Ada 9X
Implementation Modules II-IV

September 2, 1993

Ada 9X Mapping/Revision Team
Intermetrics, Inc.
733 Concord Avenue
Cambridge, Massachusetts 02138

(617)661-1840

Published by
Intermetrics, Inc.
733 Concord Avenue
Cambridge, Massachusetts 02138

This report has been produced under the sponsorship of the Ada 9X Project Office under contract F08635-90-C-0066.

1. Type Extension and Tagged Universal Types

This chapter describes implementation changes associated with type extension, universal types, and tags associated with universal types. Type extension involves specifying new discriminants, adding record components, or adding enumeration literals as part of type derivation. Run-time dispatch involves the use of a tagged universal type, which is implicitly convertible from any member of a "class" of types. Operations select the appropriate implementation based on the underlying the tag of the actual parameters.

We are interested in feedback on the following issues:

- Clarity, completeness, and consistency of the semantic specification.
- The feasibility of implementation, and possible adjustments to the implementation model.

From the user teams we are interested in whether these changes do in fact allow the use of an object-oriented programming style.

The proposal in this document is somewhat reduced from the proposal in the mapping document. This reflects the mapping team's efforts to reduce the overall scope of the Ada 9X change.

1.1 Basic Concepts

We begin by summarizing concepts from Ada 83, and then proceed to introduce new terms and concepts for Ada 9X.

Every value in Ada that is specified by an expression and every object that is denoted by a name has a single type, determinable at compile-time. If overload resolution cannot resolve an expression or name to a single interpretation with a well-defined type, then the expression or name is considered undefined or ambiguous.

The type of a value or object determines the set of operations which are visible for the value or object. Other compile-time or run-time considerations can determine whether the operation is correct (i.e. not a compile-time or run-time error) based on the scope, parameter mode, array bounds, or discriminants of a value or object, or whether a scalar value falls within a specified range, etc.

Each object has a specific subtype, which determines any additional constraints which apply to the object over and above the fundamental requirements associated with the type. Each value does not have a single specific subtype, but may belong to many subtypes, determined by whether it satisfies the constraints associated with the subtypes.

A derivative of an ancestor type is a type derived directly or indirectly from the ancestor type, through a series of one or more derived type declarations. For a parent type declared in the visible part of a package, then any derived types of that parent type declared within that visible part are called *premature derivatives*.

All types are classified as either composite or elementary. The classes which make up composite types are record, array, protected record, and task. Scalar and access types are considered elementary. Private types are considered composite outside the package where they are defined. Within that

package they are classified according to their full type.

1.1.1 Type Extension

The first new concept to be implemented in this module is that of an *extension*. An extension of an ancestor type is a derivative where one of the intervening derived type declarations defined additional discriminants, record components, or enumeration literals.

Type extension will be indicated in an Ada program by the following syntax:

```
derived_type_definition ::= . . .
  | NEW subtype_indication  WITH [LIMITED] PRIVATE
  | NEW subtype_indication  WITH record_type_definition
  | NEW subtype_indication  WITH enumeration_type_definition

type_conversion ::=
  type_mark ( expression {, simple_name => expression } )
```

The syntax for other possible extensions, which need not be implemented as part of this module, are listed in Section 3.4.2 of the Mapping Document.

1.1.1.1 Record Type Extension

A record or private type may be extended with a record or private type definition. In the private part, a record extension must be given for each type extended with a private extension in the visible part.

A record extension is treated approximately as though the new type were a record with an implicit initial component, referred to in this document as *.parent* (*.parent* is not a proposed language feature), consisting of an instance of the parent type. If the parent type is itself a record type, then the additional components are defined to follow any of the components of the parent type (e.g. in a positional aggregate).

The *primitive operations* of the parent type are available on the extended type (unless hidden), much like the operations of a designated type are available on an access type. In other words, an implicit *.parent* is supplied as necessary to make the operation meaningful.

Explicit conversion from an extended type to the parent type is equivalent to selecting the *.parent* component. Conversion to a subtype of the parent type may require a constraint check as well.

