package Structure 9
27 1.12 Examples of GNARLI Use 10

28 Section 2: Error Checking at the TRTS Interface 11
29 2.1 RTS Responsibilities 11
30 2.2 RTS Notes 11
31 2.3 Compiler Responsibilities 11
32 2.4 Compiler Notes 11

33 Section 3: Predefined Packages 13
34 3.1 System Package 13
35 3.1.1 RTS Responsibilities 13
36 3.1.2 RTS Notes 13
37 3.1.3 Compiler Responsibilities 13
38 3.2 Standard Package 14
39 3.2.1 RTS Responsibilities 14
40 3.2.2 RTS Notes 14
41 3.2.3 Compiler Responsibilities 14

42 Section 4: Representing Tasks 15
43 4.1 Package Specification 15
44 4.2 Representing Tasks to the TRTS 15
45 4.3 RTS Responsibilities 15
46 4.4 RTS Notes 16
47 4.5 Compiler Responsibilities 16
48 4.6 Compiler Notes 16

49 Section 5: Abortion 19

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

50 5.1 Package Specification 19
51 5.2 ATC Levels 19
52 5.2.1 RTS Responsibilities 19
53 5.2.2 RTS Notes 19
54 5.3 The Abortion Exception 20
55 5.3.1 RTS Responsibilities 20
56 5.3.2 RTS Notes 20
57 5.3.3 Compiler Responsibilities 20
58 5.4 Deferring Abortion 21
59 5.4.1 RTS Responsibilities 21
60 5.4.2 RTS Notes 21
61 5.4.3 Compiler Responsibilities 21
62 5.4.4 Compiler Notes 21
63 5.5 Aborting Tasks 22
64 5.5.1 RTS Responsibilities 22
65 5.5.2 Compiler Responsibilities 22
66 5.5.3 Compiler Notes 22

67 Section 6: Task Creation and Activation 23
68 6.1 Package Specification 23
69 6.2 Task Masters 24
70 6.2.1 RTS Responsibilities 24
71 6.2.2 Compiler Responsibilities 24
72 6.2.3 Compiler Notes 24
73 6.3 Current Master 25
74 6.3.1 RTS Responsibilities 25
75 6.3.2 Compiler Notes 25

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

76 6.4 Master Entry 25
77 6.4.1 RTS Responsibilities 25
78 6.4.2 RTS Notes 25
79 6.4.3 Compiler Responsibilities 26
80 6.4.4 Compiler Notes 26
81 6.5 Master Exit 26
82 6.5.1 RTS Responsibilities 26
83 6.5.2 Compiler Responsibilities 27
84 6.5.3 Compiler Notes 27
85 6.5.4 Example of Use of Masters 30
86 6.6 Activation Chains 31
87 6.6.1 RTS Responsibilities 31
88 6.6.2 RTS Notes 31
89 6.6.3 Compiler Responsibilities 32
90 6.6.4 Compiler Notes 32
91 6.7 Task Creation 32
92 6.7.1 RTS Responsibilities 32
93 6.7.2 RTS Notes 33
94 6.7.3 Compiler Responsibilities 33
95 6.7.4 Compiler Notes 34
96 6.7.5 Rationale 34
97 6.8 Task Activation 35
98 6.8.1 RTS Responsibilities 35
99 6.8.2 Compiler Responsibilities 35
100 6.8.3 Compiler Notes 35
101 6.9 Expunction of Unactivated Tasks 36
102 6.9.1 RTS Responsibilities 36

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

103 6.9.2 Compiler Responsibilities 36
104 6.9.3 Compiler Notes 36
105 6.9.4 Example of Task Allocation and Activation 37
106 6.9.5 Example of Activation and Creation 39
107 6.10 Activation Completion 40
108 6.10.1 RTS Responsibilities 40
109 6.10.2 Compiler Responsibilities 40
110 6.10.3 Compiler Notes 41
111 6.11 Task Completion 41
112 6.11.1 RTS Responsibilities 41
113 6.11.2 RTS Notes 41
114 6.11.3 Compiler Responsibilities 42
115 6.11.4 Compiler Notes 42
116 6.12 Task Termination 42
117 6.12.1 RTS Responsibilities 42
118 6.12.2 RTS Notes 42
119 6.12.3 Compiler Responsibilities 42
120 6.12.4 Example of Task Activation, Completion, and Termination . . 43
121 6.13 Terminated Attribute 43
122 6.13.1 RTS Responsibilities 43
123 6.13.2 Compiler Responsibilities 43
124 6.13.3 Compiler Notes 43
125 6.14 Environment Tasks 44
126 6.14.1 RTS Responsibilities 44
127 6.14.2 RTS Notes 44
128 6.14.3 Compiler Responsibilities 44
129 6.14.4 Compiler Notes 44

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

130 Section 7: Delays 45
131 7.1 Package Specifications 45
132 7.1.1 RTS Responsibilities 45
133 7.1.2 Compiler Notes 45
134 7.1.3 Example of a Simple Relative Delay Statement 46
135 7.1.4 Example of a Simple Absolute Delay Statement 46
136 7.1.5 Example of an Asynchronous Delay Statement 46

137 Section 8: Rendezvous 49
138 8.1 Package Specification 49
139 8.2 Entry Indices 50
140 8.2.1 RTS Responsibilities 50
141 8.2.2 Compiler Responsibilities 50
142 8.2.3 Compiler Notes 50
143 8.3 Entry Calls 50
144 8.3.1 RTS Responsibilities 51
145 8.3.2 Compiler Responsibilities 51
146 8.3.3 Compiler Notes 52
147 8.3.4 Example of a Conditional Entry Call 52
148 8.4 Simple Entry Calls 52
149 8.4.1 RTS Responsibilities 52
150 8.5 Completing Asynchronous Task Entry Calls 53
151 8.5.1 RTS Responsibilities 53
152 8.5.2 Compiler Responsibilities 53
153 8.5.3 Compiler Notes 53
154 8.5.4 Example of an Asynchronous Task Entry Call 54
155 8.5.5 Example of a Timed Task Entry Call 55

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

156 8.6 Requeuing an Entry Call 55
157 8.6.1 RTS Responsibilities 55
158 8.6.2 Compiler Responsibilities 56
159 8.6.3 Compiler Notes 56
160 8.7 Selective Waits 56
161 8.7.1 RTS Responsibilities 56
162 8.7.2 Compiler Responsibilities 58
163 8.7.3 Compiler Notes 58
164 8.7.4 Example of a Selective Wait Statement 59
165 8.7.5 Example of a Selective Wait Statement With Delay Alternatives 61
166 8.8 Simple Accept Statements 63
167 8.8.1 RTS Responsibilities 63
168 8.8.2 Compiler Notes 63
169 8.9 Trivial Accepts 63
170 8.9.1 RTS Responsibilities 63
171 8.9.2 Compiler Notes 64
172 8.10 Completing Rendezvous 64
173 8.10.1 RTS Responsibilities 64
174 8.10.2 RTS Notes 64
175 8.10.3 Compiler Responsibilities 65
176 8.10.4 Compiler Notes 65
177 8.11 Count Attribute 65
178 8.11.1 RTS Responsibilities 65
179 8.11.2 Compiler Responsibilities 65
180 8.11.3 Compiler Notes 65
181 8.12 Callable Attribute 66
182 8.12.1 RTS Responsibilities 66

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

183 8.12.2 Compiler Responsibilities 66
184 8.12.3 Compiler Notes 66

185 Section 9: Protected Objects 67
186 9.1 Description 67
187 9.2 Representation of Protected Objects 68
188 9.2.1 RTS Responsibilities 68
189 9.2.2 RTS Notes 68
190 9.2.3 Compiler Notes 68
191 9.3 Entry Indices 68
192 9.3.1 RTS Responsibilities 68
193 9.3.2 Compiler Responsibilities 69
194 9.3.3 Compiler Notes 69
195 9.4 Parameter Passing 69
196 9.4.1 Compiler Notes 69
197 9.5 Initializing and Finalizing Protected Objects 69
198 9.5.1 RTS Responsibilities 69
199 9.5.2 Compiler Responsibilities 70
200 9.5.3 Compiler Notes 70
201 9.6 Initiating and Completing Protected Actions 70
202 9.6.1 RTS Responsibilities 70
203 9.6.2 RTS Notes 71
204 9.6.3 Compiler Responsibilities 71
205 9.6.4 Compiler Notes 71
206 9.7 Pending Entry Calls 72
207 9.7.1 RTS Responsibilities 72
208 9.7.2 RTS Notes 73

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

209 9.7.3 Compiler Responsibilities 73
210 9.7.4 Compiler Notes 73
211 9.8 Completing Asynchronous Protected Entry Calls 73
212 9.8.1 RTS Responsibilities 73
213 9.8.2 Compiler Responsibilities 74
214 9.8.3 Compiler Notes 74
215 9.8.4 Example of an Asynchronous Protected Entry Call 74
216 9.8.5 Example of a Timed Protected Entry Call 75
217 9.9 Servicing Entry Calls 76
218 9.9.1 RTS Responsibilities 76
219 9.9.2 Compiler Responsibilities 77
220 9.9.3 Compiler Notes 77
221 9.10 Completing a Protected Entry Call 78
222 9.10.1 RTS Responsibilities 78
223 9.10.2 Compiler Responsibilities 78
224 9.10.3 Compiler Notes 78
225 9.11 Waiting for a Protected Entry Call 78
226 9.11.1 RTS Responsibilities 78
227 9.11.2 Compiler Notes 78
228 9.12 Count Attribute 79
229 9.12.1 RTS Responsibilities 79
230 9.12.2 Compiler Responsibilities 79
231 9.12.3 Compiler Notes 79
232 9.13 Propagating Exceptions 79
233 9.13.1 RTS Responsibilities 79
234 9.13.2 Compiler Responsibilities 80
235 9.13.3 Compiler Notes 80

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

236 9.14 Example of a Protected Object 80
237 9.15 Requeuing an Entry Call 85
238 9.15.1 RTS Responsibilities 85
239 9.15.2 Compiler Responsibilities 85
240 9.15.3 Compiler Notes 85
241 9.15.4 Example of the Requeue of a Protected Entry Call 86
242 9.16 Rationale 89
243 9.16.1 Protected Actions—The Eggshell Model 89
244 9.16.2 Asynchronous Transfer of Control 90
245 9.16.3 Execution Models 90

246 Section 10: Runtime Support for Storage Management 93
247 10.1 Package Specification 93
248 10.2 RTS Responsibilities 93
249 10.3 RTS Notes 93
250 10.4 Compiler Notes 95
251 10.5 Rationale 95

252 Section 11: Compiler Support for Exceptions 97
253 11.1 Package Specification 97
254 11.2 Compiler Responsibilities 97
255 11.3 Compiler Notes 98
256 11.4 RTS Responsibilities 99
257 11.5 RTS Notes 99

258 Section 12: Acknowledgements 101

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 1: Introduction

This document describes the interface between the code generated by the Gnu NYU Ada Translator (GNAT) and the tasking support portion of the Gnu Ada Runtime Library (GNARL). We call it the GNARL Interface (GNARLI, for short). This section describes why this document was produced, what it contains, and how it is intended to be used.

The interface between the GNARL and an underlying operating system or executive is defined in a separate document.

1.1 Purpose

The purpose of this document is to serve as a contract between the developers of an Ada compiler (GNAT) and the portion of an Ada runtime system (RTS) that supports tasking (the TRTS, for short).

1.2 Scope

The scope of the GNARLI is limited to the runtime system services that are required to implement the proposed Ada 9X [1] tasking semantics, and certain other runtime system services which may not be directly related to tasking but whose implementation we have not been able to separate cleanly from that of tasking.

The scope includes only RTS services that are invoked implicitly by compiler-generated code. In particular, it does not include those interfaces which are already defined by the Ada language, such as `Calendar` and `Text_IO`.

At present, the GNARLI covers the following services:

1. Task lifetime control
   (a) Task creation
   (b) Task activation
   (c) Task termination
   (d) `Terminated` attribute
2. Rendezvous management
   (a) Entry calls
   (b) **`Select`** and **`accept`** statements

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

(c) `Callable` attribute

(d) `Count` attribute

3. Protected Objects

   (a) Protected Procedures and Functions

   (b) Protected Entry Calls

   (c) Count attribute

4. Abortion

5. Delays

6. Storage Management

Besides the Ada 9X features, we need to include support for interrupt entries.

1.3 Documentation Conventions

Subsections marked "RTS Responsibilities" describe features which an implementation of the GNARLI is required to provide. These are the features which the GNARLI client can count on.

Subsections marked "Compiler Responsibilities" describe invariants that the compiler or the code that it generates must maintain. Failure to do this results in unpredictable or erroneous behavior.

Subsections marked "RTS Notes" provides rationale for TRTS features, advice on the implementation the GNARLI, and other information on the intent of the interface. Text in these sections is not normative.

Subsections marked "Compiler Notes" provide advice on how GNARLI can be used to implement Ada features. Text in these sections is not normative.

All non-normative text is marked by grey areas in the margins to the left and right of the text. For example:

This paragraph is not normative.

A number of typographic conventions are also used to indicate the intended use of the associated text, as shown in the following list.

— `Ada program text`

— **`ada keywords`**

— *Definitions*

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

1.4 Principles

The intent of the GNARLI is to specify what the developer of a compiler needs to know about the TRTS and what the developer or tailorer of the TRTS needs to know about the compiler and the rest of the RTS. This involves specifying a clear division of duties.

There is clearly no single "best" way to divide these duties. The starting point for the GNARLI was CARTS [2], which was derived from MRTSI [5]. An attempt has been made to follow the CARTS interface faithfully, except where there is a very strong justification for difference. The main justifications for differences are:

1. Ada 9X requires new or different functionality.
2. A CARTS feature represents an extension to the Ada 83 tasking model, and we choose not to support that extension.
3. Design goals of the GNAT compiler conflict with some aspect of the CARTS interface.
4. CARTS operations provided for to support Ada 83 are redundant to operations needed to support Ada 9X.

In the design of extensions to support Ada 9X, an attempt has been made to follow the original principles of the CARTS and MRTSI interfaces, which were:

1. Necessity. Features should all be necessary to support standard Ada 9X for real-time applications.
2. Separation. The interface should cleanly separate the functions of the TRTS from those of the compiler-generated code and the rest of the RTS.
3. Information hiding. The interface should minimize the knowledge that the TRTS needs to have about the generated code and the rest of the RTS, and *vice versa*.
4. Implementation freedom. If possible, the interface should not force one particular implementation to the exclusion of another equally viable one.
5. Target-independence. Features should be specified in a way that is not specific to one machine or underlying operating system; that is, the interface should admit efficient implementations on a wide spectrum of machines.
6. Robustness. Features should favor the potential ability of the TRTS to detect and react gracefully to errors (if the implementor chooses), if this can be done without undue overhead.

These principles have several consequences. For example, a consequence of (1) and (2) is that none of the GNARLI features is intended to be invoked directly by an application programmer (though an implementation is not forbidden to export them

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

to that level). Another consequence of (1) is that the GNARLI provides no support for debugging. An important consequence of (3) is that the GNARLI is procedural. The executive and compiler-generated code should not share access to any data structures—they communicate only through operations with explicit parameters.

1.5 Dependences of the TRTS on the Compiler

This document primarily describes the dependences of the compiler on the TRTS. Unfortunately, there are also some ways in which the TRTS implementation will (unavoidably) depend on the compiler. These include:

1. The representation of several types of data, including the predefined types `Integer`, `Boolean`, `Duration`, and `System.ADDRESS`, as well as all the types defined in the GNARLI.

2. Conventions for calling subprograms, passing parameters, and returning function values.

3. The representation of exceptions as data, and the mechanism of raising an exception (see Section 11).

4. The representation of a special exception for abortion. This exception cannot be caught by a **`when others`** exception handler. Since this exception is defined by the TRTS, it would not usually be visible to the application, and could not be caught by it.

5. The representation of task bodies, which must be compiled as procedures (see Section 6.7).

We assume the TRTS is written in Ada. Thus, most compiler-dependences will not be visible in the TRTS source code. This is the case with item 1, and, for the most part, item 2. However, there will remain some small parts of the TRTS implementation that will require specific knowledge of the compiler's conventions for subprogram calls. The required capability is described in this document as the TRTS calling a procedure "within the context of" a task (see Section 6.7). Examples of situations where the TRTS may need to do this are raising an exception in a blocked task, notifying a task of a hardware-detected exception, and (possibly) causing a task to abort itself. Sufficient information for the TRTS to be able to do this must be provided to the TRTS implementor by the compiler developer, and some (small, encapsulated) portion of the TRTS implementation will need to depend on it.

Item 3 shows up in TRTS interface via the exception ID parameter of `Exceptional_End_Rendezvous`, `Exceptional_Complete_Entry_Body`, `Exceptional_Cancel_Task_Entry_Call`, and `Exceptional_Cancel_Protected_Entry_Call`, whose type must be visible to the TRTS.

Item 4 shows up in the TRTS interface in the `Abortion` exception in the `System.Abortion` package (see Section 5). This document assumes that the compiler

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

provides a `No_Others` pragma that takes an exception as an argument, and which prevents that exception from being caught by a **`when others`** handler.

The remaining dependences of the TRTS on the compiler are not visible in the TRTS interface, but some will be manifest in the form of small segments of compiler-dependent code within the TRTS.

1.6 Dependences of the Compiler on the TRTS

The `System.Machine_Specifics` package contains declarations of types that represent conventions about the use of the target machine architecture on which the compiler, TRTS, and other runtime system components may depend. The structure, size, representation, and contents of these types would ordinarily be determined by the compiler developer, based on what is natural and efficient for the specific target machine. The TRTS should be coded in a modular fashion, so that portions which depend on the definitions of these types can be modified easily. Moreover, for a given target machine, it should be possible to fix a set of definitions for these types which can be common across compilers for that machine.

1.7 Dependences of the TRTS on the Target Machine

```
package System.Machine_Specifics is
   type Init_State is implementation-defined;
   type Pre_Call_State is private;
   type Task_Storage_Size is range implementation-defined;
   type Machine_Exceptions is range implementation-defined;
   type Interrupt_ID is range implementation-defined;
   type Interrupt_Info is implementation-defined;
   type Error_Information is implementation-defined;
private
  ... implementation-defined
end System.Machine_Specifics;
```

1.7.1 Initial Task State

The `Init_State` type shall be used to represent the initial hardware state of a task, when it starts executing. It contains information needed for initialization of a task. It must allow the code for the body of the task to be executed within a new thread of control, and allow the Ada environment for this code (e.g. uplevel references) to be established.

1.7.1.1 GNARL Implementation Notes

The specific details below are preliminary specifications for the GNAT Ada compiler and GNARL TRTS for the Sun SPARC[1] architecture.

[1] Sun is a trademark of Sun Microsystems, Inc.; SPARC and SPARCstation are trademarks of SPARC International, Inc.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Ada 9X provides subprogram access values that allow the referenced subprogram to be called in the proper environment. One way to implement `Init_State` values is to compile task bodies as procedures and use a procedure access type to these subprograms as the implementation of `Init_State`. A task could then call this procedure through an `Init_State` value at activation.

Whether this works or not depends on the implementation of uplevel references. The Ada 9X reference manual 2.0 3.10(7.a) points out that for implementations which use a global display, it will generally be possible to call a subprogram using only the address of the subprogram code, since the environment needed for uplevel references will be in the display at the point of call. On such implementations, it may also be possible for the TRTS to call the task body procedure from the new thread of control using only the address of its code, but this will depend on how "global" accesses are done. If some context is needed to access the global display (e.g. there is a distinguished register pointing to global data), then `Init_State` may have to be defined to allow this context to be established in the new thread.

In systems that use static chains for uplevel references, it will usually be desirable to include the address of the statically enclosing frame with the code address as part of a subprogram access value[2]. In this case, the TRTS can use this value to call the task procedure in the new thread of control using the normal Ada calling mechanism.

The GNAT compiler does implement access to subprogram in this way, so `Init_State` will be implemented as an access to procedure type in GNARL:

```
private
   type Init_State is access procedure(Arg: System.Address);
   ... implementation-defined
end System.Machine_Specifics;
```

Note that `Init_State` is a global procedure access type. Ada accessibility rules only allow access values referring to library level procedures to be assigned to variables of such a type. This is to prevent dangling references in the form of access variables referring to procedures whose static environment no longer exists. Using access values to execute task body procedures depends on those procedures being nested at the level of the task bodies that they represent, so in general it will violate Ada semantics to assign access values referring to these procedures to variables of type `Init_State`. Note that this conservative compile-time check is not needed in this case, since Ada task termination processing assures that the environment of a task will exist until the task is terminated. There must therefore be some means of circumventing accessibility checks to allow arbitrary procedure pointers to be assigned to variables of `Init_State`; for GNAT, this consists of using `Unchecked_Conversion` to convert procedure access values to `Init_State` values.

[2] Even in this case, it is possible to do without a static link in a procedure access value. A subprogram can find the enclosing frame from the current frame by traversing the static links to the outermost frame and then traversing them back down for the requisite number of nesting levels. However, it is unlikely that any implementation would choose to add this much distributed overhead to subprogram calls to reduce the size of access to subprogram values by one address.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

1.7.2 Pre-Call State

The `Pre_Call_State` type shall be the portion of the hardware state of a task that is not preserved by the subprogram-calling convention. For example, this might be an array of integers, representing the values of certain registers. (No relationship is specified between this type and the `Init_State` type.)

1.7.3 Task Storage Size Type

The `Task_Storage_Size` type shall be used to specify task storage-sizes. This shall be an integer type.

1.7.4 Machine Exceptions Type

The `Machine_Exceptions` type shall be used to identify a hardware-detected exception when notifying the exception-recovery system in the rest of the RTS. This might be an enumerated or integer type.

1.7.5 Interrupt ID Type

The `Interrupt_ID` type shall be used to identify hardware interrupts and virtual interrupts.

Need to decide whether to merge POSIX signals with interrupts, or provide a special type and package(s) for signals.

1.7.6 Interrupt Info Type

The `Interrupt_Info` type shall be used to specify information (if any) that is passed from a hardware interrupt to an interrupt handler procedure.

1.7.7 Error Information Type

The `Error_Information` type shall be used to provide any additional information associated with a hardware-detected error. For example, this might be a record type.

1.8 RTS Service Call Conventions

TRTS actions are assumed to be initiated by one of the following:

1. Request by an Ada task (including the environment task), which is formally termed a "TRTS call". From the point of view of the calling task, these are indistinguishable from ordinary subprogram calls.

2. A hardware interrupt (or trap). Examples of this include completion of an I/O operation, and expiration of a delay.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

3. A software signal. These are similar to interrupts or traps, but are generated by an underlying operating system. They may serve as an interface to interrupts and traps, or exist together with them.