Explicit conversion from a parent type to an extended type allows additional components to be specified using named notation within a type-conversion construct. Any component which is not specified will be initialized by the default expression specified in its component declaration. The component must be specified explicitly if no default appeared in its declaration.

When a subprogram is derived for an extended type, calls on the subprogram implicitly select *.parent* for all parameters (of any mode). Inherited functions which return the parent type must be overridden in the package where the derived type is declared.

1.1.1.2 Enumeration Type Extension

Only a visible enumeration type may be extended with additional enumeration literals. The additional enumeration literals are considered to have position numbers starting with one more than the last position number of the parent enumeration type.

Explicit conversion from an extended enumeration type to the parent type involves a constraint check to ensure that the value is within the range of the parent. Explicit conversion from a parent type to an extended enumeration type requires no check, though it may involve a representation change.

1.1.1.3 Discriminant Extension

Types other than elementary types may be extended with a new set of discriminants. The new type either specifies a whole new set of discriminants or it inherits exactly the discriminants of the parent type.

The value of a discriminant is implied by the discriminant/bound of the parent type if the discriminant is used to define the parent discriminant/bound. Such discriminants act more as renames than as extensions to the type.

Example:

```
type Matrix(Num_Rows, Num_Columns : Positive) is
  array (1..Num_Rows, 1..Num_Columns) of Float;
    -- Discriminated array type,
    -- discriminants implied by bounds of array

type Square_Matrix(Rank : Positive) is new Matrix(Rank, Rank);
  -- Derived type, with only one discriminant, implied by
  -- the discriminant of the parent type.

type Vehicle is ...;
-- Some type describing all vehicles

type Automobile(Number_Of_Doors : Short_Integer) is
   new Vehicle with ...;
  -- Derived type, specialized to automobiles, with a totally
  -- new discriminant for the number of doors.
```

If a new set of discriminants is specified, then they may be used to constrain the parent type in the same way component subtype indications may depend on discriminants. In particular, they may only appear alone -- not part of a larger expression, and only within index or discriminant constraints.

1.1.2 Classes and Universal Types

A *class* of types is defined to be a set of types with common operations. In particular, a type, the *root type*), and all of its proper derivatives form such a class, the class *rooted* at the type.

A *universal type* for a class is one which, in certain contexts, allows implicit conversion to and from the types in the class, subject possibly to constraint and/or scope checking. For every type T declared in the visible part of a package, a corresponding universal type for the class rooted at T is implicitly declared in the same visible part, with the same set of operations as the type T. This universal type is denoted by the name T'CLASS.

A *tagged type* is a type with a run-time type tag to identify its values. If the root type of a class is declared to be a tagged type, or is derived from a tagged type, then the universal type for the class is a *tagged universal type*, meaning that each instance carries the type tag of the non-universal type from which it was converted. This also informs the compiler that run-time dispatch should be used with T'CLASS, requiring the allocation and initialization of a run-time type descriptor for T and each type later derived from T.

New syntax for declaring tagged types and indicating universal types:

type_name ::= ... | type_mark'CLASS

type identifier [(discrim_part)] **is tagged** type_definition;

1.2 Type Conversions

1.2.1 Conversions Between Related Types

For both record and enumeration extensions, explicit conversion between two extensions of the same type is defined to be equivalent to first converting to their nearest common ancestor type, and then converting to the target type.

Any component without default initial expressions, and not implied by the underlying discriminants/bounds of the parent type, must be specified as part of an explicit conversion from a parent type to an extended type. A constraint check may be required after converting to an extended type with new discriminants.

1.2.2 Implicit Conversion

In certain contexts types within a class are implicitly convertible to and from the universal type for the class. Generally, an implicit conversion is never applied if an interpretation can be found without applying the implicit conversion. The implementation module on Operator Visibility defines the rules regarding when implicit conversions may occur. The legality of all conversions between universal and non-universal types is discussed below.

Implicit conversions to or from a universal type are allowed when calling explicitly declared operations of the universal type, as well as when calling operations of other types. No implicit conversion may be applied to an operand or result of a universal type as part of an implicitly declared operation of the universal type (note that this disallows implicit conversion as part of a dispatching operation since all dispatching operations are implicitly declared).