Only TRTS calls are visible at the TRTS interface. Interrupt, trap, and signal handlers are installed by the TRTS, and are therefore entirely internal to the TRTS implementation.

The syntax of Ada subprograms is used to name the TRTS calls and specify their parameters.

The TRTS is assumed to be written in Ada. Therefore, the compiler should be able to generate in-line code for those TRTS calls where it makes sense. Ideally, the compiler may perform optimizations on the expanded TRTS code.

The compiler is expected to provide correct arguments to the TRTS. In particular, if there is a constraint on a parameter to a TRTS call the compiler is expected to perform any necessary checking and raising of exceptions, or ensure that the constraint is always satisfied.

The TRTS may (but need not) check for bad arguments. If such an error is detected, the TRTS is free to choose an appropriate response.

1.9 Uninterpreted Data in the TRTS

Some TRTS operations require the TRTS to transfer *uninterpreted data* between the compiler-generated code of one task and that of another.

1.9.1 RTS Responsibilities

When uninterpreted data must be passed between tasks, the TRTS shall provide a means of transferring a a single value of type `System.Address` from one task to the other. The TRTS shall provide the means for task identification and synchronization necessary for this transfer, as documented under the individual operations.

1.9.2 RTS Notes

An example of where uninterpreted data is needed is task activation. The value of these discriminants are known to the task's activator, but must be available within the task itself. The TRTS must perform this transfer, since it mediates communication between tasks. However, the TRTS itself does not need to access the discriminants, so the TRTS simply accepts uninterpreted data when creating a task and later provides this data to the new task.

1.9.3 Compiler Responsibilities

Exactly how the compiler-generated code uses a `System.Address` value specifying uninterpreted data is up to the compiler.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

1.9.4 Compiler Notes

Compiler generated code should assure that the `System.Address` parameter can be used to access the appropriate uninterpreted data. All tasks are assumed to share a common virtual address space, so this address can be used as a pointer to a data block, allowing access to arbitrary data.

1.10 Executing Compiler-Generated Code

The TRTS effects the execution of compiler generated code in complex ways. This is more true of Ada 9X than it was with Ada 83, with the addition of complex control structures such as asynchronous **`select`** statements and protected objects. One way to allow the TRTS to control execution is to pass pointers to compiler-generated subprograms into the TRTS. These subprograms execute fragments of the structures whose execution must be controlled by the TRTS. The TRTS can then call these subprograms (a *callback* to compiler-generated code) as appropriate.

Another way for the TRTS to control execution is through embedded calls to the TRTS within compiler-generated code. This gives the TRTS the opportunity to block the task or communicate with the compiler-generated code through parameters. This only allows for local control of execution; the influence of the TRTS is to code immediately preceding or following the call, in general requiring support from additional compiler-generated code. This makes for a more complex interface and for more complex compiler-generated code to use it. However, it has the advantage of greater isolation of the TRTS and the compiler-generated code, allowing greater protection between the two, and requiring less protection within the TRTS for callbacks to less predictable application code. For these reasons, the GNARLI avoids the use of callbacks.

1.11 Package Structure

The TRTS interface is described via several Ada package specification, based on a natural functional division. These packages are:

1. `System.Task_IDs`—in Section 4.
2. `System.Abortion`—in Section 5.
3. `System.Task_Stages`—in Section 6.
4. `System.Delays`—in Section 7.
5. `System.Rendezvous`—in Section 8.
6. `System.Protected_Objects`—in Section 9.
7. `System.Storage_Allocation`—in Section 10.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Both the specification and body of any of the GNARLI packages may include additional declarations of types, objects, and operations required by the implementation of that package or other TRTS packages. The compiler must not rely on the existence (or nonexistence) of any such declarations.

The GNARLI package specifications are presented more fully in the following sections, with the intended semantics and rationale for each of their individual components. In some cases the description of the semantics refers to the Ada language definition.

1.12 Examples of GNARLI Use

To clarify the intended use of the GNARLI, this document provides examples of Ada tasks implemented using it. This is usually done by showing an Ada code fragment containing task structures, followed by a translation of this fragment into Ada code free of tasking features using GNARLI constructs. Please note that these examples are to be understood in a conceptual sense, for illustrative purposes only. In the actual implementation, all Ada constructs almost certainly would be translated directly to machine code by the compiler, rather than back into Ada code, and compiler optimizations are likely to result in some simplifications.

In fact, it is not always possible to provide the correct tasking semantics with Ada code. For this reason, some examples are expressed using extensions to the Ada language. These are defined as needed in the text accompanying the examples.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 2: Error Checking at the TRTS Interface

This section describes general rules regarding errors, and propagation of exceptions, at the TRTS interface.

2.1 RTS Responsibilities

TRTS calls shall not propagate exceptions back to the caller, except as explicitly required or permitted in this document.

The TRTS is responsible for causing certain correct TRTS calls to propagate a language-defined exception, under certain conditions. For example, an attempt to rendezvous with a terminated task causes `Tasking_Error` to be raised. This is considered part of the "normal" behavior of the TRTS call, and is specified as such in the description of the RTS responsibilities for the TRTS call.

In contrast, there are possible situations where a call to the TRTS violates the conventions of the TRTS interface. The compiler is trusted to produce correct TRTS calls, so such a violation is a compiler defect. The TRTS is *not required* to check for incorrect TRTS calls. The effect of an incorrect TRTS call is unspecified by the GNARLI.

Elaboration of the TRTS packages (specifications or bodies) shall raise no exceptions. In particular, elaboration of the TRTS packages shall not raise `Program_Error`, due to access before elaboration problems in the TRTS implementation.

2.2 RTS Notes

If a TRTS implementation does not check for incorrect TRTS calls, the result of an incorrect call may be to "crash" the system. The TRTS implementor may choose to check for incorrect calls, to preserve the integrity of the TRTS. If so, the choice of error recovery action may be made, *ad hoc*, by the TRTS implementor.

2.3 Compiler Responsibilities

The compiler is responsible for generating only correct TRTS calls. In particular, all parameter values must be valid, and satisfy any constraints imposed by this document.

2.4 Compiler Notes

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The compiler should generate TRTS calls in a fashion that if the TRTS performs its specified responsibilities the result will be consistent with the Ada language definition. However, the GNARLI implementation should not attempt to check for errors in implementing Ada semantics.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 3: Predefined Packages

This section describes aspects of the TRTS interface related to the packages `System` and `Standard`.

3.1 System Package

3.1.1 RTS Responsibilities

The TRTS implementation shall use the mechanism provided by the compiler implementor to ensure the constants and subtypes in the package `System` are consistent with the ranges of priorities and the clock resolution supported by the TRTS.

3.1.2 RTS Notes

The range of priorities used internally by the TRTS may be wider than that provided to the Ada user. However, it cannot be narrower, since tasks with distinct Ada priorities must have distinct priorities in the TRTS, if the Ada 9X Real-Time Annex is supported.

3.1.3 Compiler Responsibilities

The compiler implementor is responsible for providing a mechanism that permits the TRTS implementor to define the values and ranges of the following constants and subtypes, which are declared in the package `System`:

1. `Tick`, the nominal tick of `Calendar.Clock`.
2. `Priority`, the range of non-interrupt priorities which is supported.
3. `Any_Priority`, the full range of priorities supported.
4. `Interrupt_Priority`, the range of interrupt priorities supported.
5. `Default_Priority`.

The TRTS implementation must be able to define child packages of `System`. In particular, it must be able to define those required by the Ada 9X Annexes, including `System.Real_Time`. All TRTS packages are declared as children of package `System`.

The TRTS implementation packages (specifications as well as bodies) must be allowed to depend on the package `System`, as well as child packages defined by the RTS.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

3.2 Standard Package

3.2.1 RTS Responsibilities

The TRTS implementation shall use the mechanism provided by the compiler implementor to ensure the type `Duration` is consistent with the clock resolution supported by the TRTS.

3.2.2 RTS Notes

The representation of the standard `Duration` type is presumed to be built into the compiler. However, for efficient conversion, it is required that the time unit used in the implementation of delays be an integral multiple of `Duration'Small`.

3.2.3 Compiler Responsibilities

The compiler implementor is responsible for providing a mechanism that permits the TRTS implementor to tailor the value of `Duration'Small`, so that it is an integral multiple or divisor of the underlying clock tick.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 4: Representing Tasks

This package introduces the `Task_ID` type, which is used to represent tasks at the TRTS interface. It also provides some fundamental operations on task IDs.

4.1 Package Specification

```
package System.Task_IDs is
   type Task_ID is private;
   Null_Task: constant Task_ID;
   function Same_Task(L, R: Task_ID) return Boolean;
   function Self return Task_ID;
   subtype  Task_String is String(1..16);
   function Image(T: Task_ID) return Task_String;
private
   type Task_ID is ...;
end System.Task_IDs;
```

4.2 Representing Tasks to the TRTS

4.3 RTS Responsibilities

A value of the `Task_ID` type (a *task ID*) is *valid* if and only if it identifies some task that has been created by the `System.Task_Stages.Create_Task` and the corresponding task has not subsequently been expunged.

A `Task_ID` value that is returned by the `System.Task_Stages.Create_Task` operation represents the created task to the TRTS (here distinguished from an unelaborated task declaration, task type, or unevaluated task allocator) until the task is expunged.

Note that the value `Null_Task` is not a "valid" task ID, though this value shall be accepted by certain GNARLI operations with parameters of the `Task_ID` type.

A task shall be *expunged* by the TRTS (only) at the earliest of the following:

1. Exit from the task's immediate master (including termination of the master, if the master is a task).

2. A call to `Expunge_Unactivated_Tasks` for the activation list containing the task.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The `Same_Task` function shall return True if-and-only if the `L` and `R` parameters specify the same task or are both equal to `Null_Task`.

`Self` shall return a value of type `Task_ID` that uniquely identifies the calling task.

`Image` shall return a string that uniquely identifies the task identified by the `T` parameter, or that identifies the null task if `T` is equal to `Null_Task`. The effect is undefined if `T` is invalid and is not equal to `Null_Task`.

4.4 RTS Notes

The TRTS must be able to determine the identity of the currently executing Ada task on a processor (in the case of an interrupt handler, it also needs to be able to determine the identity of the interrupted task.) This operation must be implemented very efficiently, perhaps as in-line code, since it is likely to be used frequently. It generally requires some cooperation with the compiler. One solution is to reserve a global register to contain a reference to the currently executing task. A variation on this is to store the task ID at one end of the corresponding task's runtime stack, or to put a pointer to it at that location.

It is recommended that the TRTS implementation provide a mechanism for using the string returned by `Image` to determine the declaration of the corresponding task or task type. Since the `Image` function is only provided for debugging output, the mechanism need not be available at run time in the target environment. (See also the `Image` function for the `Exception_ID` type, Section 11.1.)

4.5 Compiler Responsibilities

For TRTS calls which require a valid task ID as parameter, the compiler is responsible for ensuring that it is valid.

4.6 Compiler Notes

A valid task ID represents a unique Ada task. However, there may be more than one task ID which represents a given task. For this reason, the `Same_Task` function is provided. This operation should always be used to determine if two `Task_ID` values represent the same task.

The compiler-generated code may need to access the task ID corresponding to the currently executing task. For this reason, the `Self` operation, already needed by the TRTS, is made visible in the interface.

When the name of a task type is used to refer to the object that designates the task currently executing the body of the task type (see [1], Section 9.1(15)), the compiler must find the corresponding `Task_ID` value. The function `Self` can be used directly, but it will generally be more efficient to call this function once whenever a task of the type in question starts to activate. This would probably be done by keeping the task

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

ID in a local variable of the task and translating the name of the task type within the body of the task to a reference to this local variable. This will work even if the reference to the task comes from some other task nested within it.

An task ID is no longer valid after return from a subprogram where the task is declared. It is the compiler's responsibility to raise `Program_Error` in functions that attempt to return local tasks, since tasks are return-by-reference types.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

73 *This side is intentionally blank.*

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 5: Abortion

The `System.Abortion` package provides operations to abort tasks or sequences of statements (for ATC).

5.1 Package Specification

```
with System.Task_IDs;
package System.Abortion is
   Max_ATC_Nesting: constant Natural:= implementation-defined;
   Abortion: exception;  -- pseudo-exception
     pragma No_Others(Abortion); -- cannot be caught by when others =>
   procedure Defer_Abortion;
   procedure Undefer_Abortion;
   type Task_List is array(Positive range <>) of Task_IDs.Task_ID;
   procedure Abort_Tasks(Tasks: Task_List);
end System.Abortion;
```

The description of this package is divided into several subsections.

5.2 ATC Levels

Asynchronous **select** statements can be nested, so that a task can have several entry calls pending at once. `Max_ATC_Nesting` is the maximum number of ATCs that can be pending on a given task at one time.

5.2.1 RTS Responsibilities

The TRTS shall allow a task to make a combined total of `Max_ATC_Nesting` calls to the `Task_Entry_Call` or `Protected_Entry_Call` operations that can be pending on a task simultaneously. The effect of attempting to queue more than `Max_ATC_Nesting` entry calls is undefined.

If `Max_ATC_Nesting` is zero, the implementation places no fixed limit on the number of entry calls that can be pending on a task at one time. The number of entry calls is limited only by the availability of the necessary TRTS resources in this case.

5.2.2 RTS Notes

In Ada 83, a task was always blocked when it made an entry call, so only one such call could be pending on a task at a time. This could be implemented by a single queue element, possibly part of a task control block. In contrast, an asynchronous **select** statement may be queued waiting for an entry while the calling task continues to

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

execute, and this can be repeated any number of times. This implies some scheme for dynamically allocating queue elements to allow a task to be on multiple queues, subject only to storage availability.

Allowing the implementation to limit the depth of nesting allows for the preallocation of all needed queue blocks at task creation time, avoiding the need to do dynamic allocation as part of the entry call mechanism. This preallocation could take the form of `Max_ATC_Nesting` queue elements allocation as part of a task control block.

5.3 The Abortion Exception

Task abortion requires the abortion of the sequence of statements of the task body; ATC requires the abortion of the sequence of statements comprising the abortable part. This interface uses a special exception to perform abortion.

5.3.1 RTS Responsibilities

The RTS will defer abortion and raise the `Abortion` exception asynchronously in a task whenever it is aborted or completes the triggering statement of an asynchronous select statement.

The `Abortion` exception is raised by various TRTS calls responsible for completing triggering statements, including `Complete_Rendezvous` (Section 8) and `Complete_Entry_Body` (Section 9).

Abortion will remain deferred until the abortable part or task being aborted is left.

5.3.2 RTS Notes

Task abortion and ATC both cause a task to stop its current execution no later than an abortion completion point. A task that has been aborted will not execute again once the abortion has taken effect, where ATC will resume execution after completing some sequences of statements. In both cases, all finalization for the completed structures must be performed.

This similarity makes it desirable to use the same mechanism for both features. This interface allows a task to be aborted as if it were the abortable part to some enclosing asynchronous select statement.

5.3.3 Compiler Responsibilities

The compiler must support a special kind of pseudo-exception, `Abortion`, which cannot be caught by a handler for **`others`**. A possible interface for this is indicated above by the pragma `No_Others`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

5.4 Deferring Abortion

The `Defer_Abortion` and `Undefer_Abortion` operations allow the effect of abortion on a task to be controlled.

5.4.1 RTS Responsibilities

`Defer_Abortion` shall defer the effects of abortion in the calling task. The TRTS shall keep track of all abortion deferrals, either by `Defer_Abortion` or by some other means (e.g. a call to `Lock`). It shall continue to defer abortion until all such deferrals are removed. Abortion itself also defers abortion, and as such must be completed by some TRTS call that undefers abortion.

`Undefer_Abortion` shall raise the `Abortion` exception if it corresponds to the earliest pending `Defer_Abortion` call made by this task, and it finds that abortion to an outer asynchronous call level or the whole task is pending. Otherwise it will remove the effects of the most recent call to `Defer_Abortion` made by the current task. Note that `Undefer_Abortion` leaves abortion deferred if it raises the `Abortion` exception; abortion is always deferred while abortion is in progress.

The effect of calling `Undefer_Abortion` is undefined unless abortion is deferred. The effect of calling `Undefer_Abortion` when abortion has been deferred most recently in the calling task by a call other than `Defer_Abortion` (e.g. `Lock`) is undefined.

5.4.2 RTS Notes

The intent of the definition of `Defer_Abortion` and `Undefer_Abortion` is that abortion deferral be nestable. For example, a `Defer_Abortion`/`Undefer_Abortion` pair can be called within a protected object without concern that the `Undefer_Abortion` will allow abortion within an eggshell, which would violate Ada semantics.

5.4.3 Compiler Responsibilities

The compiler shall not call `Undefer_Abortion` unless abortion is deferred, and only if the most recent abortion deferral was the result of a call to `Defer_Abortion`.

5.4.4 Compiler Notes

Abortion should be deferred during the execution of certain constructs, including protected actions, finalizations, while waiting for the completion of a rendezvous in progress, and during abortion itself. In most cases, abortion can be controlled as part of the TRTS calls used to implement these constructs. However, there are vulnerable sections of code that are under the control of the compiler, such as finalization, where the compiler must assure that abortion is deferred (see Section 6.5.3).

The `Abortion` exception should be handled in each nested asynchronous select statement being executed by the task in LIFO order, making the appropriate calls

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

to the TRTS to undefer abortion. If the asynchronous **select** statement is nested in another construct that is being aborted, this will raise the `Abortion` exception again; otherwise it will undefer abortion and return. This in effect allows the TRTS to decide when the correct level of ATC has been aborted.

5.5 Aborting Tasks

The `Abort_Tasks` operation allows a group of tasks to be aborted in parallel.

5.5.1 RTS Responsibilities

`Abort_Tasks` shall abort the set of tasks specified by the `Tasks` parameter by raising the `Abortion` exception and deferring further abortion in the effected tasks, and by marking the tasks as abnormal. If a task is suspended and not abortion deferred, it will be completed before `Abort_Tasks` returns. Tasks dependent on the aborted tasks are also aborted. It also causes all operations that undefer abortion (`Undefer_Abortion`, above; `Unlock`, in Section 9) to re-raise the exception whenever they are called by the affected task.

The effect of `Abort_Tasks` is undefined if `Tasks` contains any references to invalid task IDs or the lower index bound is not 1.

5.5.2 Compiler Responsibilities

The parameter `Tasks` of `Abort_Tasks` shall be a list of valid task IDs. The lower index bound of the array shall be 1.

5.5.3 Compiler Notes

The **abort** statement corresponds directly to the `Abort_Tasks` procedure. This will raise the abortion exception in the affected tasks. Whenever the `Abortion` exception is handled (as for an asynchronous **select**) statement, and attempt to undefer abortion should be made. This raises `Abortion` again if the abortion is to a higher level, which will always be the case when the whole task is aborted. Eventually, target tasks will reach the abortion handler for the whole task, which will call `Complete_Task`. The task procedure will then return to the TRTS, which will terminate it (see Section 6.12.4).

The statement `Abort_Tasks((1=>T1,2=>T2));` is not necessarily equivalent to `Abort_Tasks((1=>T1)); Abort_Tasks((1=>T2));` in the case that the calling task is `T1` or a dependent of `T1`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 6: Task Creation and Activation

This section describes the TRTS interface that supports transitions between the stages of a task's lifetime, from nonexistence through activation, normal execution, completion, and termination. This interface is provided by the package `System.Task_Stages`.

6.1 Package Specification

```
with System.Machine_Specifics;
with System.Rendezvous;
with System.Task_IDs; use System.Task_IDs;
package System.Task_Stages is
  type Master_ID is private;
  function  Current_Master return Master_ID;
  procedure Enter_Master;
  procedure Complete_Master;
  type Access_Boolean is access Boolean;
  type Size_Type is new Machine_Specifics.Task_Storage_Size;
  Unspecified_Size: constant Size_Type:= Size_Type'First;
  type Activation_Chain is limited private;
  Unspecified_Priority: constant Integer:= implementation-defined;
  procedure Create_Task(
      Size: Size_Type;
      Priority: Integer;
      Num_Entries: System.Rendezvous.Task_Entry_Index;
      Master: Master_ID;
      State: Machine_Specifics.Init_State;
      Uninterpreted_Data: System.Address;
      Elaborated: Access_Boolean;
      Chain: in out Activation_Chain;
      Created_Task: out Task_ID);
  procedure Activate_Tasks(Chain: in out Activation_Chain);
  procedure Expunge_Unactivated_Tasks(Chain: in out Activation_Chain);
  procedure Complete_Activation;
  procedure Complete_Task;
  function  Terminated(T: Task_ID) return Boolean;
private
  type Master_ID is ...;
  type Activation_Chain is ...;
end System.Task_Stages;
```

The description of this package is divided into several subsections.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.2 Task Masters

6.2.1 RTS Responsibilities

A value of the type `Master_ID` is called a *master ID*. A master ID shall be *valid for a task* `T` if and only if it has been returned by a call to `Current_Master` from `T`, and only until `T` executes a call to `Complete_Master` or `Complete_Task`.

A valid master ID shall uniquely identify a master within a particular task and its dependents (here distinguished from other tasks and masters within them) until that master is exited.

6.2.2 Compiler Responsibilities

The compiler shall assure that a stack frame is generated for each nontrivial master.

6.2.3 Compiler Notes

The concept of a "master" is the basis of the rules for task termination in Ada. A master is a construct that performs finalization of local objects after it is complete, but before leaving, and can be a task body, block statement, subprogram body, or **`accept`** statement.

Task objects declared by the master, or referred to by objects of access types declared by the master, are said to *depend* on that master. Dependent tasks must terminate before a master performs finalization on other objects that it has created. Since **`accept`** statements have no declarative part, tasks are never dependent on them, and so a master for purposes of task termination is a task body, block statement, or subprogram body.

Finalization of objects other than tasks is the responsibility of the compiler, and is not part of the TRTS. The TRTS is only concerned with masters for purposes of task termination. For this reason, a master which is a task body or has a local declaration of either (1) a task object or an object with a task component, or (2) an access type whose designated type is a task type or a type with a task component is referred to as a *nontrivial master*; in the exposition that follows, the term master will generally mean a non-trivial master. In order to properly implement task termination, the TRTS must be notified of the change in master-nesting whenever execution is about to enter or leave a nontrivial master. This is done via procedures `Enter_Master` and `Complete_Master` (or, in the case of tasks, via `Create_Task` and `Complete_Task`), which are described further below.

Master IDs are used to specify masters of tasks. A master ID is guaranteed to be valid for a task so long as it is only referenced from within the master which obtained that value by calling `Current_Master` (including masters that may be called from or nested within that master).

The requirement for a stack frame for each nontrivial master is to allow the TRTS to use the combination of current frame pointer and task ID for the current task to

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

78 determine the master of a task (see Section 6.4.2, below).

79 6.3 Current Master

80 The `Current_Master` function returns a value of the `Master_ID` type that identifies
81 the master from which the function is called.

82 6.3.1 RTS Responsibilities

83 A master ID that is returned by the `Current_Master` function shall uniquely identify
84 the currently executing master within the calling task until that master is exited.
85 A master that is not a task shall be considered to have been exited when the `Com-`
86 `plete_Master` operation is called by that master, and a master that is a task shall
87 be considered to have been exited when the `Complete_Task` operation is called by
88 that task.

89 6.3.2 Compiler Notes

90 `Current_Master` should be used by the compiler-generated code to associate a master
91 ID with tasks dependent on it. For task object declarations, this can be done by
92 calling `Current_Master` at the point where the task is created (see `Create_Task`,
93 below). For task access types, `Current_Master` should be called at the point where
94 the type is elaborated and saved, so that it can be accessed for any task created by
95 the allocation of an object of that type.

96 6.4 Master Entry

97 `Enter_Master` notifies the TRTS that execution is entering a new master.

98 6.4.1 RTS Responsibilities

99 The TRTS is responsible for keeping track of the master nesting of each task.

100 6.4.2 RTS Notes

101 If the TRTS keeps track of master nesting via a counter per task, this procedure
102 can be implemented rather simply, as an increment operation on the master nesting
103 count of the calling task.

104 Another possibility is to use the master frame pointer as the level counter. With this
105 implementation, `Enter_Master` does not perform any operation. `Current_Master`
106 returns the current frame pointer, which the compiler-generated code associates,
107 together with the *parent* task (the task which elaborated the master), with the tasks
108 dependent on the master. In fact, the frame pointer alone is adequate to identify
109 the master, since only the parent's stack can contain a given master frame pointer.
110 However, explicitly associating the parent with each dependent task avoids the need
111 to search all tasks for it.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

When `Complete_Master` is called, the frame pointer of the master at that point is compared with the master ID associated with children of the task that elaborated the master. Children which are more deeply nested must terminate before the master can complete. This search can be made less expensive by having each task keep a list of its children. When one of these masters is exited, only the list of the task which elaborated the master needs to be searched.

6.4.3 Compiler Responsibilities

The compiler shall ensure that `Enter_Master` is called (exactly once) whenever execution enters a nontrivial master. It shall not be called on entry to task masters that are task bodies. (For masters that are task bodies, the effect is subsumed by `Create_Task` and for the environment task the effect is subsumed by start-up of the TRTS of the associated active partition.)

6.4.4 Compiler Notes

The compiler is permitted to call `Enter_Master` on entry to a trivial master. This may be desirable in cases where the compiler cannot determine if a master is trivial or not. However, the implementation of `Enter_Master` may involve significant overhead, so the compiler should avoid calling it for trivial masters whenever possible.

6.5 Master Exit

`Complete_Master` notifies the TRTS of completion of a master.

6.5.1 RTS Responsibilities

For this operation, the TRTS shall perform all actions required before exit from a master of tasks. These actions are:

1. Terminate any dependent tasks that have not been activated.
2. Block the task calling `Complete_Master` until all the tasks dependent on the current master are either terminated or are waiting on an open **`terminate`** alternative. Once this state has been reached, terminate the tasks waiting on **`terminate`** alternatives.
3. When all the tasks dependent on the master have terminated, deallocate any remaining storage associated with these tasks.
4. Update the TRTS's internal state to recognize the (new) currently executing master's identity within the task that made the TRTS call.

The effect of `Complete_Master` is undefined if there is a rendezvous that was begun by a call to `Selective_Wait` or `Accept_Simple` from within that master but has not yet been completed by a matching call to `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous`. It is also undefined if called from a task executing a protected action.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.5.2 Compiler Responsibilities

The compiler shall ensure that `Complete_Master` is called (once) after the master completes, whether normally, via a **`return`**, **`exit`** or **`goto`** statement, or through propagation of an exception, but prior to finalization of the master. It shall not be called on exit from masters that are task bodies. (The effect in the latter case is subsumed by `Complete_Task`.)

After return from a call to `Complete_Master`, or termination of a task that is a master, no further reference to this master's ID or to the task ID of any dependent task of this master shall be made by the compiler-generated code. (Such master IDs will no longer be valid. Of course, all such dependent tasks must have terminated by this time.)

6.5.3 Compiler Notes

`Complete_Master` must be called before finalization of a task master, regardless of how the master was completed. This requires that `Complete_Master` be called in a handler that is always executed, and which is protected against abortion. Even if the master is exited due to an exception raised in its declarations, it must call `Complete_Master` to terminate tasks that have been created but not yet activated.

This requirement on the compiler is beyond that made by the Ada language itself for exception handling and finalization. For example, consider a declare block which contains tasks:

```
  declare
    ... mixed task and other object declarations
  begin
    ... code for the sequence of statements of the task body
  end;
```

A possible translation of this to use TRTS calls might be:

```
  declare
    Enter_Master;
    ... code for elaboration of master declarations
  begin
    ... code for the sequence of statements of the master.
  when others=>
    Complete_Master;
    raise;
  end;
```

However, if an exception is raised in the elaboration code, it will be propagated; exception handlers are not executed unless the exception is raised in the corresponding handled sequence of statements, which does not have a declarative part. This would mean that `Complete_Master` would not be called. `Complete_Master` cannot be called from a dynamically enclosing exception handler, since the finalization of the declare block would already have been done, contrary to Ada semantics.

A possible alternative is to use the finalization mechanism to call `Complete_Master`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

For illustrative purposes task completion is shown implemented by a dummy controlled object created by the last declaration in the procedure representing the task body. The idea is that, since finalization is performed in LIFO order, `Complete_Master` will be called before any user finalization is performed:

```
Complete_Master_Type is new Finalization.Root_Limited_Controlled;
procedure Finalize(C: in out Complete_Master_Type) is
begin
  Complete_Master;
end Finalize;
declare
  Enter_Master;
  ... mixed master and other object declarations
  Complete_Master_Object: Complete_Master_Type;
begin
  ... code for the sequence of statements of the master.
end;
```

However, this also fails if an exception is raised during the declarative part of the master, since `Complete_Master_Object` will not have been elaborated yet, and will therefore not be finalized.

Any compiler will have to solve this problem with an extension to Ada semantics. The solution used in the GNAT compiler is a special clause that can be added to any master containing a sequence of statements that will be executed when that master (in the Ada sense, including **accept** statements) is left, either through normal completion or through the propagation of an exception (including the `Abortion` exception), even when the exception is raised in the declarative part. The syntax that has been adapted to express this construct is a sequence of statements, starting with the keywords **at end**, between the exception handlers (if any) and the **end** keyword of the master.

```
declare
  Enter_Master;
  ... mixed master and other object declarations
begin
  ... code for the sequence of statements of the master.
at end
  Complete_Master;
end;
```

The **at end** clause will be used throughout this document to express operations that must be executed when leaving a master or other construct which requires cleanup before being left. If the handler was entered as the result of an exception, the exception is propagated to the surrounding execution when the handler completes.

An additional problem is posed by asynchronous ATC. If the `Abortion` exception can be raised at any point in the code, any handler is vulnerable to being aborted before it can complete its processing. For TRTS calls like `Complete_Master` which must complete to ensure proper Ada semantics, this cannot be permitted. Note that it is not enough to simply defer abortion in `Complete_Master`, since the handler might be aborted before it is called.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

For this reason, the compiler assures that abortion is deferred before the handler is entered, and that it is enabled again by calling `Undefer_Abortion` when the handler is complete. If the handler is entered because of an `Abortion` exception, abortion will already be deferred. In other cases, the TRTS must defer it using `Defer_Abortion`, perhaps as part of raising the exception.

The `Undefer_Abortion` operation at the end of the **at end** handler will either undefer abortion or raise the `Abortion` exception, leaving abortion deferred. The propagation of an `Abortion` exception indicates that abortion is in progress, and it should not be propagated unless abortion is deferred. Therefore, unlike other exceptions, the `Abortion` exception, an **at end** handler should not propagate it to the surrounding execution unless this is done as part of undeferring abortion.

For constructs implemented using matched initialization/finalization TRTS calls like `Enter_Master` and `Complete_Master`, there is a similar problem at the beginning of the construct. `Complete_Master` must be called to leave a master, but the master does not begin until `Enter_Master` is called. If `Enter_Master` is called before entering the block protected by the handler that executes `Complete_Master`, an abortion could occur between the return from `Enter_Master` and the beginning of the block. `Complete_Master` would never be called, resulting in undefined behavior. If `Enter_Master` were protected by the handler, an abortion occurring before `Enter_Master` was called would be handled, resulting in the execution of an unmatched `Complete_Master`.

This problem can be addressed by deferring abortion before entering the block and undeferring it only after the initialization call returns:

```
Defer_Abortion;
declare
  Enter_Master;
  Undefer_Abortion;
  ... mixed master and other object declarations
begin
  ... code for the sequence of statements of the master
at end
  Complete_Master;
end;
```

This is awkward, particularly for procedures, where the `Defer_Abortion` call would have to be made before each procedure call. As a shorthand notation for this technique **at start** handler is therefore defined which can be attached to a block, and which will always be executed before the block starts. It defers abortion, executes the specied operations, enters the block, and undefers abortion:

```
at start
  Enter_Master;
declare
  ... mixed task and other object declarations
begin
  ... code for the sequence of statements of the task body
at end
  Complete_Master;
end;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.5.4 Example of Use of Masters

The following is an example inteded to illustrate how masters enter into the creation of tasks. In this example, the locations of key TRTS calls and implicit variables that must be allocated by the compiler are indicated in comments. For simplicity of exposition, code to assure that all exit and cleanup calls are made in the presense of exceptions is omited.

```
package Pkg_1 is
  task T1;
  -- Create_Task(...Master=> Current_Master,...);
  task type Type_2;
  procedure P;
end Pkg_1;

package body Pkg_1 is
  X: Integer:= 1;
  task body T1 is
  begin
    P;
  --at end
    -- Complete_Task;
  end T1;
  task body Type_2 is
  begin
    if X=1 then X:=X+1; P; end if;
  --at end
    -- Complete_Task;
  end Type_2;
  --at start
    -- Enter_Master;
  procedure P is
    T2: Type_2;
    -- Create_Task(...,Master=> Current_Master,...);
    type Acc_T is access Type_2;
    -- Acc_T_Master: Master_ID:= Current_Master;
    -- is master of tasks created via allocators
    -- returning type Acc_T.
  begin
    --at start
      -- Enter_Master;
    B: declare
      task T3;
      -- Create_Task(..., Master=> Current_Master,...);
      task body T3 is
        procedure R is
          --a trivial master, since it has no local tasks
          --or task access types.
          X: Acc_T;
          --at start
            -- Enter_Master;
          procedure S is
            T4: Type_2;
            -- Create_Task(...,Master=> Current_Master,...);
          begin
            X:= new Type_2;
            -- Create_Task(...,Master=> Acc_T_Master,...);
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
337             --at end
338               -- Complete_Master;
339             end S;
340           begin
341             S;
342           end R;
343         begin
344           R;
345         --at end
346           -- Complete_Task;
347         end T3;
348       begin ...
349       --at end
350         -- Complete_Master;
351       end B;
352     --at end
353       -- Complete_Master;
354     end P;
355   begin ...
356   end Pkg_1;

357   with Pkg_1;
358   --at start
359     -- Enter_Master;
360   procedure Main is
361     T5: Pkg_1.Type_2;
362     -- Create_Task(...,Master=>Current_Master,...);
363   begin null;
364   --at end
365     -- Complete_Master;
366   end Main;
```

367 6.6 Activation Chains

368 The `Activation_Chain` type is a limited private type used to specify sets of tasks
369 that are to be activated together. Activation chain objects are initialized to a state
370 that specifies the empty set of tasks. An object of the type `Activation_Chain` is
371 called an *activation chain*.

372 6.6.1 RTS Responsibilities

373 An activation chain object is *valid for a task* `T` if it is either empty or specifies a set
374 of unactivated tasks, all of which were created by calls to `Create_Task` made by `T`.

375 The TRTS shall define the representation of the `Activation_Chain` type, and en-
376 sure that the default initial value specifies the empty set. If the implementation
377 of activation chains involves dynamic storage allocation (other than on the runtime
378 stack), the the TRTS shall provide appropriate finalization to ensure all storage for
379 activation chain objects is recovered on scope exit.

380 6.6.2 RTS Notes

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The Activation_Chain type may be implemented by the TRTS in several different ways. In particular, if tasks are chained together directly, an activation chain may be represented via a reference to the task control block of the last task in the chain.

If separate TRTS storage is used for the chain Activate_Tasks can reclaim this storage in the normal case, and either Expunge_Unactivated_Tasks or Complete_Master can reclaim it in the failure case.

6.6.3 Compiler Responsibilities

The compiler shall pass a valid activation chain to each call to Create_Task.

6.6.4 Compiler Notes

The compiler should create an object of type Activation_Chain for each master and for each allocator that creates tasks. This chain should be passed to each call to Create_Task made by that master or allocator. Either Activate_Tasks or Expunge_Unactivated_Tasks should be called with this object as an argument before the master's body starts execution or before execution or elaboration proceeds beyond the allocator.

The compiler should only put dependents of a single master on an activation chain. Otherwise, activations might occur in the wrong order.

6.7 Task Creation

The Create_Task operation is used to create an unactivated task.

6.7.1 RTS Responsibilities

A call to Create_Task shall create a new unactivated task and return a valid task ID representing this task, in the parameter Created_Task. The newly created task shall not start execution until it is activated by a call to Activate_Tasks. The effects of the parameters shall be as follows:

1. Size shall be the initial stack size of the task. The TRTS is responsible for (eventually) allocating space for the stack and initializing it as necessary to activate the task. If Size is less than the minimum required to begin activation of a task, the TRTS may round the actual stack size up to that minimum.

2. Priority shall be the initial base priority for the task, if it is the range System.Any_Priority. If the value is Unspecified_Priority, the initial base priority of the task shall be the active priority of the activator, if any; otherwise, System.Default_Priority.

   The value Unspecified_Priority shall be defined by the TRTS, and shall be outside of the range System.Any_Priority.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

3. The `Num_Entries` parameter shall be the number of entries of the task. (How entries are counted is explained further in Section 8.2).

4. `Master` shall be the master on which the task directly depends. The effect is undefined if `Master` is not a valid master ID for the calling task. If `Master` was created by a call made to `Enter_Master` while abortion was deferred, the TRTS may raise `Program_Error` without creating a task.

5. `State` shall be the initial state of the new thread of control. Establishing this state in the new thread of control will be equivalent to calling a procedure with a single argument of type `System.Address` within the context of that thread.

6. `Uninterpreted_Data` is uninterpreted data passed as the actual parameter of the procedure called through `State`.

7. The actual object corresponding to `Elaborated` shall be the *elaboration-check variable* associated with the task. `Activate_Tasks` shall raise `Program_Error` if the value of this object is `False` when `Activate_Tasks` is called (later) for the chain containing this task.

8. The task shall be added to the activation chain specified by the `Chain` parameter. The effect is undefined if the chain already contains a task with a different activator.

The TRTS shall create a new task unless it has insufficient resources to do so. If the TRTS does not have does not have sufficient resources to create the new task it shall raise `Storage_Error`. (Resources needed to create a new task include (at least) storage for the task's runtime stack and storage for TRTS internal data structures.)

6.7.2 RTS Notes

Before returning control to the calling task, the TRTS should initialize all the data structures that it needs in order to be able to handle any of the other TRTS calls that take a `Task_ID` parameter, and save the value of the `State` parameter so that it will become the initial state of the new task when it begins activation.

The TRTS may allocate and initialize the stack any time between the call to `Create_Task` and the activation of the task.

6.7.3 Compiler Responsibilities

The compiler shall provide correct parameters to `Create_Task`. In particular:

1. `Size` shall be a non-negative value.

2. `Priority` shall be `Unspecified_Priority` or a value in the range `Any_Priority`.

3. `Master` shall be a valid master ID for the calling task.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

4. `State` shall permit a procedure with a single `System.Address` parameter to be called with the value of `Uninterpreted_Data`.

5. `Chain` shall be a valid activation chain for the calling task.

6. `Created_Task` shall be discriminated with a value enough to cover the range of entry indices which the compiler will use for rendezvous operations for that task, as defined in Section 8.2.

6.7.4 Compiler Notes

Ada 9X allows tasks to be declared with discriminants. Some means must be provided to allow a task to obtain the values of its discriminants when it is created. For this purpose, `Create_Task` takes a value of type `System.Address` in its `Uninterpreted_Data` parameter and passes it to the compiler-generated procedure which executes the task body.

Exactly how the compiler-generated code uses the address parameter is up to the compiler. All tasks are assumed to share a common virtual address space, so this address can be used as a pointer to a data block containing the discriminants. The compiler can translate the formal discriminant specification of the task to an equivalent record type, with a component for each discriminant, and pass the address of this record as the `Uninterpreted_Data` parameter of `Create_Task`.

The value of the `Priority` parameter should be `Unspecified_Priority` if the initial base priority of the task is not specified by pragma. This allows the TRTS to choose the priority.

The variable passed as `Elaborated` should be initialized to `False` by compiler-generated code when the task type declaration (possibly anonymous) corresponding to the new task is elaborated. The value of this object should be changed to `True` by the compiler-generated code when the task body is elaborated.

Creation of the environment task requires special consideration, and is discussed further below.

6.7.5 Rationale

The `Priority` parameter does not have a constraint, because Ada does not permit raising an exception for incorrect parameters to pragmas, and because it allows use of an out-of-range value to indicate there is no priority specified. The priority of a task where no priority is specified is the priority at which it was created, that is, the priority of the activator at the time that it calls `Create_Task`. This is most easily calculated by the TRTS, so the priority value `Unspecified_Priority` is provided to indicate that no `Priority` pragma was specified for a task.

The `Elaborated` parameter is provided to make implementation of elaboration checks more convenient for the compiler. The compiler cannot always determine statically which tasks will be activated and what their task types are. For example,

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

consider the elaboration of a record containing task objects within variant components that depend on the results of function calls; the determination of which tasks are created is made at run time. Without this feature, in those cases where elaboration checks cannot be performed at compile time the compiler would have to generate additional data structures or a special runtime routine to perform these checks for each unconstrained type with task components.

Note also that AI-00149 requires that when activating a list of tasks, if the activation of one task raises `Program_Error` as a result of the task body not being elaborated yet, no task shall be allowed to proceed. Therefore, in order to avoid another synchronization point in the created tasks (after they check their elaboration status), the activator task is required to make the check for all tasks in the chain before beginning the activation of any of them. Furthermore, it is not clear that on all systems one can rely on the task code itself to check its own elaboration status.

6.8 Task Activation

`Activate_Tasks` causes a set of tasks to begin activation.

6.8.1 RTS Responsibilities

`Activate_Tasks` shall cause the TRTS to start activation of the set of tasks specified by the `Chain` parameter. There shall be no return from this call until all tasks in the set complete activating, normally or through an exception, or all are determined by the TRTS not to be activatable, because one or more of the corresponding task bodies has not been elaborated. If all tasks in the set complete activation normally, the call shall return normally and update the `Chain` parameter to designate the empty set. Otherwise, if one or more of the tasks has not completed activating normally, the return shall be via propagation of `Tasking_Error` or `Program_Error`, as specified by the Ada language definition.

The effect of `Activate_Tasks` is undefined if `Chain` is not valid for the calling task, or if the chain contains any tasks that depend on a master that has become invalid (due to exit from the master).

6.8.2 Compiler Responsibilities

The activation chain specified by the `Chain` parameter shall be valid for the task that calls `Activate_Tasks`.

`Chain` shall not contain any tasks that depend on a master which has become invalid.

6.8.3 Compiler Notes

The compiler should create a variable of the `Activation_Chain` type for each non-trivial task master, and should create additional chains to handle the evaluation of allocators that create (and activate) tasks. Whenever it creates a task, the compiler should know the master or allocator evaluation to which it belongs, and pass the

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

appropriate chain. The compiler needs to keep track of which chains need to be activated and generate a call to `Activate_Tasks` for each activation chain at the appropriate location.

6.9 Expunction of Unactivated Tasks

`Expunge_Unactivated_Tasks` expunges a set of unactivated tasks, specified by an activation chain.

6.9.1 RTS Responsibilities

The TRTS shall expunge the tasks on `Chain`. The effect of this call shall be that all the tasks in the specified group become terminated, the associated task IDs become invalid, and all associated resources shall be recovered. The effect is undefined if the activation chain is not valid for the calling task.

6.9.2 Compiler Responsibilities

The activation chain specified by the `Chain` parameter shall be valid for the task that calls `Activate_Tasks`.

`Chain` shall not contain any tasks that depend on a master which has become invalid.

The compiler shall make no further use of task IDs identifying tasks on the activation chain. If any further reference is made to these tasks, the compiler should arrange for a value of `Null_Task` to be substituted for the value originally returned by `Create_Task`.

6.9.3 Compiler Notes

Ada requires a task to be terminated before leaving its master. This includes tasks which were never activated. There must be a call to either `Complete_Master` or `Complete_Task` before exiting the master, and unactivated tasks could be terminated, and any resources allocated to the reclaimed, by this calls. However, tasks can be created by packages and allocators, and these constructs are not masters. Failure of the elaboration of a package spec, or of an allocator, can leave resources allocated to unactivated tasks for as long as the master exists. In the case of library level packages, this is essentially for the lifetime of the program. One possible result is storage leakage.

`Expunge_Unactivated_Tasks` allows an implementation to reclaim resources allocated for unactivated tasks upon the failure of a package or allocator that creates tasks. This also allows storage allocated by a failed allocator to reclaim storage immediately, since it must first finalize any subcomponents elaborated before the point of failure, and all of the tasks that it has created must be terminated before finalization can take place.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.9.4 Example of Task Allocation and Activation

Consider the following declare statement:

```
B:  declare
    task type Task_Type is
      pragma Priority(9);
    end Task_Type;
    type Task_Pointer is access Task_Type;
    for Task_Type'Storage_Size use 200;
    task body Task_Type is ...
    begin ... end Task_Type;
    T: Task_Pointer;
    procedure Inner is
    begin
      T:= new Task_Type;
    end Inner;
          ... other local declarations of B.
  begin
     Inner;
  end B
```

One way to implement this statement is to put **at end** handlers around the allocation to make sure that `Expunge_Unallocated_Tasks` is always called, as explained in Section 6.5.3. If the call to `Activate_Tasks` is executed, it will remove all tasks from the chain, in which case calling `Expunge_Unactivated_Tasks` on it will have no effect.