These implicit conversion rules are constructed so that the Ada 83 rules for implicit conversion of universal integer and real follow as a consequence. Furthermore, constructs in Ada 83 which describe an operand as being "of any real type" or "overloaded on all integer types" may be replaced in Ada 9X with simply universal real or universal integer, respectively, with the appropriate effect.

For example, the specification for 'VAL of a discrete type could be written as:

function Discrete'VAL(Position : Root_Integer'CLASS) return Discrete;

where "Root_Integer'CLASS" is intended to be the same thing as universal integer.

1.2.3 Conversion To and From Universal Types

A type within a class may be converted (implicitly or explicitly) to the universal type for the class, except when the source type violates certain requirements, as follows:

- The source type is defined at a deeper level of nesting than the universal type, and the type is tagged.
- The source type overrides a non-dispatching operation of the universal type, and the type is tagged.

A universal type for a class may only be converted to a subtype within the class if the value satisfies the constraints associated with the target subtype.

For a composite type, no components may be added when converting from a universal to a non-universal type.

1.3 Operations

When the universal type T'CLASS is not tagged the primitive operations of T are carried over to T'CLASS without change, other than the systematic replacement of T with T'CLASS (just like for a derived type).

When T'CLASS is tagged, each user-defined primitive operation of T is implicitly overridden with a dispatching operation, with the operands originally of type T becoming operands of type T'CLASS and controlling the run-time dispatch. If the result is of type T, then the result type becomes T'CLASS, and the context of a call may be used to control the run-time dispatch. The parameters of a tagged universal type to a dispatching operation are used to select the actual subprogram to call. If the operation has multiple parameters of T'CLASS, then at run-time a check is made to ensure that all parameters specify the same tag. CONSTRAINT_ERROR is raised if they do not, except in the case of the "=" (and "/=") operators, which indicate inequality if the tags differ.

Explicit primitive operations for T'CLASS may also be declared in the visible part of the package where T is declared. These augment the implicit operations declared as part of the implicit derivation from T. These operations are not dispatching.

1.3.1 Attributes

In Ada 9X, users may define attributes of a type in the visible part of the package where the type is declared. For example:

```
function T'Test returns Boolean;
```

This function is a primitive operation of type T, and is inherited by types derived from T, and may be overridden in types derived from T. User defined attributes defined on types may also be applied to objects of type T'CLASS. In this case the tag of the object controls the selection of the attribute function

called. (See 1.6.4)

```
type Window is tagged ... ;
function Window'DEFAULT_LOCATION return Window_Location;

-- In another package ...
type Fancy_Window is new Window with ...;
function Fancy_Window'DEFAULT_LOCATION return Window_Location;

-- In the body of some other function...

Win: Window'Class := .... ;
-- This could be initialized to a Window or a Fancy_Window

if Win'DEFAULT_LOCATION = Origin then ...;
```

A new predefined attributes is added to the language.

'TAG	is defined for all tagged non-universal types and for objects of universal tagged type. When a applied to a type, the attribute yields a unique tag associated with that type, of type SYSTEM.Tag. When applied to an object the attribute yields the tag associated with the underlying non-universal type of its value.

1.3.2 Membership

The membership operator **in** is defined for objects of universal type.

```
-- Assume X : T'CLASS

X in T1'CLASS   (1)
X in T1         (2)
```

(1) is true if and only if X can be converted to T1 without raising a constraint error (See 1.2.3).

For a tagged X, (2) is true if and only if X'TAG equals T1'TAG and X satisfies the constraints of T1.

For an untagged X, (2) is legal when X **in** T1'CLASS is TRUE, and is true if X satisfies the constraints of T1.

1.4 Declared and Allocated Objects

In Ada 9X the type of a declared variable may be unconstrained, in which case the variable is constrained by its initial value. This is essentially the way constant objects work now. Tagged universal types are considered unconstrained as far as declared objects are concerned.

1.4.1 Untagged Universal types

Objects of untagged universal types behave as if they had been declared to be the root type of the class, except that the implicit conversion rules also apply to such objects.