```
B:  declare
    Task_Type_Elaborated: aliased Boolean:= False;
    Task_Master: Master_ID:= Current_Master;
    procedure Task_Type_Body is ...
    begin ... end Task_Type_Body;
    Task_Type_Elaborated:= True;
    T: Task_ID;
    procedure Inner is
    begin
      declare
        Local_Chain: Activation_Chain;
      begin
         --... code to put initialization data into Init_Data.
         -- For GNAT:
         -- Init_Data:= Task_Type_Body'Access;
         Create_Task(
             Size=> 200,
             Priority=>  9,
             Num_Entries=> 0,
             Master=> Task_Master;
             State=> Init_Data,
             Uninterpreted_Data=> System.Null_Address,
             Chain=> Local_Chain,
             Elaborated=> Task_Type_Elaborated'Access,
             Created_Task=> T);
         Activate_Tasks(Local_Chain);
      at end
        Expunge_Unactivated_Tasks(Local_Chain);
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
      end;
    end Inner;
          ... other local declarations of B.
  begin
     Inner;
  end B
```

Another possibility is to make Activation_Chain part of a finalizable object. Since Local_Chain is the only declaration in the declare block used to enclose both the creation and activation of the task, no other finalization routines can execute before its finalization[1]. The translation is the same, except that the inner declare block does not need handlers:

```
B:  declare
    Task_Type_Elaborated: aliased Boolean:= False;
    Task_Master: Master_ID:= Current_Master;
    procedure Task_Type_Body is ...
    begin ... end Task_Type_Body;
    Task_Type_Elaborated:= True;
    T: Task_ID;
    procedure Inner is
    begin
      declare
        type Finalizable_Chain is new Controlled with record
          A: Activation_Chain;
          end record;
        procedure Finalize(Object: in out Finalizable_Chain) is
        begin
          Expunge_Unactivated_Tasks(Object.A);
        end Finalize;
        Local_Chain: Finalizable_Chain;
      begin
         --... code to put initialization data into Init_Data.
         -- For GNAT:
         -- Init_Data:= Task_Type_Body'Access;
         Create_Task(Size=> 200,
           Priority=>  9,
           Num_Entries=> 0,
           Master=> Task_Master;
           State=> Init_Data,
           Uninterpreted_Data=> System.Null_Address,
           Chain=> Local_Chain.A,
           Elaborated=> Task_Type_Elaborated'Access,
           Created_Task=> T);
         Activate_Tasks(Local_Chain.A);
      end;
    end Inner;
          ... other local declarations of B.
  begin
     Inner;
  end B
```

[1] If there were any such finalization routines, they would be able to see the tasks as unterminated, contrary to Ada semantics.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.9.5 Example of Activation and Creation

For an example of creation and activation through the elaboration of a task object, consider the following block statement, containing two local tasks.

```
B: declare
           ... other local declarations of B.
     task type TT(D: Integer) is
       pragma Priority(9);
     end TT;
     for TT'Storage_Size use 200;
     T1: TT(1);
     task T2 is
       entry E1(1..10);
       entry E2;
       pragma Priority(10);
     end T2;
     for T2'Storage_Size use 100;
     task body T1 is ...
     begin ... end T1;
     task body T2 is ...
     begin ... end T2;
           ... other local declarations of B.
   begin ... sequence of statements of B.
   end B;
```

The structure of the translated code for this block statement is given below.

```
... code for elaboration of other local declarations of B
T1_ID, T2_ID: Task_ID;
T1_Elaborated: constant aliased Boolean:= False;
T2_Elaborated: constant aliased Boolean:= False;
Local_Chain: Activation_Chain;
type Discriminant_Record is record
  D: Integer;
  end record;
Discriminant_1: Discriminant_Record;
procedure T1(Discriminants: System.Address) is ...
begin ... end T1;
T1_Elaborated:= True;
procedure T2(Discriminants: System.Address) is ...
begin ... end T2;
T2_Elaborated:= True;
...
at start
  Enter_Master;
begin
  ...
  -- code to put initialization data into INIT_DATA,
  -- including the starting address of T1.
  -- For GNAT (See Section 1.7.1.1): Init_Data:= T1'Access;

  Discriminant_1.D:= 1;
  Create_Task(
      Size=> 200,
      Priority=>  9,
      Num_Entries=> 0,
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
715       Master=> Current_Master,
716       State=> Init_Data,
717       Uninterpreted_Data=> Discriminant_1'Address,
718       Chain=> Local_Chain,
719       Elaborated=> T1_Elaborated'Access,
720       Created_Task=> T1_ID);

721   -- code to put initialization data into Init_Data.
722   -- For GNAT: Init_Data:= T2'Access;

723   Create_Task(
724       Size=> 100,
725       Priority=> 10,
726       Num_Entries=> 11,
727       Master=> Current_Master,
728       State=> Init_Data,
729       Uninterpreted_Data=> System.Null_Address,
730       Chain=> Local_Chain,
731       Elaborated=> T2_Elaborated'Access,
732       Created_Task=> T2_ID);
733   ... code for elaboration of other local declarations of B
734   Activate_Tasks(Local_Chain);
735   ... code for sequence of statements of B.
736 at end
737   Complete_Master;
738 end;
```

739 In this example, `Ti_ID` is an implicit local variable to receive the task ID identifying
740 the new task `Ti`, and `Init_Data` specifies how to call the `Ti` task procedure (e.g. a
741 procedure access value). `Ti` is a procedure used to implement the body of the task
742 `Ti`. As explained in Section 6.5.3, **`at start`** and **`at end`** handlers are used to express
743 the requirement that `Complete_Master` must be called if and only if `Enter_Master`
744 is called, even in the presence of exceptions or abortion.

745 6.10 Activation Completion

746 `Complete_Activation` notifies the TRTS that the calling task has completed acti-
747 vation.

748 6.10.1 RTS Responsibilities

749 If this is the last task to complete activation in a chain of tasks, the TRTS shall release
750 the activator of the chain to resume execution. The effect of calling `Complete_-`
751 `Activation` from a task that has not started activation (i.e. is not on an activation
752 chain for which `Activate_Tasks` has been called), or of calling it more than once
753 from a given task, is undefined.

754 6.10.2 Compiler Responsibilities

755 The compiler will assure that `Complete_Activation` is called no more than once
756 from each task. It will assure that it is called only from tasks that have started
757 activation.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.10.3 Compiler Notes

`Complete_Activation` should be called (exactly once) by each task when it has just completed activation, after the elaboration of its own declarative part is complete.

`Complete_Activation` should not be called by a task that fails to complete activation, due to an exception being raised and not handled during the activation. Its functions will be subsumed by `Complete_Task` in this case.

6.11 Task Completion

`Complete_Task` notifies the TRTS that the body of the calling task has "completed" in the Ada language sense. (The task ordinarily is not completely finished executing, since it may still need to wait for dependents and then execute finalization code.)

6.11.1 RTS Responsibilities

If the task has dependents, the actions of the TRTS for `Complete_Task` shall include all actions required for `Complete_Master`. In addition, the TRTS shall perform all actions required by the Ada language for completion of a task.

1. Check for pending calls on entries of the task, and raise `Tasking_Error` in any such calling tasks.

2. Arrange to raise `Tasking_Error` in the activator, if the present task is not finished activating (by calling `Complete_Activation`).

3. Set the `Callable` attribute of the task to `False`.

The effect of `Complete_Task` is undefined if there is a rendezvous that was begun by a call to `Selective_Wait` or `Accept_Simple` but has not yet been completed by a matching call to `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous`. Its effect is also undefined if called after a call to `Lock` or `Lock_Read_Only` without a corresponding call to `Unlock`. Its effect is also undefined if it is called between a call to `Task_Entry_Call` and the corresponding `Cancel_Task_Entry_Call`, or `Protected_Entry_Call` and the corresponding `Cancel_Protected_Entry_Call`.

6.11.2 RTS Notes

`Complete_Task` has no parameters. All the context information needed to process this call needs to be kept by the TRTS.

Ada requires a task to be completed before finalization. Since finalization will need access to objects declared in the task, `Complete_Task` cannot in general reclaim the resources used for such objects. This usually means that `Complete_Task` will be unable to deallocate the task's stack. This must be deferred until task termination.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.11.3 Compiler Responsibilities

The compiler shall ensure that `Complete_Task` is called (once) after the task body completes, whether normally, via a **`return`**, **`exit`** or **`goto`** statement, or through propagation of an exception, but prior to finalization of the task.

6.11.4 Compiler Notes

As with `Complete_Master`, the requirement that `Complete_Task` be called before an exception is propagated out of a task body cannot be expressed directly in Ada (see Section 6.5.3). Examples of task completion are shown using Ada with extensions.

`Complete_Task` will wait for completion of all tasks dependent on it, even those created by masters executed by the task. However, it will not ensure the finalization of such internal masters, and those masters must wait on their own dependent tasks before finalization. Therefore, `Complete_Task` should not be called until all of the masters it has started executing have been completed and left.

6.12 Task Termination

The TRTS "terminates" a task, in the Ada language sense, when the task body is left. At this point, the task has completed execution, all of its dependents are terminated, and all of its finalization has been performed.

6.12.1 RTS Responsibilities

The TRTS shall set the `Terminated` attribute of the task to `True` on leaving the body of a task.

The effect of leaving a task body is undefined if the task has not previously called `Complete_Task`. Its effect is also undefined if finalization has not been performed on the task.

6.12.2 RTS Notes

The TRTS calls the procedure representing the task body to elaborate the task body declarations and execute its sequence of statements. This procedure returns to the point of call in the TRTS when the task completes, after calling `Complete_Task`, as specified above, and after all finalization has been completed. The TRTS terminates the task at this point.

Since finalization must be complete when the body is left, there should be no further reference to objects declared in the task body. The TRTS can therefore deallocate these objects, usually by deallocating the task stack.

6.12.3 Compiler Responsibilities

The compiler shall implement the body of a task as a procedure (see Section 1.7.1).

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.12.4 Example of Task Activation, Completion, and Termination

Consider a task body such as:

```
task body T is
  -- declarations
begin
  -- sequence of statements
end T;
```

`Complete_Activation`, `Complete_Task`, and `Terminate_Task` can be used in a task body procedure to control the lifetime of this task as follows:

```
procedure T_Body(Discriminants: System.Address) is
begin
  ... code for elaboration of the task's local declarations
  Complete_Activation;
  ... code for the sequence of statements of the task body.
at end
  Complete_Task(T);
end T_Body;
```

6.13 Terminated Attribute

The `Terminated` function shall return the value of the language-defined `Terminated` task attribute

6.13.1 RTS Responsibilities

`Terminated` shall return `True` if its argument represents a task that has been terminated, or is `Null_Task`; otherwise, it returns false. The effect of calling `Terminated` with an invalid task ID that is not `Null_Task` is undefined.

6.13.2 Compiler Responsibilities

The compiler shall assure that the argument to `Terminated` is a valid task ID or `Null_Task`.

6.13.3 Compiler Notes

For each instance of the task attribute `T'Terminated`, the compiler should generate a call to the `Terminated` function with a task ID identifying `T` as its argument. If the compiler-generated code has expunged a task before its master is left, this task is also terminated, though the task ID assigned to it is no longer valid. In such cases, the compiler-generated code can translate all references to the task to `Null_Task`, or simply replace its `Terminated` attribute with `True`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

6.14 Environment Tasks

The environment task is implicitly created by the elaboration of the Ada TRTS packages.

6.14.1 RTS Responsibilities

The TRTS shall treat the thread of control that elaborates the TRTS packages as the environment task. After the elaboration of the TRTS packages is complete, it shall be allowable for this thread of control to make any legal TRTS call.

6.14.2 RTS Notes

When an environment task completes, it should wait for the termination of tasks created by library packages, which are dependent on the environment task. The TRTS should also perform whatever additional cleanup is needed to shut down the partition. This might include closing of any open files. A call-out for user-defined finalization should perhaps be provided, to execute after all the Ada 9X object finalization.

Immediate shut-down of an entire process, as via the POSIX *exit()* operation, should bypass normal finalization and termination processing. Otherwise, there is risk of a program never being able to shut down at all.

6.14.3 Compiler Responsibilities

The elaboration order of the load-image must ensure that the TRTS code is elaborated to completion, by a single thread of control, before the execution of any code that makes TRTS calls.

6.14.4 Compiler Notes

Each nontrivial master is required to be bracketed by calls to `Enter_Master` and `Complete_Master`. This includes the main program procedure, which is executed by the environment task. Tasks created by the main program procedure are dependent on it, and not (directly) on the environment task, and the main program procedure must wait for them to terminate before returning to the TRTS.

The compiler should translate an Ada program (or active partition) into a program that can be loaded as a single-threaded process under the host operating system.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 7: Delays

This section describes the TRTS support for various forms of **delay** statements.

7.1 Package Specifications

```
package System.Real_Time.Delays is

   protected Delay_Until_Object is
     entry Wait(T: Time);
   end Delay_Until_Object;

end System.Real_Time.Delays;

package Calendar.Delays is

   protected Delay_Object is
     entry Wait(T: Duration);
   end Delay_Object;

   protected Delay_Until_Object is
     entry Wait(T: Time);
   end Delay_Until_Object;

end Calendar.Delays;
```

7.1.1 RTS Responsibilities

A package *time_package* provided by the TRTS which defines a time type shall have a child package *time_package*.Delays which defines Delay_Until_Object with a Wait entry which takes that time type as a parameter. A call to the Wait entry will be queued for the amount of time specified by its argument, as measured by the Clock function in the corresponding package. After this time has arrived, the entry call will be dequeued. The Wait entry body has no other effect.

Relative delays are supported by the Delay_Object defined in System.Real_Time package. The the GNARLI provides a similar Calendar.Delays.Delay_Object which defines a Wait entry that takes a Duration parameter.

7.1.2 Compiler Notes

The compiler should generate a simple protected entry call on Calendar.Delays.-Delay_Object.Wait for a **delay** statement, passing its Duration expression to the entry.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The compiler should generate a simple protected entry call on *time_package*.Delays.Delay_Object.Wait for a **delay until** statement, where *time_package* is the Ada package defining the time type of the delay expression.

Asynchronous delay statements should be implemented as asynchronous protected entry calls to the appropriate Wait entry.

Timed entry calls should be implemented as asynchronous entry calls, where the abortable part is a simple **delay** or **delay until** statement.

Select statements should be implemented as asynchronous protected entry calls to the appropriate Wait entry, with the Selective_Wait TRTS call as the abortable part.

7.1.3 Example of a Simple Relative Delay Statement

Consider a simple relative delay statement:

```
delay T;
```

This can be implemented by the following code fragment:

```
Calendar.Delays.Delay_Object.Wait(T);
```

7.1.4 Example of a Simple Absolute Delay Statement

Consider a simple absolute delay statement on a time type implemented by the package Some_Time_Type.

```
declare
  T: Some_Time_Type.Time:= Some_Time_Type.Time_Of(1993, 12, 10);
begin
  delay until T;
end;
```

Note that there need not be a Time_Of function in this package; it is only shown to indicate some method of obtaining a time type value. This can be implemented by the following code fragment:

```
declare
  T: Some_Time_Type.Time:= Some_Time_Type.Time_Of(1993, 12, 10);
begin
  Some_Time_Type.Delays.Delay_Until_Object.Wait(T);
end;
```

7.1.5 Example of an Asynchronous Delay Statement

Consider the following asynchronous **select** statement with a **delay until** trigger:

```
declare
  T: Some_Time_Type.Time:= Some_Time_Type.Time_Of(1993, 12, 10);
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
66  begin
67    select
68      delay until T;
69    then abort
70      S;
71    end;
72  end;
```

73 This can be implemented by the following code fragment:

```
74  declare
75    T: Some_Time_Type.Time:= Some_Time_Type.Time_Of(1993, 12, 10);
76  begin
77    select
78      Some_Time_Type.Delays.Delay_Until_Object.Wait(T);
79    then abort
80      S;
81    end;
82  end;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

83 *This side is intentionally blank.*

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 8: Rendezvous

The TRTS operations required to implement rendezvous are provided by the `System.Rendezvous` package.

8.1 Package Specification

```
with System.Compiler_Exceptions;
with System.Task_IDs; use System.Task_IDs;
with System.Protected_Objects;
package System.Rendezvous is

   Null_Task_Entry: constant:= 0;
   Max_Task_Entry: constant:= ...; -- value defined by TRTS.
   type Task_Entry_Index is range Null_Task_Entry..Max_Task_Entry;

   procedure Task_Entry_Call(
       Acceptor: Task_ID;
       E: Task_Entry_Index;
       Uninterpreted_Data: System.Address;
       Mode: System.Protected_Objects.Call_Modes;
       Rendezvous_Successful: out Boolean);
   procedure Call_Simple(
       Acceptor: Task_ID;
       E: Task_Entry_Index;
       Uninterpreted_Data: System.Address);
   procedure Cancel_Task_Entry_Call(Cancelled: out Boolean);
   procedure Requeue_Task_Entry(
       Acceptor: Task_ID;
       E: Task_Entry_Index;
       With_Abort: Boolean);
   No_Rendezvous: constant:= 0;
   type Select_Modes is (Simple_Mode, Else_Mode, Terminate_Mode);
   Max_Select: constant:= implementation-defined;
   type Select_Index is range No_Rendezvous..Max_Select;
   subtype Positive_Select_Index is
       Select_Index range 1 .. Select_Index'Last;
   type Accept_Alternative is record -- should be packed
      Null_Body: Boolean;
      E: Task_Entry_Index;
      end record;
   type Accept_List is
       array(Positive_Select_Index range <>) of Accept_Alternative;
   -- Lower index bound shall be 1.
   procedure Selective_Wait(
       Open_Accepts: Accept_List;
       Select_Mode: Select_Modes;
       Uninterpreted_Data: out System.Address;
       Index: out Select_Index);
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
45    procedure Accept_Call(
46        E: Task_Entry_Index;
47        Uninterpreted_Data: out System.Address);
48    procedure Accept_Trivial(E: Task_Entry_Index);
49    function  Count(E: Task_Entry_Index) return Natural;
50    function  Callable(T: Task_ID) return Boolean;
51    procedure Complete_Rendezvous;
52    procedure Exceptional_Complete_Rendezvous(
53        Ex: Compiler_Exceptions.Exception_ID);
54 end System.Rendezvous;
```

55 The description of this package is divided into several subsections.

56 8.2 Entry Indices

57 8.2.1 RTS Responsibilities

58 The `Task_Entry_Index` type is used to identify specific entries of a task. An entry
59 index value `i` is *valid for a task* `T` if `i` is in the range `1..Num_Entries`, where `Num_-`
60 `Entries` is the number of entries specified in the call to `Create_Task` that created
61 `T`.

62 8.2.2 Compiler Responsibilities

63 The compiler shall assign a unique positive value of the type `Task_Entry_Index` to
64 each entry of a task type. In assigning indices to entries, each component of an entry
65 family shall be considered to be an entry.

66 The largest entry index value used for a task shall be communicated by the compiler-
67 generated code to the TRTS at the time of task creation, via the `Num_Entries`
68 parameter of `Create_Task`.

69 8.2.3 Compiler Notes

70 It is recommended that the entry indices used for a task be assigned in a compact
71 range, in the order in which the entry declarations appear. The compiler should
72 not leave gaps in the range of entry indices assigned to a task, since the TRTS may
73 allocate storage for a separate entry queue for each entry index value in the range
74 `1..Num_Entries`.

75 The compiler should allocate indices for members of entry families in a fashion
76 similar to the allocation of storage for arrays, as if an entry family were an array of
77 entries, and the entry indices were addresses of individual elements.

78 8.3 Entry Calls

79 The `Task_Entry_Call` operation is used to implement Ada entry calls.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

8.3.1 RTS Responsibilities

This operation shall make an entry call to the task entry specified by the `Acceptor` and `E` parameters. The effect of the operation is undefined unless `Acceptor` is a valid task ID, and `E` is a valid entry index for that task. The value of `Uninterpreted_Data` shall be passed to the acceptor, as the `Uninterpreted_Data` of the corresponding accept operation (see Section 1.9). If `Acceptor` represents a terminated task, this call will raise `Tasking_Error` in the caller.

`Task_Entry_Call` makes different kinds of entry calls according to the value of the `Mode` parameter. The specific actions of `Task_Entry_Call` for the various values of `Mode` are:

1. `Simple_Call` shall make an unconditional entry call.
2. `Conditional_Call` shall make a conditional entry call. It shall set the parameter `Rendezvous_Successful` to `False` if there is no rendezvous, and `True` if there is a successful rendezvous.
3. `Asynchronous_Call` shall make an asynchronous entry call, during which the calling task may continue execution of an abortable sequence of statements.

The detailed TRTS actions required for these calls are:

1. Check that the acceptor is callable, and raise `Tasking_Error` if it is not.
2. For `Conditional_Call`, return immediately if the called task is not ready to accept.
3. For `Simple_Call`, if the called task is not ready to accept, block the calling task.
4. For `Asynchronous_Call`, if abortion is deferred, the TRTS may raise `Program_Error`. Otherwise, if the called task is not ready to accept, queue the calling task on the specified entry.
5. When the called task is ready to accept, transfer the parameter address and begin the rendezvous. (This involves unblocking and possibly adjusting the dispatching priority of the accepting task.) Except for `Asynchronous_Call`, block the calling task.

8.3.2 Compiler Responsibilities

The compiler is required to pass valid parameter values:

1. `Acceptor` must be a valid task ID.
2. `E` must be an entry index in the range `1..Num_Entries`, where `Num_Entries` is the parameter passed to the call to `Create_Task` that created the `Acceptor` task.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

8.3.3 Compiler Notes

With Ada 9X we can have a task continue execution while an entry call is pending. We must take care to put the entry parameters some place where they will not be clobbered. This will probably be done by using the runtime stack. The effect will be similar to treating the asynchronous select statement as a declare-block, where the entry parameter block is a local variable of the declare-block.

`Task_Entry_Call` is intended to be used to implement simple, conditional, or asynchronous entry calls when used with a mode of `Simple_Call`, `Conditional_Call`, or `Asynchronous_Call`, respectively. It can also be used in conjunction with **`delay`** to implement timed entry calls in the `Asynchronous_Call` mode.

The following examples are intended to show how these operations could be used by a compiler to implement conditional entry calls. Examples of the implementation of timed and asynchronous calls are given in Section 8.5.

8.3.4 Example of a Conditional Entry Call

An example of a conditional entry call is:

```
select T.E(X); ... sequence of statements (1).
else           ... sequence of statements (2).
end select;
```

Suppose `E` is the entry of task `A` associated with index 1, and the task ID identifying `T` is stored in the variable `T`. The statement above would then be translated into something like the following, where `S` is a local temporary variable supplied by the compiler:

```
... code to copy the parameter-passing information into location P.
Task_Entry_Call(
    Acceptor=> T,
    E=> 1,
    Uninterpreted_Data=> P,
    Mode=> Conditional_Call,
    Rendezvous_Successful=> S);
if S then ... code for sequence of statements (1).
else            ... code for sequence of statements (2).
end if;
```

8.4 Simple Entry Calls

The `Call_Simple` operation is provided to implement the simple call statement with potentially less overhead than if then `Task_Entry_Call` were used.

8.4.1 RTS Responsibilities

`Call_Simple(T, E, P)` shall be equivalent to

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
153   Task_Entry_Call(
154       Acceptor=> T,
155       E=> E,
156       Uninterpreted_Data=> P,
157       Mode=> Simple_Call,
158       Rendezvous_Successful=> S)
```

159 where S is a dummy variable whose value is ignored.

160 8.5 Completing Asynchronous Task Entry Calls

161 Cancel_Task_Entry_Call is used to removed a pending asynchronous task entry
162 call from the corresponding queue.

163 8.5.1 RTS Responsibilities

164 Cancel_Task_Entry_Call shall remove the task entry call (that has not already
165 been removed) made by the most recent call to Task_Entry_Call executed by the
166 calling task. If the call has not yet completed, removing it entails cancellation. If
167 the call is not either complete or queued non-abortably, Cancel_Task_Entry_Call
168 shall block until the it is. If the call is actually cancelled, the Cancelled parameter
169 will be True on return; otherwise, it will be False.

170 The effect of calling Cancel_Task_Entry_Call when the last entry call made by
171 the task which has not yet been removed is not an asynchronous task entry call is
172 undefined.

173 8.5.2 Compiler Responsibilities

174 The compiler shall assure that Cancel_Task_Entry_Call is called only when the
175 most recent entry call made by the calling task is an asynchronous protected entry
176 call, and that Cancel_Task_Entry_Call is called only once for this call.

177 8.5.3 Compiler Notes

178 The compiler should call Cancel_Task_Entry_Call exactly once for each asyn-
179 chronous task entry call when the abortable part completes. This applies to the
180 case where the abortable part is aborted by the completion of the triggering task
181 entry call. This implements the semantics of the completion of an abortable part,
182 cancelling the call if queued abortably and waiting for it to complete or be queued
183 abortably if not. In either case, this call removes the current asynchronous task
184 entry call and makes the previous asynchronous entry call (if any) current. This is
185 the reason that this call must be made exactly once for each asynchronous task entry
186 call; it must be called once to remove the call, but calling it again would remove the
187 previous call prematurely, or result in undefined behavior if no such call exists.

188 The compiler should arrange for calls to Task_Entry_Call with an Asynchronous_-

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Call mode parameter and Cancel_Task_Entry_Call to be executed in pairs, even in the presence of asynchronous abortion. This is similar to the requirement for paired execution of Enter_Master and Complete_Master, as explained in Section 6.5.3, and the same extensions to Ada used to express that requirement are used in the following examples of the use of Cancel_Task_Entry_Call

8.5.4 Example of an Asynchronous Task Entry Call

Consider the following asynchronous **select** statement on a task entry:

```
select
  T.E;
  S1;
then abort
  S2;
end select;
```

This call might be translated as:

```
at start
  declare
    Dummy: Boolean;
  begin
    Task_Entry_Call(
        Acceptor=> T,
        E=> E,
        Uninterpreted_Data=> System.Null_Address;
        Mode=> Asynchronous_Call,
        Rendezvous_Successful=> Dummy);
  end;
declare
  Cancelled: Boolean;
begin
  S2;
at end
  Cancel_Task_Entry_Call(Cancelled);
end;
if not Cancelled then
  S1;
end if;
```

Task_Entry_Call either starts or queues the entry call abortably and returns, at which time the abortable part S2 is executed. If the abortable part completes normally, it attempts to cancel the entry call using Cancel_Task_Entry_Call. If the entry call is queued abortably, it will be canceled; otherwise, Cancel_Task_Entry_Call waits until the call is complete. If the entry call completes successfully, the task will be marked with the nesting level of this asynchronous select and an Abortion pseudo-exception will be raised by Complete_Rendezvous. This will be caught by the compiler-generated exception handler which will call Cancel_Task_Entry_Call.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

8.5.5 Example of a Timed Task Entry Call

The asynchronous form of `Task_Entry_Call` can be used to implement a timed task entry call. For example, the time entry call:

```
select
  T.E;
  S1;
or delay until Calendar.Time_Of(1993, 12, 10);
  S2;
end select;
```

might be translated as:

```
at start
  Task_Entry_Call(
      Acceptor=> T,
      E=> E,
      Uninterpreted_Data=> System.Null_Address,
      Mode=> Asynchronous_Call,
      Rendezvous_Successful=> Dummy);
declare
  Deadline: Some_Time_Type.Time:= Some_Time_Type.Time_Of(1993, 12, 10);
  Timed_Out: Boolean:= False;
  Ex: Exception_ID:= Null_Exception;
begin
  delay until Deadline;
at end
  Cancel_Task_Entry_Call(Timed_Out);
end;
if Timed_Out then
  S2;
else
  S1;
end if;
```

Note that this translation uses a **`delay until`** statement, which itself should be implemented in terms of TRTS operations (see Section 7).

8.6 Requeuing an Entry Call

8.6.1 RTS Responsibilities

`Requeue_Task_Entry` shall requeue the entry call that is currently being served to a specified task entry. The effect is undefined unless the call occurs while the calling task is serving an entry call. The entry call being serviced can be a task or protected entry call (see Section 9).

The effects of the parameters are as follows:

1. `Acceptor` shall be the task on whose entry on which the call is requeued. The effect is undefined unless this is a valid task ID.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

2. E shall be the index of the entry on which the call is to be requeued. The effect is undefined unless this is a valid entry index for the specified task.

3. With_Abort specifies that the requeued entry call shall be abortable. The call cannot be cancelled until it is queued abortably.

8.6.2 Compiler Responsibilities

Requeue_Task_Entry shall only be called by a task that has executed Selective_Wait or Accept_Simple and before it has executed the corresponding Complete_Rendezvous or Exceptional_Complete_Rendezvous; or which has executed Next_Entry_Call and before it has executed the corresponding Complete_Entry_Body or Exceptional_Complete_Entry_Body. The compiler shall provide correct parameters; in particular:

1. Acceptor shall be a valid task ID.

2. E shall be valid for the task corresponding to Acceptor.

8.6.3 Compiler Notes

Requeue_Task_Entry is intended to implement the Ada **requeue** statement to a task entry when executed within an **accept** or **entry** body. It should be called with With_Abort equal to True to implement an abortable **requeue** statement, and equal to False to implement a non-abortable **requeue** statement.

8.7 Selective Waits

Selective_Wait is provided to implement selective wait statements.

8.7.1 RTS Responsibilities

The effect of the Selective_Wait call shall be to implement the form of selective wait statement specified by the Select_Mode parameter. The specific effects of the parameters are:

1. Open_Accepts shall specify the set of open accept alternatives, on which the calling task is prepared to accept calls. Each component Open_Accepts(I) specifies one accept alternative. If Open_Accepts(I).E is equal to Null_Task_Entry, the accept alternative shall be considered closed. Otherwise, the alternative shall be considered open and ready to accept a call on entry of the calling task with index Open_Accepts(I).E. If Open_Accepts(I).E is not a valid entry index for the calling task, the effect of the call is undefined. The TRTS must allow the same entry index to appear in more than one component of the list.

   If more than one open accept alternative has a pending entry call, which is chosen depends on the queuing policy implemented by the TRTS. Under FIFO

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

queuing, the TRTS will choose one arbitrarily. With priority queuing, more than one accept alternative has a pending entry call, and there is a tie for highest priority, the tie shall be broken in favor of the earlier alternative in the list.

2. `Uninterpreted_Data` is set to the value provided by the caller in its `Uninterpreted_Data` argument, if a rendezvous is successful.

3. `Index` is set to the index of the alternative selected, in the list `Open_Accepts`, if a rendezvous is successful. Otherwise, it is set to `No_Rendezvous`.

The actions performed by the TRTS for the `Selective_Wait` operation are:

1. If a call cannot be accepted immediately:

   (a) For `Else_Mode`, return.

   (b) For `Simple_Mode`, block the accepting task and record that it is waiting for a call to one of the open entries.

   (c) For `Terminate_Mode`, check for possible termination of all the dependent tasks of a master. If the task is not ready for termination, block the accepting task and record that it is waiting for either termination or a call to one of the open entries.

2. If a call can be accepted, raise the acceptor's dispatching priority if necessary, and return control to the acceptor, with the value of `Index` set to indicate which **`accept`** alternative is selected, and the value of `Uninterpreted_Data` set to the value provided by the corresponding entry call operation.

3. On exit from `Selective_Wait`, abortion is deferred, just as if `Defer_Abortion` had been called.

If the call to `Selective_Wait` returns without a rendezvous, the value of `Index` shall be set to `No_Rendezvous`. Otherwise, the value of `Index` shall be the index of the accept alternative chosen for rendezvous

The processing of a rendezvous depends on the value of the `Null_Body` component of the accept alternative selected:

1. If `Null_Body` equals `True`, the rendezvous shall be assumed to be *trivial*. If the entry call accepted was initiated by an asynchronous **`select`** statement, the corresponding abortable part shall be aborted before return from `Selective_Wait` (see Section 5); otherwise, the task that made the entry call shall be made eligible for execution before return from `Selective_Wait`.

2. If `Null_Body` equals `False`, the rendezvous is not trivial, and the caller will continue to execute its abortable part (if any) and then block until the acceptor task signals the end of rendezvous, by calling `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Between the time that `Selective_Wait` returns with a non-trivial rendezvous selected and the time that the corresponding `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous` is executed, a task is said to be *serving a task entry call*.

8.7.2 Compiler Responsibilities

The lower bound of the array passed for `Open_Accepts` shall be one. The `E` component of every array component must be either `Null_Task_Entry` or a valid entry index for the calling task.

The compiler shall not call `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous` unless it is serving a task entry call.

8.7.3 Compiler Notes

The `Selective_Wait` operation can be called to implement all the forms of selective wait statement. The `Select_Modes` type is used to specify a particular form of selective wait statement. The meanings of the values are:

1. `Simple_Mode`. There are only **`accept`** alternatives.
2. `Else_Mode`. There are only **`accept`** alternatives and an **`else`** clause.
3. `Terminate_Mode`. There are only **`accept`** alternatives and a **`terminate`** alternative.

Ordinarily, the compiler should use the mode that corresponds to the syntactic form of the statement. However, if it can determine that certain guards are closed, it may use a more restrictive form. In particular, the compiler can use `Simple_Mode` for various syntactic forms, including simple **`accept`** statements and selects with **`terminate`** alternatives if it can determine that guards on all the non-**`accept`** alternatives are closed.

Note that the components of the `Open_Accepts` parameter naturally correspond to the alternatives of a selective wait statement, and can be generated in the order in which they appear within the text of the Ada source program. Thus, the compiler will be able to translate a selective wait as a **`case`** statement, using the value returned in the `Index` parameter. Components corresponding to **`accept`** alternatives that have no sequence of statements should have the `Null_Body` field set to `True`. If such an alternative is selected, a trivial rendezvous will be performed before `Selective_Wait` returns.

After a return from `Selective_Wait` for which a nontrivial rendezvous has started, the compiler-generated code is responsible for (later) calling `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous` to end the rendezvous. The value of `Index` and prior knowledge of the value of the corresponding `Null_Body` component of `Open_Selects` can be used to determine whether a nontrivial rendezvous has started.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The compiler may use the value `Null_Task_Entry` to turn off those alternatives whose guards evaluate to `False` or which are not **accept** alternatives, so as to preserve a one-to-one correspondence between cases and positions in the open entry list. Alternatively, the compiler may simply omit components for non-**accept** entries and **accept** entries that are closed.

The textual order of the **select** alternatives should be reflected in the `Select_Index` field of the corresponding `Open_Accepts` component. This allows the TRTS to implement queuing policies where textual order is significant.

The compiler-generated code for a selective wait should do several things before calling `Selective_Wait`:

1. It should evaluate any guards on the alternatives of the selective wait statement, then if there are entry family members in open alternatives it should evaluate the entry index expressions.
2. If there are no open alternatives, and there is no **else** part, it should raise `Tasking_Error`.
3. If there are no open alternatives, and there is an **else** part, it should transfer control to the **else** part.
4. For each (open) **accept** alternative, it should add a component to `Open_Accepts`.

> **This requires the compiler to do some complicated stuff in code generation. An alternative is to have the TRTS do some of this processing. The reason MRTSI/CIFO chose to put the burden on the compiler was a hope that sometimes this processing could be optimized, using local context.**

In cases where there are no guards, the work can mostly be done at compile time, and `Open_Accepts` can be a constant. This optimization may significantly reduce the execution time of a **select** statement.

8.7.4 Example of a Selective Wait Statement

An example of a selective wait statement and rendezvous code is:

```
select
  when B1=> accept A do S1; end A; T1;
  or when B2=> accept B; T2;
  or when B3=> accept C do S3; end C; T3;
  or when B4=> accept A; T4;
  else T5;
end select;
```

One of several equivalent ways to structure the TRTS calls for this statement would be as follows:

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
421 declare
422   Q: Accept_List:=
423       ((Null_Body=>False,E=>A),
424       (Null_Body=>True, E=>B),
425       (Null_Body=>False,E=>C),
426       (Null_Body=>True, E=>A));
427   X: Select_Index;
428   P: System.Address;
429 begin
430   if not B1 then Q(1).E:= Null_Task_Entry; end if;
431   if not B2 then Q(2).E:= Null_Task_Entry; end if;
432   if not B3 then Q(3).E:= Null_Task_Entry; end if;
433   if not B4 then Q(4).E:= Null_Task_Entry; end if;
434   -- or some "optimized" version of the above.

435   at start
436     Selective_Wait(
437         Open_Accepts=>Q,
438         Select_Mode=> Else_Mode,
439         Uninterpreted_Data=>P, Index=>X);
440   begin
441     case X is
442     when No_Rendezvous=>
443       null;
444     when 1=>
445       S1;
446     when 2=>
447       null;
448     when 3=>
449       S3;
450     when 4=>
451       null;
452     when others=> raise Program_Error;
453     end case;
454   at end
455     if X /= No_Rendezvous and then not Q(X).Null_Body then
456       Exceptional_Complete_Rendezvous(Current_Rendezvous);
457     end if;
458   end;

459   case X is
460   when No_Rendezvous=>
461     T5;
462   when 1=>
463     T1;
464   when 2=>
465     T2;
466   when 3=>
467     T3;
468   when 4=>
469     T4;
470   end case;
471 end
```

472 In this example, Q is first initialized to a constant vector, then those entries corre-
473 sponding to closed alternatives are "turned off". The creation of Q could be optimized
474 by eliminating items corresponding to closed alternatives from Q, or if B1, B2, B3,

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

and B4 are static or loop-invariant, the construction of Q could be performed earlier. P is used by the translated images of S1..S4 to access the actual entry parameters.

The identifier Current_Exception stands for the current exception, which the compiler-generated code is responsible for knowing, possibly with the help of the exception-handling portion of the RTS. (Recall that the interface between the exception handler code and the exception handling portion of the RTS is not part of the TRTS interface.) This exception will be raised by the RTS in the calling task at the point of the call. If the **at end** handler was not entered as the result of an exception, Current_Exception should be Null_Exception, which will not raise an exception in the calling task. Alternately, Complete_Rendezvous could be called in cases where no exception has been raised. If the select statement is aborted, Current_Exception should be Abortion; the TRTS will raise Tasking_Error in the caller in this case.

Note that there is no rendezvous code, and no call to End_Rendezvous or Complete_Rendezvous, for the cases where Null_Body is true.

8.7.5 Example of a Selective Wait Statement With Delay Alternatives

Selective_Wait can also be used in conjunction with delay protected objects to implement **select** statements with **delay** alternatives. This can be implemented using the Selective_Wait operation, with the timeout provided by an asynchronous **select** statement with a Delay_Object trigger.

Selective_Wait should first be called with Else_Mode; otherwise, a very short timeout could cancel the Selective_Wait before it could accept a call that was already queued, contrary to Ada semantics. If there is more than one open **delay** alternative, the compiler should choose the alternative with the minimum delay, and use the corresponding delay expression as the argument to the Wait entry call used to implement the timeout. If there is more than one minimum delay, the alternative should be chosen according to the queuing policy implemented by the TRTS. If the minimum delay value is negative or zero, there is no need for a timeout, and the Selective_Wait operation with mode Else_Mode is sufficient.

For example, consider the following selective wait statement.

```
select
  when B1=> accept A do S1; end A; T1;
  or when B2=> accept B; T2;
  or delay until Time1; T3;
  or delay until Time2; T4;
  else T5;
end select;
```

One possible translation of this statement is shown below.

```
declare
  X: Select_Index:= No_Rendezvous;
  Q: Accept_List;
  P: System.Address;
  T: Some_Time_Type.Time;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
518 begin
519   Q:= ((Null_Body=>False,E=>A),
520        (Null_Body=>True, E=>B));
521   if not B1 then Q(1):= Null_Task_Entry; end if;
522   if not B2 then Q(2):= Null_Task_Entry; end if;
523   if Time1 > Time2 then
524     T:= Time2;
525   else
526     T:= Time1;
527   end if;
528   -- or some "optimized" version of the above.

529   -- Try for immediate rendezvous first.

530   at start
531     Selective_Wait(Q, Else_Mode, P, X);
532   begin
533     case X is
534     when No_Rendezvous=>
535       null;
536     when 1=>
537       S1;
538     when 2=>
539       T2;
540     when others=>
541       null;
542     end case;
543   at end
544     if X /= No_Rendezvous and then not Q(X).Null_Body then
545       Exceptional_Complete_Rendezvous(Current_Rendezvous);
546     end if;
547   end;

548   -- If immediate rendezvous does not work, try waiting for the
549   -- specified delay.

550   if X = No_Rendezvous then
551     select
552       Some_Time_Type.Delays.Delay_Object.Wait(T);
553       Defer_Abortion;
554     then abort
555       at start
556         Selective_Wait(Q, Simple_Mode, P, X);
557       begin
558         case X is
559         when No_Rendezvous=>
560           null;
561         when 1=>
562           S1;
563         when 2=>
564           T2;
565         when others=>
566           null;
567         end case;
568       at end
569         Exceptional_Complete_Rendezvous(Current_Exception);
570       end;
571     end select;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
572   end if;

573   if X = No_Rendezvous then
574     if Time1 > Time2 then
575       T4;
576     else
577       T3;
578     end if;
579   end if;

580 end;
```

581 8.8 Simple Accept Statements

582 The `Accept_Call` operation is provided to implement the simple **accept** statement
583 with potentially less overhead than if the `Selective_Wait` operation were used.

584 8.8.1 RTS Responsibilities

585 `Accept_Call(E,L)` shall be equivalent to

```
586   Selective_Wait(
587       (1..1=>(E=>E, Null_Body=>False)),
588       Simple_Mode,
589       P,
590       X)
```

591 where `P` and `X` are compiler-generated variables to receive the parameter block
592 pointer and chosen select index values returned by `Selective_Wait`; `X` would be
593 ignored in this case.

594 8.8.2 Compiler Notes

595 `Accept_Call` should be called for an **accept** statement with a sequence of state-
596 ments. It can also be used to implement selective wait statements where the
597 compiler-generated code can determine which alternative will be selected.

598 8.9 Trivial Accepts

599 The `Accept_Trivial` operation provides a way to implement trivial accept state-
600 ments.

601 8.9.1 RTS Responsibilities

602 `Accept_Trivial(E)` shall be equivalent to

```
603   Selective_Wait(
604       (1..1=>(E=>E, Null_Body=>True)),
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
605         Simple_Mode,
606         P,
607         X)
```

608 where `P` and `X` are dummies that are ignored by the compiler-generated code.

609 8.9.2 Compiler Notes

610 `Accept_Trivial` should be called for an **`accept`** statement that has no internal
611 sequence of statements.

612 8.10 Completing Rendezvous

613 The `Complete_Rendezvous` and `Exceptional_Complete_Rendezvous` operations
614 are provided to complete a non-trivial rendezvous.

615 8.10.1 RTS Responsibilities

616 Before return from `Complete_Rendezvous` or `Exceptional_Complete_Ren-`
617 `dezvous`, the TRTS shall perform the following actions:

618 1. Lower the dispatching priority of the accepting task, if necessary.

619 2. Defer abortion for the caller.

620 3. Propagate the exception specified by the exception-ID parameter `E` to the caller,
621 in the case of `Exceptional_Complete_Rendezvous`.

622 4. Make the caller eligible for execution.

623 5. If the call was made by an asynchronous **`select`** statement, abort the associ-
624 ated abortable part (see Section 5).

625 The effect of these operations is undefined unless the calling task has a rendezvous
626 under way—that is, there must have been a successful preceding `Selective_Wait`
627 or `Accept_Call` operation for which no `Complete_Rendezvous` or `Exceptional_-`
628 `Complete_Rendezvous` operation has yet been performed.

629 The effect of calling `Complete_Rendezvous` or `Exceptional_Complete_Ren-`
630 `dezvous` operation when no rendezvous is in progress is undefined.

631 8.10.2 RTS Notes

632 The `Complete_Rendezvous` and `Exceptional_Complete_Rendezvous` operations
633 require the TRTS to keep track of the current caller, so that it can be awakened and
634 so that exceptions can be propagated to it, as necessary. Since **`accept`** and **`select`**
635 statements can be nested, the TRTS has to keep track of all callers being serviced by

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

a given task, and has to restore the current caller at the start of the corresponding rendezvous when `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous` is called. This can be done by keeping a stack of current callers for each task, for example.

8.10.3 Compiler Responsibilities

The compiler shall assure that `Complete_Rendezvous` or `Exceptional_Complete_Rendezvous` operation are called only when a nontrivial rendezvous started by the `Selective_Wait` or `Accept_Call` operation is in progress.

8.10.4 Compiler Notes

`Complete_Rendezvous` should be called by the compiler-generated code for normal completion of the rendezvous, just before exit from the **accept** body. `Exceptional_Complete_Rendezvous` should be called by the compiler-generated code (or runtime system code implementing exception propagation) when an exception is propagated out of the **accept** body. The `Ex` parameter of `Exceptional_Complete_Rendezvous` should be the ID of the exception that is to be propagated to the calling task, or `Null_Exception` if there is no such exception (this has the same effect as `Complete_Rendezvous`).

These rendezvous completion operations should not be called for rendezvous initiated by the `Accept_Trivial` operation, or for cases of a selective wait where `Null_Body` is specified as `True`, since the rendezvous will be complete on return from these calls.

8.11 Count Attribute

8.11.1 RTS Responsibilities

The `Count` function shall return the number of calls queued on the specified entry in the calling task. The effect of the call is undefined unless the `E` parameter is a valid entry index for the calling task.

8.11.2 Compiler Responsibilities

The value of `E` shall be a valid entry index for the calling task.