1.4.2 Tagged Universal Composite Types

For a tagged composite type T, each object of type T'CLASS must be explicitly initialized, and is constrained thereafter to hold only values with tag and discriminants, if any, matching those of its initial value. An attempt to change the tag or discriminants of such an object, by assigning to the whole object, raises CONSTRAINT_ERROR.

1.4.3 Tagged Universal Elementary Types

For a tagged elementary type T, each variable of type T'CLASS may be reassigned to hold any possible value of the universal type. If not explicitly initialized, such a variable has the tag for T as its initial tag. In other words, such objects are not constrained by their initial tag.

1.5 Universal Integer and Universal Real

Universal integer is the universal type for the integer class of types. In Ada 9X package STANDARD declares an anonymous integer type capable of representing the longest supported integer type. Package SYSTEM will declare a renaming (subtype) of the universal type rooted at that anonymous type. This type provides a run-time representation of *universal integer*. This subtype should be named ANY_INTEGER. Similarly there should be an ANY_REAL, which is the run-time representation of *universal real*.

To avoid confusion with universal types, we propose the term *large_real* instead of *universal fixed* to represent the result of a fixed-point multiply or divide.

1.6 Implementation Models

1.6.1 Composite Types

The tagged universal composite type may be considered a discriminated type, with the first discriminant being the type tag (a reference to a run-time type descriptor). A tagged universal composite type has an unknown number of additional discriminants and components, and therefore is inherently an "unconstrained" type, though all instances are constrained to a particular underlying non-universal type for their lifetime. This implies that tagged composite T'CLASS is illegal as a component subtype indication, but is allowed as a formal parameter, a result type, and as an initialized declared object.

1.6.2 Enumeration Types

The model for a tagged universal elementary type is a discriminated type with a default for the one discriminant, and one component of a size corresponding to the maximum size non-universal elementary type. The default for the discriminant identifies the root type for the class. This implies that that tagged universal types may appear in a component subtype indication, as well as in a formal parameter specification or elementary object declaration.

1.6.3 Tags

The model of a tag is an access type designating a type descriptor. A type descriptor contains an array of accesses-to-subprogram, where the designated subprograms are the set of dispatchable operations for the type. A derived type's descriptor can be thought of as an extension of the parent type's with additional fields corresponding to additional derivable operations.

In order to efficiently implement the membership operator, the type descriptor should also contain an indication of the distance from the type to the root-type of the largest tagged class containing this type (i.e., the ultimate tagged ancestor of the type), and an array containing the tags of each ancestor type.

1.6.4 Run-Time Dispatch

A run-time dispatch requires a lookup through the descriptor, and an indirect call of the appropriate operation. All the subprograms which can participate in a given dispatch must have the same calling convention. Subprograms which might participate in a dispatch, that is, all user-defined derivable operations of a tagged type, should make no assumptions about constraint checks that may have been performed by the caller, and assume that their parameters are passed in the form of the universal type. Compilers may wish to optimize this situation by providing two bodies to such procedures. One that can be called when the operation is known statically, and the other that performs constraint checks on the parameters, that can be called via a dispatch.

The language rules require all parameters that can potentially control a dispatch to have the same tag. The only exception to this are the "=" and "/=" operations.

1.6.5 Record Layout

For tagged records, the implicit *.parent* component must always be allocated at offset 0. For untagged records this is not required, since a conversion to universal type may discard parts of the record.

If there are subcomponents of dynamic size, then record layout becomes more complex. For untagged records, the straightforward implementation is to always use a level of indirection to indicate the *.parent* component if the parent type has dynamic size.

For tagged types, the *.parent* must always be located at offset 0, so it becomes reasonable to pre-allocate a reference to a possible extension when allocating the parent type. A likely way to do this is to have the size of the record located at a known offset within all tagged records whose size may be dynamic. With this scheme, the extension part of a record can be accessed via a level of indirection from the parent. Obviously this approach could be used for untagged types as well, but it causes extra memory to be allocated in all dynamically sized records, just to deal with the possibility of record extension, which should be rare for untagged types.

1.7 Implementation Strategies

The following sections outline the steps which must be performed at various points in the compilation process in order to implement the features described above.