8.11.3 Compiler Notes

`Count` should be called to obtain the value of the `Count` attribute of a task entry.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

8.12 Callable Attribute

8.12.1 RTS Responsibilities

The `Callable` function shall return `False` if `T` identifies a task which is completed or abnormal, or if it is `Null_Task`; otherwise, it returns `True`.

The effect of the call is undefined unless the `T` parameter is valid or `Null_Task`.

8.12.2 Compiler Responsibilities

The compiler shall assure that the argument to `Callable` is a valid task ID or `Null_Task`.

8.12.3 Compiler Notes

The `Callable` function should be called to obtain the value of the `Callable` task attribute.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 9: Protected Objects

The TRTS operations required to implement protected objects are provided by the `System.Protected_Objects` package.

9.1 Description

```
with System.Compiler_Exceptions;
package System.Protected_Objects is
   Null_Protected_Entry: constant:= 0;
   Max_Protected_Entry: constant:= implementation-defined;
   type Protected_Entry_Index is range
       Null_Protected_Entry..Max_Protected_Entry;
   subtype Positive_Protected_Entry_Index is
       Protected_Entry_Index range <1..Protected_Entry_Index'Last>;
   type Call_Modes is (Simple_Call, Conditional_Call, Asynchronous_Call);
   type Protection(Num_Entries: Protected_Entry_Index) is private;
   procedure Initialize_Protection(
       Object: in out Protection;
       Ceiling_Priority: Integer);
   procedure Finalize_Protection(Object: in out Protection);
   procedure Lock(Object: in out Protection);
   procedure Lock_Read_Only(Object: in out Protection);
   procedure Unlock(Object: in out Protection);
   procedure Protected_Entry_Call(
       Object: in out Protection;
       E: Protected_Entry_Index;
       Uninterpreted_Data: System.Address;
       Mode: Call_Modes);
   procedure Cancel_Protected_Entry_Call(Cancelled: out Boolean);
   type Barrier_Vector is
       array(Positive_Protected_Entry_Index range <>) of Boolean;
   procedure Next_Entry_Call(
       Barriers: Barrier_Vector;
       Uninterpreted_Data: out System.Address;
       E: out Protected_Entry_Index);
   procedure Wait_for_Completion(Call_Successful: out Boolean);
   procedure Complete_Entry_Body;
   procedure Exceptional_Complete_Entry_Body(
       Ex: Compiler_Exceptions.Exception_ID);
   procedure Requeue_Protected_Entry(
       New_Object: in out Protection;
       E: Protected_Entry_Index;
       With_Abort: Boolean);
   function Count(E: Protected_Entry_Index) return Natural;
   procedure Broadcast_Program_Error;
private
   -- TRTS-defined
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
46 end System.Protected_Objects;
```

47 The description of this package is divided into several subsections.

48 9.2 Representation of Protected Objects

49 Protected objects are identified to the TRTS as objects of type `Protection`.

50 9.2.1 RTS Responsibilities

51 The TRTS shall maintain data structures for controlling access to a protected object
52 and for queuing tasks on protected entries.

53 9.2.2 RTS Notes

54 The data structures used to implement protected actions are accessed only by the
55 TRTS. The data is not assumed to be organized in any particular way, and is not
56 assumed to have any particular relationship to the corresponding `Protection` ob-
57 ject. For example, the `Protection` type might contain all necessary structures, or
58 it might refer indirectly to some structures built by the TRTS. In the most extreme
59 case, the `Protection` type might consist of just its discriminant plus a pointer or
60 index to the information needed to control the corresponding protected object.

61 9.2.3 Compiler Notes

62 For each protected object, the compiler should declare an object of the `Protection`
63 type, constrained by the maximum entry index, at the point where the protected
64 object declaration appears.

65 A protected object can contain private components declared in the corresponding
66 protected type. These components are accessed only by compiler generated code;
67 the TRTS has no access to this information. The compiler is responsible for the
68 association between this information and the corresponding `Protection` object.

69 9.3 Entry Indices

70 As with task entries, entries in protected objects are represented by nonnegative
71 integers, with each member of an entry family considered to be an entry. However,
72 the two entry mechanisms are different, and the range of entry values supported by
73 the implementation of one may not be suitable for the implementation of the other.
74 A separate type `Protected_Entry_Index` is therefore defined in this package to
75 represent protected entries.

76 9.3.1 RTS Responsibilities

77 An entry index value `i` is *valid for a protected object* `P` if `i` is in the range `1..Num_-`
78 `Entries`, where `Num_Entries` is the number of entries specified by the discriminant

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

constraint with which the corresponding `Protection` type was declared.

9.3.2 Compiler Responsibilities

The largest entry value used for a protected object shall be used as the `Num_Entries` discriminant constraint in the elaboration of the corresponding `Protection` object.

9.3.3 Compiler Notes

The compiler should assign a unique positive value of the type `Protected_Entry_Index` to each entry of a protected object. In assigning indices to entries, each component of an entry family shall be considered to be an entry. The indices should be assigned in textual order to allow for queuing policies (e.g. `Priority_Queuing`) where textual order is significant.

The assignment of entry index values to protected entries is analogous to the assignment of entry index values to task entries, as outlined in Section 8.2.3. That is, the compiler should assign a compact range of index values to protected entries, and members of entry families should be mapped onto indices as if they were arrays of entries.

9.4 Parameter Passing

Entry bodies can be executed by tasks other than that which made the corresponding entry call. It must therefore be possible to pass entry parameters as uninterpreted data through the TRTS between tasks, as described in Section 1.9.

9.4.1 Compiler Notes

The compiler can translate the formal parameter specification of the entry to an equivalent record type, with a component for each parameter. The rendezvous parameter would then be the address of a record of this type. For parameters passed by value, the component is of the same type as the parameter and contains the parameter value. For parameters passed by reference, the component is a pointer to the actual parameter object. For parameters passed by result, the component may contain both a value and a pointer.

9.5 Initializing and Finalizing Protected Objects

9.5.1 RTS Responsibilities

Before any of the other operations described in this section can be performed on a `Protection` object, it must be initialized by calling `Initialize_Protection`. `Initialize_Protection` shall assign the specified ceiling priority to the object.

When no further calls to operations defined in this section will be made on a protected object, it should be finalized by calling `Finalize_Protection`. `Finalize_Protec-`

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

tion will remove any remaining calls on the entry queues of the object and raise Program_Error in the callers at the point of the call.

A Protection object is *valid* if Initialize_Protection has been called for it, and if Finalize_Protection has not yet been called for it.

9.5.2 Compiler Responsibilities

The compiler-generated code shall call Initialize_Protection on an object of type Protection before it calls any other TRTS operation with this object as a parameter. Similarly, it shall make no such calls on the object after Finalize_Protection is called on it. It shall call Finalize_Protection on a Protection object before deallocating it.

9.5.3 Compiler Notes

The compiler-generated code should call Initialize_Protection on a Protection object immediately after its elaboration. The value of the Ceiling_Priority parameter should be that specified by the Priority or Interrupt_Priority pragma for that object. If there is no such pragma, the value System.Priority'Last should be used, unless there is an Interrupt_Handler or Attach_Handler pragma appears in the corresponding protected definition, in which case Unspecified_Priority should be used.

The compiler-generated code should call Finalize_Protection on a Protection object immediately before any other finalization of the associated protected object.

9.6 Initiating and Completing Protected Actions

9.6.1 RTS Responsibilities

The Lock procedure functions as a write lock; once a task has successfully executed this call on a protected object, no other tasks will be able to complete a call to Lock or Lock_Read_Only on the same object until Unlock is called with the same protected object as its argument.

The Lock_Read_Only functions as a read lock; this procedure can be called any number of times on the same protected object. However, tasks executing Lock will not be able to proceed until a call to Unlock is made corresponding to each call to Lock_Read_Only.

After executing either Lock or Lock_Read_Only on a protected object, the caller is protected against abortion, and, if priority scheduling is in effect, its effective priority is raised to at least that of the protected object's ceiling priority. The calling task is said to be executing a *protected action* on the specified Protection object. These changes remain in effect until the corresponding Unlock is executed.

The argument to Unlock must be the same as that of the most recent call, within the same task, to Lock or Lock_Read_Only, that is, unlocking must be in LIFO order.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The result of calling `Unlock` with any other argument is undefined. A call to `Unlock` shall raise the `Abortion` exception if it corresponds to the earliest pending call to lock by the calling task, and if it finds that abortion to an outer execution is pending (see Section 5).

The effect of calling any of these operations with an invalid argument is undefined.

9.6.2 RTS Notes

Note that all of these procedures require that the protected object to which they apply be provided as an argument. The TRTS could provide this value for `Unlock`, since it must be the argument to the most recent call to `Lock` or `Lock_Read_Only` made by the calling task. However, since protected actions can be nested, this would require the TRTS to associate a dynamic stack of protected objects with each task. Since these operations should be as lightweight as possible, the burden of keeping track of protected action nesting is placed on the compiler-generated code, which may be able to avoid such a stack under some circumstances.

On some platforms, reader/writer locks may be much more expensive than simple mutual exclusion locking. In such cases, it may be desirable to implement `Lock` and `Lock_Read_Only` identically, allowing only one task to perform a protected action at a time, regardless of the operation being performed. This is particularly true of uniprocessor implementations, where the increased potential for parallelism is of no value.

9.6.3 Compiler Responsibilities

Neither `Lock` nor `Lock_Read_Only` shall be called on a protected object during a protected action on that same object. A task shall call `Unlock` only on the protected object on which it most recently executed a `Lock` or `Lock_Read_Only` operation, that is, protected actions shall be left in LIFO order. The compiler shall only call `Lock`, `Lock_Read_Only`, and `Unlock` on valid `Protection` objects.

9.6.4 Compiler Notes

Before executing the code of a protected procedure, the compiler-generated code should call `Lock` on the corresponding protected object. Before completion of a protected action, the compiler-generated code should execute the entry bodies corresponding to entry calls on that protected object for which the barrier expression is `True`, as described in Section 9.9.3 below. When all remaining entries are queued on `False` barriers, `Unlock` should be called to complete the protected action.

Before executing the code of a protected function, the compiler-generated code should call `Lock_Read_Only`. On completion of the function body, `Unlock` should be called to complete the protected action. The entry queues should not be checked in this case.

RTS operations which may block should not be called while executing protected actions. This is defined as a bounded error in the Ada 9X specification. The com-

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

piler may detect and reject a compilation containing a potentially blocking operation within a protected action, but this is in general impossible. The RTS may raise `Program_Error` exception on detecting the execution of a such a potentially blocking operation; otherwise, this situation may eventually lead to deadlock.

The potentially blocking TRTS operations are:

```
                              Complete_Master
Selective_Wait                Create_Task
Accept_Call                   Activate_Tasks
Accept_Trivial                Expunge_Unactivated_Tasks
Task_Entry_Call               Complete_Activation
Call_Simple                   Complete_Task
Complete_Rendezvous           Cancel_Task_Entry_Call
Abort_Tasks                   Exceptional_Cancel_Task_Entry_Call
Enter_Master                  Cancel_Protected_Entry_Call
                              Exceptional_Cancel_Protected_Entry_Call
```

Other parts of the RTS (e.g. I/O) may define other potentially blocking operations that should not be called during a protected action.

This assumes that all entry calls are potentially blocking, even asynchronous select statements. This is in keeping with RM9X 3.0 9.5.1(9).

9.7 Pending Entry Calls

`Protected_Entry_Call` pends a protected entry call on the specified protected object.

9.7.1 RTS Responsibilities

`Protected_Entry_Call` shall pend the specified entry call on the specified object. This call will either be queued or its index returned by the next call to `Next_Entry_Call` (below) on this object, if the corresponding barrier is `False` or `True`, respectively. The specified object shall be locked on return, just as if `Lock` had been called on the object.

The effect of the parameters is as follows:

1. The parameter `Object` specifies the target object.
2. The parameter `E` specifies the entry within the protected object to be called.
3. The parameter `Uninterpreted_Data` shall be passed to the entry body, as the `Uninterpreted_Data` of the corresponding `Next_Entry_Call` operation (see Section 1.9).
4. The parameter `Mode` specifies the type of call. It controls the action of `Next_Entry_Call`, `Complete_Entry_Body`, `Exceptional_Complete_Entry_Body`,

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

and Wait_For_Completion, as described below. If it is Asynchronous_Call, the TRTS may raise Program_Error if it detects that the call is being made while abortion is deferred (that is, if abortion was already deferred before the protected action started).

9.7.2 RTS Notes

The semantics of this call are described in terms of "pending" the call, rather than queuing it. The distinction is made because a call cannot be queued, and in particular cannot be reflected in the Count attribute of its entry, if the barrier evaluates to True when the call is made (RM9X 3.0, 9.5.3(8–11)).

9.7.3 Compiler Responsibilities

The compiler shall assure that the Object parameter is valid, and that the E parameter is a valid entry index for it.

9.7.4 Compiler Notes

The compiler-generated code should call Protected_Entry_Call with a Mode parameter of Simple_Call for a simple protected entry call, of Conditional_Call for a conditional entry call, and Asynchronous_Call for an asynchronous **select** statement. It should then terminate the protected action after servicing all eligible pending or queued entry calls, as described under Next_Entry_Call below. This has the effect of servicing or queuing the pended call. Asynchronous_Call is also used for implementing timed protected entry calls, as described in Section 8.5.5.

The Uninterpreted_Data argument should point to a record allowing access to the actual parameters of the call, as described in Section 9.4. Uninterpreted data is provided between Protected_Entry_Call and Next_Entry_Call since another task may execute the entry body.

9.8 Completing Asynchronous Protected Entry Calls

Cancel_Task_Entry_Call is used to removed a pending asynchronous protected entry call from the corresponding queue.

9.8.1 RTS Responsibilities

Cancel_Protected_Entry_Call shall remove the protected entry call (that has not already been removed) made by the most recent call to Protected_Entry_Call executed by the calling task. If the call has not yet completed, removing it entails cancellation. If the call is not either complete or queued non-abortably, Cancel_Protected_Entry_Call shall block until the it is. If the call is actually cancelled, the Cancelled parameter will be True on return; otherwise, it will be False. On return from this operation, the associated Protection_Object shall be locked, as if Lock had been called.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The effect of calling `Cancel_Protected_Entry_Call` when the last entry call made by the task which has not yet been removed is not an asynchronous protected entry call is undefined.

9.8.2 Compiler Responsibilities

The compiler shall assure that `Cancel_Protected_Entry_Call` is called only when the most recent entry call made by the calling task is an asynchronous protected entry call, and that `Cancel_Protected_Entry_Call` is called only once for this call.

9.8.3 Compiler Notes

The compiler should call `Cancel_Protected_Entry_Call` exactly once for each asynchronous protected entry call when the abortable part completes. This applies to the case where the abortable part is aborted by the completion of the triggering task entry call. This implements the semantics of the completion of an abortable part, cancelling the call if queued abortably and waiting for it to complete or be queued abortably if not. In either case, this call removes the current asynchronous protected entry call and makes the previous asynchronous entry call (if any) current. This is the reason that this call must be made exactly once for each asynchronous protected entry call; it must be called once to remove the call, but calling it again would remove the previous call prematurely, or result in undefined behavior if no such call exists.

The compiler should arrange for calls to `Protected_Entry_Call` with an `Asynchronous_Call` mode parameter and `Cancel_Protected_Entry_Call` to be executed in pairs, even in the presence of asynchronous abortion. This is similar to the requirement for paired execution of `Enter_Master` and `Complete_Master`, as explained in Section 6.5.3. However, no **`at start`** construct is needed here, as `Protected_Entry_Call` defers abortion as part of locking the protected object. It can be called before entering the block that will execute `Cancel_Protected_Entry_Call` in its **`at end`** handler, since there is no possibility of being aborted until the block is entered and the `Protection` object unlocked.

Cancellation of a protected entry call is a protected operation, so `Cancel_Protected_Entry_Call` locks the associated `Protection` object. Before unlocking the object, the entry queues of the protected object must be serviced, since the protected cancellation operations may change the `Count` attribute of the associated entry and therefore change one or more entry barriers.

9.8.4 Example of an Asynchronous Protected Entry Call

Consider an asynchronous call on the protected object declared in Section 9.14:

```
select
   PO.E1(1);
   S6;
then abort
   S7;
end select;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

This call might be translated as:

```
declare
  P: PO.E1_Params:= (Param=> 1);
  Cancelled: Boolean;
begin
  Protected_Entry_Call(
      Object=> PO.Object,
      E=> 1,
      Uninterpreted_Data=> P'Address,
      Mode=> Asynchronous_Call);
  begin
    PO.Service_Entries;
    S7;
  at end
    Cancel_Protected_Entry_Call(Cancelled);
  end;
  PO.Service_Entries;
  if not Cancelled then
    S6;
  end if;
end;
```

If the abortable part completes normally, it attempts to cancel the entry call with `Cancel_Protected_Entry_Call`; this will only be successful if the entry call is queued abortably.

If the entry barrier later opens and the entry body completes, the task will be marked with the nesting level of this asynchronous **`select`** (if not already marked for abortion to a lower nesting level) and an `Abortion` pseudo-exception will be raised in the abortable part by `Complete_Entry_Body`.

`Cancel_Protected_Entry_Call` is a protected operation, so abortion deferral is provided by locking the protected object. Since `Cancel_Protected_Entry_Call` can change the state of the barriers of a protected object, the entry queues of the object must be serviced after calling it before unlocking the protected object.

Other details are as for asynchronous **`select`** with a task entry call trigger (Section 8.5.4).

9.8.5 Example of a Timed Protected Entry Call

The asynchronous form of `Protected_Entry_Call` can also be used to implement a timed entry call on a protected object. For example, a timed entry call such as:

```
select
  PO.E1(1);
  S1;
or delay 10.0;
  S2;
end select;
```

might be translated as:

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
declare
  Timed_Out: Boolean:= False;
  P: PO.E1_Params:= (Param=> 1);
  Deadline: System.Real_Time.Time:=
      System.Real_Time.Clock + System.Real_Time.To_Time_Span(10.0);
  Protected_Entry_Call(Object=> PO.Object,
      E=> 1,
      Uninterpreted_Data=> P'Address,
      Mode=> Asynchronous_Call);
  begin
    PO.Service_Entries;
    delay until Deadline;
  at end
    Cancel_Protected_Entry_Call(Timed_Out);
  end;
  PO.Service_Entries;
  if Timed_Out then
    S2;
  else
    S1;
  end if;
end;
```

The relative delay is translated to an absolute time, here a value of `System.Real_Time.Time`, since `Protected_Entry_Call` might have to wait for `Object` to be unlocked. This can take an arbitrary amount of time, and it must be done before the **`delay`** is started. If a relative delay were used, the actual delay would include the time required to pend the entry call, resulting in an extremely inaccurate timeout. Note that this expansion includes a **`delay until`** statement, which is itself expanded into GNARLI operations (see Section 7).

9.9 Servicing Entry Calls

`Next_Entry_Call` is used to determine which entries have become eligible for execution as the result of a protected operation.

9.9.1 RTS Responsibilities

`Next_Entry_Call` shall return the next entry call to be serviced. The effect of calling this procedure is undefined unless the calling task is executing a protected action.

The TRTS actions for `Next_Entry_Call` are:

1. If there is a call pending on the object for which the caller is executing a protected action, and the corresponding barrier evaluates to `True`, return the entry index for that call in `E`; otherwise

2. If an element of `Barrier_Vector` corresponding to one or more awaited entries is `True`, return the entry index of one of these in `E`. Which of these is selected will depend on the entry queuing policy implemented by the TRTS.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

3. Remove the pending entry call, if any. If it was pended with `Conditional_Call` and is to be queued abortably (see Section 9.15), discard the call; otherwise, put it in the appropriate queue.

4. If no entry call was selected, return with `E=Null_Protected_Entry`.

5. If `E` specifies an entry, `Uninterpreted_Data` will contain the value passed to `Protected_Entry_Call` in its `Uninterpreted_Data` argument when this call was made pending on the object (see Section 1.9).

Between the time that `Next_Entry_Call` returns with an entry selected and the time that the corresponding `Complete_Entry_Body` or `Exceptional_Complete_Entry_Body` is executed, a task is said to be *serving a protected entry call*.

9.9.2 Compiler Responsibilities

The compiler shall assure that `Next_Entry_Call` is called during a protected action.

9.9.3 Compiler Notes

Immediately after a call to `Protected_Entry_Call`, and also just before unlocking the `Protection` object after leaving a protected procedure or protected entry body, the compiler-generated code should generate a vector of the current values of all barriers. This requires evaluating any barriers that might have changed. If any barrier evaluations propagates an exception, `Program_Error` should be raised in all callers of all entries of the protected object by calling `Broadcast_Program_Error`. Otherwise, `Next_Entry_Call` should be called with this vector and the `Protection` object. If a valid entry index is returned by `Next_Entry_Call` in the parameter `E`, the body of the corresponding entry should then be executed, using `Uninterpreted_Data` to access the actual parameters as described in Section 9.4.

The compiler-generated code should unlock a `Protection` object if and only if:

1. A protected function which is not part of another protected action on the same object is complete; or

2. A protected procedure or entry which is not part of another protected action on the same object is complete, followed by a call to `Next_Entry_Call` which returns an `E` parameter of `Null_Protected_Entry`. [1]

If called with `Simple_Call` or `Conditional_Call`, the compiler-generated code should call `Wait_For_Completion` after unlocking the object following a call to `Protected_Entry_Call`. For a conditional entry call, it should execute the sequence of statements of the entry call alternative if it returns with `Call_Successful=True`, and that of the **`else`** part if it returns with `Call_Successful=False`.

[1] The compiler-generated code may be able to determine that there are no further entries to be serviced (e.g., all barriers evaluate to `False` and neither `Protected_Entry_Call` nor `Cancel_Protected_Entry_Call` was called for this object since the last `Next_Entry_Call`), in which case it can unlock the object without calling `Next_Entry_Call`.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

9.10 Completing a Protected Entry Call

`Complete_Entry_Body` is used to inform the TRTS of the completion of an entry body.

9.10.1 RTS Responsibilities

`Complete_Entry_Body` wakes up the task waiting on the entry (if the call was pended with `Simple_Call` or `Conditional_Call`), or aborts the abortable part (if the call was pended with `Asynchronous_Call`). `Exceptional_Complete_Entry_Body` performs the same functions and also raises the indicated exception in the task that made the entry call. If the entry caller is executing the entry body, the exception will be deferred until it unlocks the associated `Protection` object.

The effect of calling either procedure is undefined unless the calling task is executing a protected action.

9.10.2 Compiler Responsibilities

The compiler shall assure that `Complete_Entry_Body` and `Exceptional_Complete_Entry_Body` is called only during a protected action.