1.7.1 Type Declarations

The new actions required for processing an explicit declaration of type T are:

1. Implicit declaration of the universal type T'CLASS

The new actions required for processing a type extension T1 derived from T are:

1. Creating an instance of T1 in the symbol table with a representation distinct from T, (This mechanism may already exist to support representation clauses)
2. Marking T1 limited if the extension part of T1 has limited elements.
3. Laying out the record extension fields when either the parent or the child is dynamically sized using the appropriate algorithm (see above).

The new actions required for processing a tagged type T are:

1. Marking in the symbol table that the type is tagged.
2. Constructing functions to compute the size, and to perform a constraint checked assignment.
3. Initializing a descriptor for the type after all the primitive operations are known (only for non-universal types).

1.7.2 Subprogram Declarations

New actions required for processing a primitive operation S of a non-universal type T are:

1. Allocating space in the type descriptor for S if T is tagged.

1.7.3 Object Declarations

When declared objects are marked "(aliased)" they must have a tag allocated along with the data. These situations are discussed in Section 3.2 of this document.

1.7.4 Allocators

Allocators for tagged types must allocate enough space for both the tag and the data, the allocated object must be initialized with the tag.

1.7.5 Overload Resolution

Overload resolution is discussed in detail in the *Operator Visibility* implementation module. There are now more types which support implicit conversion, and users are now able to declare functions with parameters that support implicit conversion. The new overload resolution rules specify how to deal with implicit conversions when selecting an interpretation for an expression.

1.7.6 Assignment Statements

When assigning to or from a universal type, the compiler must check that the other object is implicitly convertible to/from the universal type, and emit the appropriate conversion. If the type is tagged composite, then code to raise constraint error if the source and target have different tags must be

generated. Furthermore, the sizes and discriminants must be checked as well, and this must be done out of line by dispatching to a compiler generated procedure that performs these checks. It's probably most efficient for this procedure to perform the assignment as well.

1.7.7 Subprogram Calls

At a dispatching operation the compiler must

1. Determine which objects or non-dispatching functions provide controlling tags. (In the case of a dispatching function with no controlling operands this is can be complex, see below.)
2. Emit code to compare the controlling tags and raise constraint error if they are not identical.
3. Call the subprogram indicated by the controlling tags.

Perhaps the most difficult problem that must be solved by a compiler is finding the controlling tag for a subprogram, whose result type controls the dispatch, and which does not have any controlling parameters. In one sense this is a pathological case; it's object oriented programming without the object. On the other hand it appears to be a useful capability. For example:

Assume that SET implements some set abstraction. It has several children, each of which have very little in common as to representation. All must override the funtion EMPTY, which returns and empty set. The following fragments illustrate some of the difficulty in determining the controlling tag to each call to EMPTY.

The tag of MY_SET controls the call to EMPTY.

```
if MY_SET = EMPTY then
```

Here we assume that INTERSECT and UNION are dispatching operations. The tags of MY_SET, and YOUR_SET must match, and the call to empty uses that TAG as it's controlling TAG.

```
UNION(INTERSECT(MY_SET,YOUR_SET),EMPTY)
```

In general, if a controlling tag is not a parameter, then it must be gotten from consumer of the result. If that subprogram/operation specifies the tag to use directly, then that can be used. Otherwise the other parameters of that operation must be examined. If a tag can be inferred from those parameters, then they are used. If not, then a tag must be inferred from the parent's result (so the algorithm recurses.) If no tag can be determined, then the tag associated with the root type of the class is used.

It is not clear to us that the above algorithm is feasible to implement in existing compilers. We are interested in feedback, and in possible simplifications that are easier to implement.

1.7.7.1 Testing Equality

When testing equality between two universal types, a mismatch between the tags implies that the objects are not equal, and the result should indicate inequality instead of raising CONSTRAINT_ERROR.

1.7.8 Membership Tests

A membership test on a universal object is an error if an implicit conversion from the object to the type is an error.

For a membership test of a tagged universal object in a non-universal type the compiler must compare the object's tag with the types tag.

For a membership test of a tagged universal object and a universal type, the compiler must determine if the root-type of the universal type is an ancestor of the non-universal type of the object's value. Recall that in our model a type descriptor contains a distance-from-root field, and an array of the parent's tags. With this model the membership test can generate the following code:

```
 ;
-- Given the Source Code "X in T2'CLASS", where X is declared T'CLASS:

DISTANCE_FROM_TARGET :=
    X'TAG.DISTANCE_FROM_ROOT - T2'TAG.DISTANCE_FROM_ROOT;

DISTANCE_FROM_TARGET > 0 and then
    X'TAG.PARENTS_TAGS[DISTANCE_FROM_TARGET] = T2'TAG ;
```

As with Ada 83, the discriminants/bounds of the object must be checked as well.

If the compiler can determine statically that a conversion between the object and the type will raise constraint error, then the membership test indicates non-membership statically.

2. Abstraction Mechanisms

This chapter describes new mechanisms introduced in Ada 9X which extend the user's ability to create abstract data types.

We are interested in feedback on the clarity and completeness of the semantic specification, as well as the implementation difficulty and feasibility of the proposed implementation models.

From users we are interested in feedback concerning the effectiveness of the additional capabilities in building easier to use, efficient, abstract data types.

Note that this module is a significant simplification of what is proposed in the mapping document.

2.1 Basic Concepts

Below we define some terms, from Ada 83, as well as new to Ada 9X.

A basic operation is an operation, other than an operator or literal, which is implicitly declared along with a type. The basic operations in Ada 83 are assignment/initialization, allocators, membership test, short-circuit control forms, selection, indexing, slicing, qualification, explicit and implicit type conversion, literals, aggregates, and attributes. Basic operations are directly visible wherever they are used.

All types are classified as either composite or elementary. The classes which make up composite types are record, array, protected record, and task. Scalar, and access types are considered elementary. Private types are considered composite outside the package where they are defined. Within that package they are classified according to the full type.

In Ada 9X we introduce the notion of a *redefinable basic operation*. These are basic operations for which the user may define a function or procedure that gets invoked in place of the pre-defined basic operation. The redefinable basic operations described in this module are finalization and assignment. These operations are redefinable for composite limited types. Elementary types always use the pre-defined basic operations. A redefined basic operation is only used outside the package where the type is declared.

2.2 Default Initialization

The new syntax for declaring a type/subtype is:

```
full_type_declaration ::=
  type identifier [discriminant_part]
    is [tagged] type_definition [:= expression]

subtype_declaration ::=
  subtype identifier is subtype_indication [:= expression];
```

The specified default initial expression is evaluated each time an object is declared or allocated without an explicit initial value, and the resulting value is assigned as the initial value to that object.

2.2.1 Discriminants

As in Ada 83 discriminants may be referred to by default initialization expressions. In Ada 9X any elementary type may be a discriminant, and any composite type may have discriminants. For example:

```
-- This type is an unconstrained array type with a constrained
-- lower bound.
type Sequence(End: Natural) is array(0 .. End) of integer;

-- This task type is discriminated with an access to a record type.
-- Because it is a discriminant, the task need not perform a
-- rendezvous to receive the data that is is to work on.
type Data_Ptr is access Data_Type;
task type Producer(Data: Data_Ptr) is
    entry ...
end Producer;
```

If a discriminant is used to control the size or shape of the type, then it must be discrete.

2.3 Type Finalization

In the visible part of the same package specification as a composite type declaration, a finalization routine may be declared for that type. User defined finalization is not allowed for elementary types. A declaration of a finalization operation for a type, causes that type to become limited.

```
procedure T'FINALIZE(Object : in out T);
```

This operation is used to finalize the current value of an object. It is called in two situations:

1. Prior to unchecked deallocation
2. Prior to scope exit (see Mapping Document 3.11).

For a composite type, full object finalization consists of using the specific finalize operation for the composite type, followed by the finalization of components and/or finalization according to the parent type (if defined).

[NOTE: For a tagged limited composite type, a separate anonymous subprogram must be constructed by the compiler, and referenced in the type descriptor. This procedure must perform the full object finalization, so that it may be called to finalize instances of the universal type.]