9.10.3 Compiler Notes

`Complete_Entry_Body` should be called at the completion of an entry body. `Exceptional_Complete_Entry_Body` should be called in cases where the entry body raises an exception, providing the propagated exception to the TRTS in the argument `Ex`. It can also be called with an argument of `Null_Exceptions` in place of `Complete_Entry_Body`.

9.11 Waiting for a Protected Entry Call

`Wait_For_Completion` is used to wait for an entry call to complete or be cancelled, and to report on whether it completed or not.

9.11.1 RTS Responsibilities

If the call is queued, `Wait_For_Completion` blocks the calling task until the call completes (by calling `Complete_Entry_Body`), otherwise it returns immediately. On return, the parameter `Call_Successful` will be `True` if the call completed successfully, and `False` if it did not.

The effect is undefined if the calling task has not called `Protected_Entry_call` without an intervening call to `Wait_For_Completion`.

9.11.2 Compiler Notes

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

458 `Wait_for_Completion` is used to implement simple and conditional entry calls on
459 protected objects. It should be called after a call to `Protected_Entry_Call` with
460 where `Mode=Simple_Call` to block the calling task until the entry call is complete,
461 or `Mode=Conditional_Call` to report whether the call was successful. After a
462 conditional call, the sequence of statements of the entry call alternative should be
463 executed if `Call_Successful=True`; otherwise, the sequence of statments of the
464 **`else`** part should be executed.

465 9.12 Count Attribute

466 `Count` reports on the number of tasks queued on a protected entry.

467 9.12.1 RTS Responsibilities

468 `Count` returns the number of tasks that are waiting on the entry specified by the
469 entry index `E` in the object on which the caller is currently executed an protected
470 action. Note that pending calls are not included in this count.

471 The effect of calling this procedure is undefined unless the calling task is executing
472 a protected action, and `E` is valid for the corresponding protected object.

473 9.12.2 Compiler Responsibilities

474 The compiler shall assure that `Count` is called during a protected action.

475 The compiler shall assure that the `E` parameter is a valid entry index on the corre-
476 sponding protected object.

477 9.12.3 Compiler Notes

478 `Count` should be called to obtain the `Count` attribute for protected entries.

479 9.13 Propagating Exceptions

480 `Broadcast_Program_Error` propagates an exception to all tasks queued on the
481 entries of a protected object.

482 9.13.1 RTS Responsibilities

483 `Broadcast_Program_Error` shall raise `Program_Error` in all callers of the object in
484 which the caller is currently executing a protected action, and all calls to any entry
485 of the protected object shall be dequeued.

486 The effect of calling this procedure is undefined unless the calling task is executing
487 a protected action.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

9.13.2 Compiler Responsibilities

The compiler shall assure that `Broadcast_Program_Error` is called during a protected action.

9.13.3 Compiler Notes

`Broadcast_Program_Error` should be called when an exception is propagated out of the evaluation of an entry barrier. This raises `Program_Error` in all callers, as required by the Ada language. The task that calls this operation may or may not be a caller of the current object, so it must be prepared for `Broadcast_Program_Error` to raise `Program_Error` in the calling task.

9.14 Example of a Protected Object

An example of a protected object is:

```
protected type PT is
  procedure P;
  function F return Boolean;
  entry E1(Param: Integer);
  entry E2(1..10);
private
  B: Boolean:= False;
end PT;

protected body PT is

  procedure P is
  begin
    S1;
  end P;

  function F return Boolean is
  begin
    S2;
  end F;

  entry E1(Param: Integer) when B is
  begin
    S3;
  end E1;

  entry E2(for I in 1..5) when B or F is
  begin
    S4;
  end E2;

  entry E2(for I in 6..10) when F is
  begin
    S5;
  end E2;

end PT;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
529 PO: PT;
```

530 One of several equivalent ways to structure the compiler generated code and TRTS
531 calls for the above declarations is:

```
532 type PT is new Finalization.Controlled with record
533   Object: Protection(Num_Entries=> 11);
534   B: Boolean:= False;
535   end record;

536 procedure Initialize(PO: in out PT) is
537 begin
538   Initialize_Protection(PO.Object);
539 end Initialize;

540 procedure Finalize(PO: in out PT) is
541 begin
542   Finalize_Protection(PO.Object);
543 end Finalize;

544 type E1_Params is record
545   Param: Integer;
546 end record;
547 procedure P(PO: in out PT);
548 function F(PO: PT) return Boolean;
549 procedure Service_Entries(PO: in out PT);

550 ...

551 procedure P_Unprotected(PO: in out PT) is
552 begin
553   S1;
554 end P_Unprotected;

555 procedure P(PO: in out PT) is
556 begin
557   Lock(PO.Object);
558   P_Unprotected(PO);
559 at end
560   Service_Entries(PO);
561 end P;

562 function F_Unprotected(PO: PT) return Boolean is
563 begin
564   S2;
565 end F_Unprotected;

566 function F(PO: PT) return Boolean is
567   F_Temp: Boolean;
568 begin
569   Lock_Read_Only(PO.Object);
570   F_Temp:= F_Unprotected(PO);
571   return F_Temp;
572 at end
573   Unlock(PO.Object);
574 end F;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
575 procedure Service_Entries(PO: in out PT) is
576   P: System.Address;
577   subtype PO_Entry_Index is Protected_Entry_Index
578       range Null_Protected_Entry..11;
579   Barriers: Barrier_Vector(1..11);
580   E: PO_Entry_Index;
581 begin
582   loop
583     begin
584       Barriers(1):= PO.B;
585       Barriers(2..6):= (when others=> PO.B and F_Unprotected(PO));
586       Barriers(7..11):= F_Unprotected(PO);
587     exception
588     when others=>
589       begin
590         Broadcast_Program_Error;
591       exception
592       when Program_Error=>
593         Unlock(PO.Object);
594         raise;
595       end;
596     end;
597     Next_Entry_Call(Barriers, P, E);
598     begin
599       case E is

600       when Null_Protected_Entry=>
601         exit;

602       when 1=>
603         declare
604           Params: E1_Params;
605           for Params use at P;
606           Param:= Params.Param;
607         begin
608            S3;
609         end;

610       when 2..6=>
611         declare
612           type Entry_Index_Type is range 1..5;
613           I: Entry_Index_Type:= Entry_Index_Type(E - 1);
614         begin
615           S4;
616         end;

617       when 7..11=>
618         declare
619           type Entry_Index_Type is range 6..10;
620           I: Entry_Index_Type:= Entry_Index_Type(E - 1);
621         begin
622           S5;
623         end;

624       end case;
625     at end
626       if E /= Null_Protected_Entry then
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
627           Exceptional_Complete_Entry_Body(Ex=> Current_Exception);
628         end if;
629       end;
630     end loop;
631     Unlock(PO.Object);
632   end Service_Entries;

633   PO: PT;
```

634 This example implements the protected type as a controlled record containing the
635 state of the object. The TRTS state is represented by a `Protection` component,
636 and the private data is represented by additional fields in the record, here `B`. Con-
637 trolled type initialization and finalization are used to assure that the `Protection`
638 component is properly initialized and finalized by the TRTS.

639 Each protected subprogram takes the associated protected object as an implicit pa-
640 rameter. This example implements these directly with subprograms of the same
641 name, each of which takes an implicit parameter of type `PT`. This parameter con-
642 tains all of the state of the protected object.

643 `Barriers` represents an anonymous vector of barrier values associated with the
644 object. In the case of E1, the barrier is just a single boolean variable private to the
645 protected record, and all the barrier evaluation in `Service_Entries` accomplishes
646 is to copy `B` into the barrier vector. This is likely to be a common case, and an
647 important compiler optimization will be to simply allocate such a barrier variable as
648 the corresponding element in the barrier array. In this way, no evaluation would be
649 required; the correct value would already be in the barrier vector.

650 The identifier `Current_Exception` stands for the current exception, which the
651 compiler generated code is responsible for knowing, possibly with the help of the
652 exception handling portion of the RTS (The interface between the exception handler
653 code and the exception handling portion of the RTS is not part of GNARLI). If the **at**
654 **end** processing was not initiated by an exception, `Current_Exception` represents
655 `Null_Exception`.

656 The implementation of the `E1` entry body provides an example of how entry parame-
657 ters might be passed, using the `Uninterpreted_Data` parameter of `Next_Entry_-`
658 `Call`. The example provides a record type `E1_Params` with a field corresponding to
659 each parameter of `E1`. The case alternative within `Service_Entries` implementing
660 this body assumes that the `P` parameter returned by `Next_Entry_Call` the address
661 of a record of this type, with the actual parameter value in the `Param` field. It de-
662 clares an `E1_Params` object at that address and places the value of this field in a
663 local variable `Params` for use by the code sequence `S3`.

664 The procedure `P` and function `F` are then called as normal Ada subprograms. Note
665 that there are unprotected versions of these subprograms for use within the protected
666 object itself; an example of the use of these unprotected subprograms can be seen in
667 the evaluation of the barriers to `E2`.

668 Calling the entry `E1` is more complex. A simple call such as:

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
669 PO.E1(1);
```

670 might be translated as:

```
671 declare
672   P: E1_Params:= (Param=> 1);
673   S: Boolean;
674 begin
675   Protected_Entry_Call(
676       Object=> PO.Object,
677       E=> 1,
678       Uninterpreted_Data=> P'Address;
679       Mode=> Simple_Call);
680   PO.Service_Entries(PO);
681   Wait_for_Completion(Call_Successful=> S);
682 end;
```

683 `Protected_Entry_Call` pends the call on `PO`. It will be either executed or queued
684 by the call to `Next_Entry_Call` within `Service_Entries`. If the call is queued,
685 `Wait_for_Completion` will wait until this call has completed by proxy; if the call
686 has been completed in `Service_Entries`, it returns immediately. For a simple call,
687 `S` should always be true on return, and is ignored.

688 Conditional entry calls are very similar, except that the `Call_Successful` flag
689 is used to determine which alternative was selected. For example, the following
690 conditional call on `E2`:

```
691 select
692   PO.E2(3);
693   S6;
694 else
695   S7;
696 end select;
```

697 might be translated as:

```
698 declare
699   S: Boolean;
700 begin
701   Protected_Entry_Call(Object=> PO.Object,
702       E=> 4,
703       Uninterpreted_Data=> System.Null_Address,
704       Mode=> Conditional_Call);
705   PO.Service_Entries(PO);
706   Wait_for_Completion(Call_Successful=> S);
707   if S then
708     S6;
709   else
710     S7;
711   end if;
712 end;
```

713 The mode for `Protected_Entry_Call` is `Conditional_Call`, which prevents `Wait_-`
714 `For_Completion` from waiting; instead, it reports on whether the entry was serviced
715 by the call to `Service_Entries`. If so, the sequence of statements of the entry call

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

alternative is executed; otherwise, the **else** alternative is executed.

9.15 Requeuing an Entry Call

The Requeue_Protected_Entry operation is provided to requeue an entry call on a protected entry.

9.15.1 RTS Responsibilities

Requeue_Protected_Entry pends or queues the new entry call and returns. The effects of this operation is undefined if the calling task is not serving an entry call.

1. The New_Object parameter specifies the protected object containing the entry to be called.
2. The parameter E is the index of the entry on which the call is to be requeued.
3. The parameter With_Abort specifies that the requeued entry is to be abortable. Otherwise, abort will be deferred until the call is complete or requeued abortably.

If Requeue_Protected_Entry is called while serving a protected entry call, its effect is dependent on whether it is requeuing the call to the same or another protected object:

1. If the two objects are the same, the current call is added to the specified queue; otherwise
2. The current call is pended on the new object.

9.15.2 Compiler Responsibilities

The compiler shall assure that the task calling Requeue_Protected_Entry is serving an entry call. If the requeue is of a task entry call or of a protected entry call to a different protected object, the compiler-generated code shall lock New_Object before making this call.

9.15.3 Compiler Notes

The Requeue_Protected_Entry should be called to implement a **requeue** statement where the target is a protected entry. The With_Abort flag should be True if and only if the **requeue** contains a **with abort** clause.

After calling Requeue_Protected_Entry from an **accept** statement, the compiler-generated code should unlock New_Object after servicing its entries as described in Section 9.9.3 and leave the **accept** statement, bypassing the rest of its sequence of statements and calling Complete_Rendezvous. Similarly, after calling it from

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

an entry body, the compiler-generated code should leave both protected actions in LIFO order, bypassing the rest of the entry body. The protected entry to which the call is transferred should be either parameterless or must have a parameter profile compatible with that of the entry of the calling **accept** statement or entry body.

9.15.4 Example of the Requeue of a Protected Entry Call

Consider the following example, which contains entry bodies requeuing calls between and within protected objects:

```
protected body O1 is
...
entry E1 when B1 is
begin
  S1;
  requeue O2.E2;
  S2;
end E1;
...
end O1;

protected body O2 is
...
entry E2 when B2 is
begin
  S3;
  requeue E3 with abort;
  S4;
end E2;

entry E3 when B3 is
begin
  S5;
end E2;
...
end O2;
```

One way to translate this into GNARLI calls is:

```
type T1 is new Finalization.Controlled with record
  Object: Protection(Num_Entries=> 1);
  end record;

procedure Initialize(PO: in out T1) is
begin
  Initialize_Protection(PO.Object);
end Initialize;

procedure Finalize(PO: in out T1) is
begin
  Finalize_Protection(PO.Object);
end Finalize;

O1 : T1;

...
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
793 procedure Service_Entries(PO: in out T1) is
794   P: System.Address;
795   subtype PO_Entry_Index is Protected_Entry_Index
796       range Null_Protected_Entry..1;
797   E: PO_Entry_Index;
798   Barriers: Barrier_Vector(1..1);
799   Bypass_Completion: Boolean:= False;
800 begin
801   loop
802     begin
803       Barriers(1):= B1;

804     -- The following handler is not actually needed; there should be
805     -- no way for barrier evaluation to raise an exception in this object.
806     -- An optimizing compiler might not generate it.

807     exception
808     when others=>
809       begin
810         Broadcast_Program_Error;
811       exception
812       when Program_Error=>
813         Unlock(PO.Object);
814         raise;
815       end;
816     end;

817     Next_Entry_Call(Barriers, P, E);
818     begin
819       case E is

820       when Null_Protected_Entry=>
821         Bypass_Completion:= True;
822         exit;

823       when 1=>
824         S1;
825         Lock(O2.Object);
826         begin
827           Requeue_Protected_Entry(
828               New_Object=> O2.Object,
829               E=> 1,
830               With_Abort=> False);
831         at end
832           Service_Entries(O2);
833         end;
834         Bypass_Completion:= True;
835         goto End_of_E1;
836         S2;
837         <<End_of_E1>>
838         null;
839       end case;
840     at end
841       if not Bypass_Completion then
842         Exceptional_Complete_Entry_Body(Current_Exception);
843       end if;
844       Bypass_Completion:= False;
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
845       end;
846     end loop;
847     Unlock(PO.Object);
848   end Service_Entries;

849   ...

850   type T2 is new Finalization.Controlled with record
851     Object: Protection(Num_Entries=> 1);
852     end record;

853   procedure Initialize(PO: in out T2) is
854   begin
855     Initialize_Protection(PO.Object);
856   end Initialize;

857   procedure Finalize(PO: in out T2) is
858   begin
859     Finalize_Protection(PO.Object);
860   end Finalize;

861   O2 : T2;

862   ...

863   procedure Service_Entries(PO: in out T2) is
864     P: System.Address;
865     subtype PO_Entry_Index is Protected_Entry_Index
866         range Null_Protected_Entry..2;
867     E: PO_Entry_Index;
868     Barriers: Barrier_Vector(1..2);
869     Bypass_Completion: Boolean:= False;
870   begin
871     loop
872       begin
873         Barriers(1):= B2;
874         Barriers(2):= B3;
875       exception
876       when others=>
877         begin
878           Broadcast_Program_Error;
879         exception
880         when Program_Error=>
881           Unlock(PO.Object);
882           raise;
883         end;
884       end;
885       Next_Entry_Call(Barriers, P, E);
886       begin
887         case E is

888         when Null_Protected_Entry=>
889           Bypass_Completion:= True;
890           exit;

891         when 1=>
892           S3;
893           Requeue_Protected_Entry(
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
894             New_Object=> PO.Object,
895             E=> 2,
896             With_Abort=> True);
897         Bypass_Completion:= True;
898         goto End_of_E2;
899         S4;
900         end;
901         <<End_of_E2>>
902         null;
903       when 2=>
904         S5;

905       end case;
906     at end
907       if not Bypass_Completion then
908         Exceptional_Complete_Entry_Body(Current_Exception);
909       end if;
910       Bypass_Completion:= False;
911     end;
912   end loop;
913   Unlock(PO.Object);
914 end Service_Entries;
```

915 The translation calls `Complete_Entry_Body` only when the requeued call is complete,
916 since the entry call as a whole is not complete until that point. The first requeue is
917 an external call, for which the compiler generates code to enter and exit the target
918 object. `Requeue_Protected_Entry` returns after pending the call on the new object.
919 The following `Service_Entries` will either execute or queue the call. The second
920 requeue is an internal call, for which the compiler omits the code to lock `New_Object`.
921 `Requeue_Protected_Entry` queues the call to the current object.

922 9.16 Rationale

923 9.16.1 Protected Actions—The Eggshell Model

924 A protected object can contain protected procedures, functions, and entries. Tasks
925 can call these, resulting in a *protected action*, a synchronized operation on the pro-
926 tected object. Protected actions are often explained in terms of the *eggshell* of the
927 protected object. A task must enter the eggshell to perform a protected operation
928 and exit the eggshell when the operation is complete. Under priority scheduling, the
929 effective priority of a task in the eggshell will be that of the *ceiling priority* associated
930 with the protected object, and abortion is deferred for tasks executing in an eggshell.

931 Protected procedures and entries are permitted to change the state of the protected
932 object; tasks executing these actions must be alone in the eggshell. Tasks attempting
933 to enter the eggshell for one of these operations will be will not be able to proceed if it is
934 already occupied. Protected functions, on the other hand, are only permitted to read
935 the state of the object; there can be any number of tasks executing protected functions
936 simultaneously within the eggshell. Tasks can only proceed with a protected function
937 if there is no task inside executing a protected procedure or entry body. Protected
938 entry calls must enter the eggshell to evaluate the entry barrier. If it is `True`, the
939 entry body is executed; otherwise, the call is placed on the corresponding entry queue

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

to be serviced when the barrier becomes `True`. An important aspect of the eggshell model is that all eligible calls must be served before any other task can enter the eggshell.

9.16.2 Asynchronous Transfer of Control

The asynchronous **`select`** statement allows a task to execute a *triggering* statement (a delay or entry call) in parallel with an *abortable part*. If the abortable part finishes before the triggering statement starts, an attempt is made to cancel the triggering statement; cancellation can fail if the the the trigger has deferred abortion. If cancellation is successful, the asynchronous **`select`** completed; otherwise, it is complete on completion of the trigger. If the triggering statement finishes first, the abortable part is aborted. Unless the trigger is cancelled, the triggered statements are executed, possibly after an *Asynchronous Transfer of Control* (ATC) out of an executing abortable part.

9.16.3 Execution Models

9.16.3.1 Protected Entry Calls

Two models for the execution of protected entry calls are considered. The *self-service* model has the caller of the protected entry execute the entry body, whether it is executed immediately or after being queued. If the task is queued, it will later be awakened or interrupted (in the case of an asynchronous call) and returned to the eggshell to execute the entry body.

Arranging for a specific task to reenter the eggshell is difficult in some systems. For example, in the POSIX 1003.4a Threads specification (Pthreads), a reasonable operation for implementing the eggshell is the *mutex*, a mutual exclusion object which can be locked by a thread and is thereafter owned by that thread. It is not possible to pass ownership of a mutex to a specific task; the task owning it can only unlock it, allowing other tasks to contend for it. Some of these other tasks might not be queued entry calls with `True` barriers, but tasks attempting to enter the eggshell for some other protected operation. One of these latter tasks could get into the eggshell before all of the eligible entry calls were serviced, in violation of Ada 9X semantics.

For this reason, we also consider the *proxy* model of entry body execution. In this model, a calling task executes an entry body only if the barrier is true on entry. Otherwise, the call is queued and, if not canceled, executed by the task (the proxy) that causes the barrier to become `True` before it exits the eggshell.

This interface is designed to support the proxy model of execution. Since it does not require the passing of locks between threads, it is less demanding on underlying system support for mutual exclusion, and so should be efficiently implementable on a larger number of platforms. Its major disadvantage is an unbounded execution time on exit from the eggshell of a protected object. However, this assumes an unbounded cascade of entry calls becoming eligible to execute as the result of a single protected operation, which is likely to be rare in practice.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

9.16.3.2 Asynchronous Transfer of Control

Similarly, two models for the execution of an asynchronous **select** statement are considered. The *two-thread* model spawns a separate thread of control, with one thread executing the abortable part, and the other executing the entry call. On completion of the entry call, the thread that is executing the abortable part can be cancelled. When cancelled, the abortable part's finalization actions must be performed, either by the thread executing the abortable part (via asynchronous transfer of control) or by the thread executing the triggering statement.

If the proxy model is used, the triggering statement will be executed by one of the threads only in the case where the barrier is `True` at the time of the call. Otherwise, the thread spawned to execute the triggering statement will be blocked until the entry body is executed by some other task.

The *one-thread* model either executes the triggering statment immediately, or queues the asynchronous call and then continues with the execution of the abortable part. If protected entries are implemented using the self-service model, continuation of the triggering statement (i.e. by expiration of the delay or by the barrier expression becoming `True`) interrupts the abortable part. If the proxy model is used, the triggering statement may be executed in parallel with the abortable part by another task, with the abortable part being aborted only on the completion of the triggering statement. At the completion of the triggering statement, the caller executes the triggered statements; aborting the abortable part if necessary.

The two-thread model adds weight to the asynchronous **select**, since it requires a thread with its own stack to be created and destroyed. This can be expensive in both time and space. Also, the increased parallelism of the two thread model is limited by the proxy model of protected entry execution, since one of the threads will often be blocked until the entry body has been executed by another task. Finally, the need to treat multiple threads as one task complicates many other TRTS functions such as rendezvous. For these reasons, and because supporting both models while avoiding TRTS callbacks to user code is difficult, this interface is designed to support the one-thread model.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

This side is intentionally blank.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 10: Runtime Support for Storage Management

This section describes support for storage allocation provided by the TRTS.

10.1 Package Specification

```
with System.Storage_Elements;
package System.Storage_Allocation is

   procedure Allocate_Block(
       Storage_Address: out System.Address;
       Size_In_Storage_Elements: System.Storage_Elements.Storage_Count;
       Alignment: in System.Storage_Elements.Storage_Count);

   procedure Deallocate_Block(Storage_Address: System.Address);

   function Maximum_Storage return System.Storage_Elements.Storage_Count;

end System.Storage_Allocation;
```

10.2 RTS Responsibilities

`Allocate_Block` shall allocate `Size_In_Storage_Elements` storage units of contiguous storage, aligned according to `Alignment`, and return the address of the first storage unit in `Storage_Address`. None of the storage in this block will be allocated by further calls to `Allocate_Block` until `Deallocate_Block` is called with the value returned in `Storage_Address`. If there are insufficient resources to allocate the requested storage, `Storage_Error` shall be raised at the point of the call.

`Deallocate_Block` shall make the storage starting at `Storage_Address` available for further allocations using `Allocate_Block`.

`Maximum_Storage` shall return the maximum storage that can be allocated at any one time. If the RTS is unable to determine this size, `Maximum_Storage` shall return zero.

`Allocate_Block`, `Deallocate_Block`, and `Maximum_Storage` shall be synchronized with respect to tasking, that is, they will function correctly when called from multiple tasks, regardless of the relative timing of the various calls.

10.3 RTS Notes

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

The TRTS should use **new** to allocate all Ada objects not supplied as parameters from the compiler or allocated through elaboration of object declarations, in order to allow the compiler to perform the necessary initializations on the allocated objects. However, it should not depend on the default storage pool implementation provided by the compiler, as this storage pool may use TRTS calls, possibly leading to an infinite recursion in the TRTS. Instead, the TRTS should implement its own storage pool using the underlying operating system or machine services, and use this storage pool for all TRTS allocations. For example, this TRTS storage pool could be implemented with the operations defined in the `System.Storage_Management` package (though they need not):

```
with System.Storage_Pools;
with System.Storage_Elements;
package System.RTS_Storage_Pools is

   type RTS_Storage_Pool is new
       System.Storage_Pools.Root_Storage_Pool;

   procedure Finalize(Object: in out RTS_Storage_Pool);

   procedure Allocate(
       Pool: in out RTS_Storage_Pool;
       Storage_Address: out System.Address;
       Size_In_Storage_Elements: Storage_Elements.Storage_Count;
       Alignment: Storage_Elements.Storage_Count);

   procedure Deallocate(
       Pool: in out RTS_Storage_Pool;
       Storage_Address: System.Address;
       Size_In_Storage_Elements: Storage_Elements.Storage_Count;
       Alignment: Storage_Elements.Storage_Count);

   function Storage_Size(Pool: RTS_Storage_Pool)
       return Storage_Elements.Storage_Count;

end System.RTS_Storage_Pools;

with System.Storage_Allocation; use System.Storage_Allocation;
package body System.RTS_Storage_Pools is

   procedure Finalize(Object: in out RTS_Storage_Pool);
   begin
     null;
   end;

   procedure Allocate(
       Pool: in out RTS_Storage_Pool;
       Storage_Address: out System.Address;
       Size_In_Storage_Elements: System.Storage_Elements.Storage_Count;
       Alignment: in System.Storage_Elements.Storage_Count) is
   begin
     Allocate_Block(Storage_Address, Size_In_Storage_Elements, Alignment);
   end Allocate;

   procedure Deallocate(
```

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

```
73          Pool: in out RTS_Storage_Pool;
74          Storage_Address: System.Address;
75          Size_In_Storage_Elements: System.Storage_Elements.Storage_Count;
76          Alignment: in System.Storage_Elements.Storage_Count) is
77       begin
78         Deallocate_Block(Storage_Address);
79       end Deallocate;

80       function Storage_Size(Pool: RTS_Storage_Pool)
81           return System.Storage_Elements.Storage_Count is
82       begin
83         return Maximum_Storage;
84       end Storage_Size;

85    end System.RTS_Storage_Pools;
```

86 10.4 Compiler Notes

87 The compiler is responsible for implementing the default storage pool for allocations
88 using **new**. How it does this is not specified by the TRTS. The TRTS provides a
89 synchronized allocation service for this purpose, but the compiler need not use it.
90 It may decide instead to allocate all storage from the stack, using TRTS services to
91 provide the necessary synchronization. It may decided to use `Allocate_Block` to
92 allocate all of the storage for a pool, and manage the storage within this block itself,
93 again using TRTS synchronization. It may use a combination of these methods,
94 allocating storage for some pools on the stack and others using `Allocate_Block`.

95 10.5 Rationale

96 In cases where memory allocation is not synchronized, or where it is implemented
97 over a bare machine, the TRTS can still provide synchronization at least as easily
98 as the compiler can do so by using TRTS synchronization calls. On the other hand,
99 the TRTS will sometimes be built over multithreaded operating system services
100 which provide thread-safe (i.e. synchronized) memory allocation primitives, and
101 these may be the only allocation primitives available on the platform. In this case,
102 the only service available will be synchronized, whether or not GNARLI specifies it
103 as synchronized. In these cases, it would be wasteful for the compiler to assume
104 that memory allocation needed additional synchronization. For these reasons, the
105 GNARLI interface provides synchronized memory allocation to the compiler.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

106 *This side is intentionally blank.*

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 11: Compiler Support for Exceptions

The `System.Compiler_Exceptions` package provides an interface to the compiler's implementation of exceptions for the TRTS implementation. The procedures in this package are not part of the TRTS. The TRTS interface makes the implementation of exceptions and exception handling entirely the responsibility of the compiler. The package is either provided by the compiler developer or written by the TRTS builder to fit the exception-handling mechanism used by the compiler.

11.1 Package Specification

```
with System.Machine_Specifics;
package System.Compiler_Exceptions is
  type Exception_ID is private;
  Null_Exception: constant Exception_ID;
  Constraint_Error: constant Exception_ID;
  Program_Error: constant Exception_ID;
  Storage_Error: constant Exception_ID;
  Tasking_Error: constant Exception_ID;
  procedure Raise_Exception(E: Exception_ID);
  procedure Notify_Exception(
      Which: Machine_Specifics.Machine_Exceptions;
      Info:  Machine_Specifics.Error_Information;
      Modified_Registers: Machine_Specifics.Pre_Call_State);
  subtype Exception_ID_String is String(1..16);
  function Image(E: Exception_ID) return Exception_ID_String;
private ... -- compiler-defined
end System.Compiler_Exceptions;
```

11.2 Compiler Responsibilities

The `Exception_ID` type shall be used to identify exceptions at the TRTS interface. The implementation of this type shall be determined by the compiler developer. For each of the Ada language-defined exceptions that may need to be raised by the TRTS, a constant of the `Exception_ID` type shall be provided in this package, that identifies that exception.

The special value `Null_Exception` shall identify no exception. The `Raise_Exception` operation shall have the effect of raising the specified exception in the calling task. Calling `Raise_Exception` with an argument of `Null_Exception` shall have no effect.

The `Notify_Exception` call shall translate a machine exception into an Ada exception, if any, raising that exception in the effected task. If the machine exception

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

does not require the raising of an Ada exception, `Notify_Exception` shall have no effect. It must be called when a machine-specific condition arises that requires an Ada exception to be called. If called from within an error handler, it will raise the Ada exception at the point that the effected task was interrupted. If called from within a trap handler, this call may exit the handler, returning control to application code. This procedure has three parameters:

1. `Which` shall specify which hardware exception occurred. Its type, `Machine_Exceptions` need bear no particular relationship to `Exception_ID` or to any actual Ada exceptions. The values of this type need correspond to (only) the exceptions that are detected by hardware and need to be mapped to Ada. These values may include application-specific conditions that are detected by hardware, as well as conventional machine-specific error conditions. Some predefined Ada exceptions may not be represented (for example if the hardware does not detect overflow), or there may be several hardware exceptions (e.g, fixed-point overflow, floating-point underflow, etc.) that are mapped to a single exception (e.g., `Constraint_Error`) by the compiler's implementation of exceptions.

2. `Info`. This shall specify other machine-level information, that may be specific to the particular exception.

3. `Modified_Registers`. This shall specify the values of any registers that the TRTS needed to change from the values which they had at the time of the hardware exception in order to call `Notify_Exception`. This shall include the old program counter value at the time execution was interrupted.

The `Image` function returns a string that uniquely identifies the exception specified by the exception ID parameter, within the calling program. The exception implementation shall document a mechanism for using this string to determine the corresponding Ada exception declaration.

11.3 Compiler Notes

There are a number of ways exception handling may be implemented. They are all fairly complex, and it is not clear that there is one best way. The interface between the compiler's implementation of exceptions and the TRTS is therefore designed to hide all details of the implementation of exceptions from the TRTS. For interface purposes, all exceptions are assigned unique identifiers (e.g., integers) and the predefined exceptions are assigned standard identifiers known to the TRTS.

Ada 9X defines the package `System.Exceptions` which defines the `Exception_Occurrence` type. An Ada 9X exception handler can obtain the value of an exception occurrence, which specifies the exception that caused the handler to be invoked and other implementation-defined information related to the exception and the environment in which it was raised. This package defines the procedure `Reraise_Occurrence`, which takes an exception occurrence and raises it again.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

This interface can be used in place of `System.Compiler_Exceptions`. However, there may be problems with implementation-specific data attached to the occurrences. For example, when an exception is raised in an **accept** statement, it must be raised as well in the caller. If this exception occurrence is reraised in the caller, the information associated with it will be consistent with the environment where it was raised, which is in another task. This could be particularly disconcerting if the information contained, for example, a stack traceback.

However, `System.Exceptions` can be used as the basis for the implementation of `System.Compiler_Exceptions`. The compiler only needs to provide an implementation of `Raise_Exception` that can take an exception occurrence and raise a new occurrence of the same type.

11.4 RTS Responsibilities

The argument to `Raise_Exception` shall be valid values of the type `Exception_ID`. This means that they will either be one of the constant values defined in `System.Compiler_Exceptions`, or a value provided by the compiler as a parameter to some GNARLI operation.

11.5 RTS Notes

This interface has been designed so as to avoid unnecessarily restricting the implementation of exceptions and exception handling, which is considered to be entirely the responsibility of the compiler. However, under certain circumstances the TRTS is required to cause an exception to be raised in an Ada task. This requires an interface between the implementation of exception handling and the TRTS. During recovery from an exception, there may be need to propagate an exception to another task, or to abandon execution of a task.

`Raise_Exception` is provided so that the TRTS can raise an exception in a task that may have been interrupted, blocked, or have called the TRTS—by calling this procedure within the context of that task. For example, it is needed for the case where the exception specified by parameter `E` of `Exceptional_Complete_Rendezvous` must be propagated to a task that is blocked as caller in a rendezvous. Note that since the value of `E` is not known until run time, the TRTS cannot use a **raise** statement to do this.

The TRTS implementation should call `Notify_Exception` when a machine exception occurs that may require raising an Ada exception. It should be called when the error occurs, without allowing the task to execute further. `Notify_Exception` should be called with arguments provided by the underlying implementation, typically arguments to an error handler.

Note that TRTS interface requires that the implementation of recovery from exceptions take care when propagating exceptions to recover storage collections that are no longer needed, to call `Complete_Master` when exiting a master, and to call `Complete_Task` when abandoning execution of a task.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

11.5.0.3 GNARL Implementation Notes

The CARTS definition of `Raise_Exception` differs from that given above in that it had a `Pre_Call_State` parameter. This was intended for use where the TRTS had responsibility for context switching. A task would be marked as needing to raise an exception the next time that it ran. When the context was switched to this task, the context switch code would restore as much of the task state as possible and put the rest into a `Pre_Call_State` object. `Raise_Exception` would then be called to restore the remainder of the state and cause the exception to be raised. An example of state that might not be restored at this point is the program counter, since changing the program counter would cause the context switch code to loose control before it would be able to call `Raise_Exception`.

In general, the GNARLI will be implemented using the GNULLI specification for low-level tasking support. This package hides the context switch from the GNARLI implementation, so there is no mechanism for interposing a call to `Raise_Exception` in the context switch. Instead, a task can use polling to determine if it should raise an exception. For this purpose, it is only necessary to raise an exception in the current context, so the GNARLI specification of `Raise_Exception` has only a single parameter specifying the exception to be raised.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Section 12: Acknowledgements

The preparation of this report was funded in part by the Air Force, contract F08630-92-C-0019, through the GNAT project; and by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM, contract DAAB07-91-D-Q502/D.O. 0006/0017.

Portions of this document are reused from drafts of earlier runtime system interface definitions written by T.P. Baker. These are the Corset interface [3, 4], the ARTEWG Model Runtime System Interface [5], and the CARTS Software Interface Specification [2].

The MRTSI was originally drafted by T.P. Baker, then developed further by the Ada Run Time Environment Working Group (ARTEWG) of the Special Interest Group for Ada (SIGAda) of the Association of Computing Machinery (ACM), and subsequently published in *Ada Letters*. The CARTS document was written by by T.P. Baker for Westinghouse Electric Corporation and the Avionics Laboratory, Wright Laboratory, Aeronautical Systems Division, United States Air Force. Portions of the MRTSI document were reused, with permission of the ARTEWG.

The technical contributions of the ARTEWG members who worked on the MRTSI document are gratefully acknowledged, as are those of the CARTS team members who reviewed the CARTS document. The Avionics Laboratory and Westinghouse Electric Corporation are gratefully acknowledged for granting permission to reuse portions of the CARTS document.

Thanks are also due to Offer Pazy, for his many helpful comments and suggestions.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Bibliography

The following documents are related to this proposal in an informative manner, or were used as references in the preparation of this proposal.

[1] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual: Draft Version 3.0*, June 1993. Available by anonymous FTP from ajpo.sei.cmu.edu.

[2] Ted Baker. Requirements specification for the Common Ada Run-Time System. Prepared for Westinghouse Electric Corporation, February 1991.

[3] T.P. Baker. A corset for Ada. Technical Report TR-86-09-06, Computer Science Department, University of Washington, 1986.

[4] T.P. Baker. An improved Ada runtime system interface. Technical Report TR 86-07-05, Computer Science Department, University of Washington, Seattle, July 1986. subsequently published in *Journal of Systems and Software 8*, Elsevier (1988) 373-393.

[5] Ada Runtime Environment Working Group: MRTSI Task Force. A model runtime system interface for Ada. Technical report, ACM SIGAda, 1990.

[6] United States Department of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD-1815A-1983.

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Alphabetic Topical Index

A

Abortion
exception ... 4, 20–22, 28–29, 54, 61, 71, 75
Abortion
procedure ... 22
Abort_Tasks
procedure ... 22, 72
Accept_Call
procedure ... 63–65, 72
Accept_Simple
procedure ... 26, 41, 56
Accept_Trivial
procedure ... 63–65, 72
Activate_Tasks
procedure ... 32–33, 35–37, 40, 72
Activation_Chain
type ... 31–32, 35, 38
ADDRESS
type ... 4
Address
type ... 8–9, 33–34
Allocate_Block
procedure ... 93, 95
Any_Priority
subtype ... 13, 32–33
Any_Priority
type ... 32
Asynchronous_Call
constant ... 54, 73–74
Asynchronous_Call
enumeration literal ... 51–52, 73, 78
Attach_Handler
pragma ... 70
B
variable ... 83
B
... 83

B

Boolean
type ... 4
Broadcast_Program_Error
procedure ... 77, 79–80

C

Calendar
package ... 1, 13, 45
Callable
attribute ... 2, 41
Callable
exception ... 66
Callable
function ... 66
Callable
procedure ... 66
Callable
type ... 66
Call_Simple
procedure ... 52, 72
Call_Successful=
procedure ... 77
Cancel_Protected_Entry_Call
procedure ... 41, 72–75, 77
Cancel_Task_Entry_Call
procedure ... 41, 53–54, 72–73
Clock
function ... 13, 45
Compiler_Exceptions
package ... 99
Complete_Activation
procedure ... 40–41, 43, 72
Complete_Entry_Body
procedure ... 20, 56, 72, 75, 77–78, 89
Complete_Master
exception ... 29
Complete_Master
procedure ... 24–29, 32, 36, 40–42, 44, 54, 72, 74, 99
Complete_Master_Object
variable ... 28
Complete_Rendezvous
procedure ... 20, 26, 41, 54, 56–58, 61, 64–65, 72, 85
Complete_Task
procedure ... 22, 24–25, 27, 36, 41–43, 72, 99
Conditional_Call
constant ... 73
Conditional_Call
enumeration literal ... 51–52, 77–79, 84
Conditional_Call
procedure ... 77
Count
attribute ... 2, 65, 73–74, 79
Count
function ... 65, 79
Count
procedure ... 79
Create_Task

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

procedure ... 15, 24–26, 31–34, 36, 50–51, 72
Current_Exception
procedure ... 61
Current_Master
function ... 24–25
Current_Master
procedure ... 25

D

Deallocate_Block
procedure ... 93
Default_Priority
constant ... 13, 32
Defer_Abortion
procedure ... 21, 29, 57
Delays
package ... 45–46
Duration
type ... 4, 14, 45
Duration'Small
constant ... 14
E
function ... 56

E

E1_Params
type ... 83
Else_Mode
pragma ... 57–58, 61
End_Rendezvous
procedure ... 61
Enter_Master
procedure ... 24–26, 29, 33, 40, 44, 54, 72, 74
Error_Information
type ... 7
Exceptional_Cancel_Protected_Entry_Call
procedure ... 4, 72
Exceptional_Cancel_Task_Entry_Call
procedure ... 4, 72
Exceptional_Complete_Entry_Body
procedure ... 4, 56, 72, 77–78
Exceptional_Complete_Rendezvous
procedure ... 26, 41, 56–58, 64–65, 99
Exceptional_End_Rendezvous
procedure ... 4
Exception_ID
type ... 16, 97–99
Exception_Occurrence
type ... 98
Exceptions
package ... 98–99
Expunge_Unactivated_Tasks
exception ... 37
Expunge_Unactivated_Tasks
procedure ... 15, 32, 36, 72
Expunge_Unallocated_Tasks
procedure ... 37
F
function ... 83

F

False
constant ... 33–34, 41, 51, 53, 56–57, 59, 66, 71–73, 77–78
Finalize_Protection
procedure ... 69–70

G

H

I

Image
function ... 16, 98
Initialize_Protection
procedure ... 69–70
Init_State
type ... 5–7
Integer
type ... 4
Interrupt_Handler
pragma ... 70
Interrupt_ID
type ... 7
Interrupt_Info
type ... 7
Interrupt_Priority
pragma ... 70
Interrupt_Priority
subtype ... 13

J

K

L

Local_Chain
procedure ... 38
Lock
procedure ... 21, 41, 70–73
Lock_Read_Only
procedure ... 41, 70–71

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

M

Machine_Exceptions
type ... 7, 98
Master_ID
type ... 24–25
Max_ATC_Nesting
constant ... 19–20
Maximum_Storage
function ... 93
Modified_Registers
procedure ... 98

N

New_Object
procedure ... 85
Next_Entry_Call
procedure ... 56, 72–73, 76–77, 83–84
No_Others
pragma ... 5, 20
No_Rendezvous
constant ... 57
Notify_Exception
procedure ... 97
Notify_Exception.
procedure ... 98
Notify_Exception
procedure ... 98–99
Null_Body
constant ... 57–58, 61, 65
Null_Body
... 58
Null_Exception
constant ... 61, 65, 83, 97
Null_Exceptions
constant ... 78
Null_Protected_Entry
constant ... 77
Null_Task
constant ... 15–16, 36, 43, 66
Null_Task_Entry
constant ... 56, 58–59

O

P
procedure ... 83
P
... 61, 63

P

Param
... 83
Params
... 83
PO
... 84
Pre_Call_State
type ... 7, 100
Priority
generic unit ... 34
Priority
pragma ... 70
Priority
subtype ... 13
Program_Error
exception ... 17, 33, 35, 51, 70, 72–73, 77, 79
Program_Error
package ... 11
Program_Error
procedure ... 80
Protected_Entry_Call
procedure ... 19, 41, 72–77
Protected_Entry_call
procedure ... 78
Protected_Entry_Call
procedure ... 79, 84
Protected_Entry_Index
type ... 68–69
Protection
procedure ... 71, 74, 77, 83
Protection
type ... 68–70, 74, 78, 83
Protection_Object
procedure ... 73
PT
type ... 83
Q
... 60–61

Q

R

Raise_Exception
procedure ... 97, 99–100
Real_Time
package ... 13, 45, 76
Requeue_Protected_Entry
procedure ... 85, 89
Requeue_Task_Entry
procedure ... 55–56
Reraise_Occurrence
procedure ... 98
S
... 84

S

Same_Task
function ... 16

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

Select_Index
type ... 59
Selective_Wait
procedure ... 26, 41, 46, 56–59, 61, 63–65, 72
Select_Modes
type ... 58
Self
function ... 16
Self
procedure ... 16
Service_Entries
procedure ... 83–84, 89
Simple_Call
constant ... 73, 77
Simple_Call
enumeration literal ... 51–52, 78–79
Simple_Mode
pragma ... 57–58
Some_Time_Type
package ... 46
Standard
package ... 13
Storage_Address
procedure ... 93
Storage_Error
exception ... 33, 93
System
package ... 4, 8–9, 13, 32–34, 45, 76, 98–99
System.Abortion
package ... 4, 9, 19
System.Compiler_Exceptions
package ... 97, 99
System.Delays
package ... 9
System.Machine_Specifics
package ... 5
System.Priority'Last
constant ... 70
System.Protected_Objects
package ... 9, 67
System.Rendezvous
package ... 9, 49
System.Storage_Allocation
package ... 9
System.Storage_Management
procedure ... 94
System.Task_IDs
package ... 9
System.Task_Stages
package ... 9, 15, 23
T
variable ... 52

T

Task_Entry_Call
procedure ... 19, 41, 50–55, 72
Task_Entry_Index
type ... 50
Task_ID
type ... 15–16, 33
Tasking_Error
exception ... 11, 35, 41, 51, 59, 61
Task_Storage_Size
type ... 7
Terminated
attribute ... 1, 42–43
Terminated
function ... 43
Terminated
procedure ... 43
Terminate_Mode
pragma ... 57–58
Terminate_Task
procedure ... 43
Text_IO
package ... 1
Ti
procedure ... 40
Tick
constant ... 13
Ti_ID
variable ... 40
Time
type ... 76
Time_Of
function ... 46
True
constant ... 34, 42–43, 51, 53, 56–58, 65–66, 71–73, 76–79, 85, 89–91

U

Unchecked_Conversion
function ... 6
Undefer_Abortion
procedure ... 21–22, 29
Unlock
procedure ... 22, 41, 70–71
Unspecified_Priority
constant ... 32–34, 70

V

valid, activation chain...31
valid, entry index...50, 68
valid, master ID...24
valid, protected object...70
valid, task id...15

W

Wait_For_Completion
procedure ... 73, 77–78
Wait_for_Completion

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.

procedure ... 79
Wait_For_Completion
procedure ... 84
Wait_for_Completion
procedure ... 84
X
... 63

X

Y

Z

Copyright ©1993 Florida State University. All rights reserved.
This is a working draft, subject to change.