2.4 Equality

Any type (not just limited types) may have an "=" operator defined by the user. If the operator returns STANDARD.Boolean, then a "/=" (with the obvious semantics) is implicitly declared just after the declaration of "=". This may not be overriden, however, the user may declare a "/=" function if the result type is not STANDARD.Boolean.

The *Operator Visibility* Implementation Module discusses the issue of composition of operators. We would like feedback on the difficulty and usefulness of having user-defined "=" and "/=" compose automatically.

2.5 Implementation Model for Generalized Discriminants

As described above we have proposed that discriminants be generalized. Any elementary type, (ie. scalar, and access), may be a discriminant, and any composite type may have discriminants. In Ada 83 the prevailing model for record discriminants is that they are implemented as fields of the record. The prevailing model for arrays is that bounds are implemented separately from the array. In Ada 9X we propose to allow a new pragma, SEPARATE_DISCRIMINANTS, which directs the compiler to use an implementation of record discriminants where they are not necessarily contiguous with the record. This allows an array of constrained records to have just a single set of discriminants, shared by all elements of the array.

Our preferred model is for array discriminants, and separate record discriminants, to be allocated in storage separate from the object. We are interested in feedback on the difficulty of this model. A possible alternative is to implement discriminated arrays exactly as discriminated records containing a single array component.

Task discriminants may be stored wherever the implementation finds convenient. Note, however that the discriminants are accessible by the task, and since the deallocation of a task object must not affect the operation of th task body, the discriminants should not be deallocated as part of the deallocation of the task object.

Protected Record discriminants should be treated like record discriminants and stored contiguous with the data, unless SEPARATE_DISCRIMINANTS are specified via the pragma.

Discriminants used to constrain the parent type are not considered extensions and re-use the space of the parent discriminants.

2.6 Implementation Model for Finalization

There are several possible implementation models that can be adopted for finalization. It is possible that implementations will have some of the mechanisms already in place for handling task completion and collection finalization.

2.6.1 Finalizing Allocated Objects

Objects created by an allocator must be finalized prior to deallocation (via unchecked_deallocation) or if not already finalized, then at the point where its associated access type goes out of scope. This implies that all objects associated with an access type, which have been allocated and not previously deallocated, must be linked together in some fashion that allows the finalization for the access type to iterate over each one and finalize them in turn. Only after an object is finalized may the storage be reclaimed. It is essential that the finalization happen on scope-exit of the access type, since the finalization procedure may reference data which may be going away.

If each access type is a separate storage pool, then it may be possible to iterate over each object without using explicit links.

2.6.2 Finalizing Declared Objects

2.6.2.1 Normal Scope Exit

Scope exit may be normal or asynchronous (see MD 3.11). The model for finalizing during a normal exit is perfectly straightforward. The compiler can determine at compile time the objects which require finalization and emit code which calls the finalization routines in the correct order (ie. reverse of the order that the objects were elaborated.)

2.6.2.2 Asynchronous Scope Exit

Asynchronous scope exits may occur before all the objects in a region are elaborated. The set of elaborated objects must be determined dynamically in this case. There are two techniques for doing this. Most compilers will already use one of them to implement the task finalization semantics of Ada 83.

2.6.2.3 Linking Elaborated Objects

One technique for keeping track of the elaborated objects is to maintain a linked list of pointers to the objects and the finalization routine required to finalize the object. After an object is elaborated, the objects are placed into the list, and finalization is implemented by walking the list (in reverse elaboration order) and calling the finalization routine with the object as a parameter. This technique introduces time and space overhead.

2.6.2.4 Program Counter Maps

A second technique is to maintain a mapping between program counter locations and the generated code for finalizing objects. If an asynchronous scope exit occurs the RTS uses the PC map to determine where finalization must begin. This mechanism is very similar to the way exception tables are used in many Ada 83 run time models. This technique has lower time and space overhead than the other, and is probably more suitable for real-time embedded systems than the linked-list approach.

2.6.2.5 Incompletely Elaborated Objects

An object which is incompletely elaborated should not be finalized, however any fully elaborated components (that require finalization) of such an object must be finalized. This can be accommodated by the PC map solution if the initialization of the object is done in-line without loops. If the initialization is done out of line or via a loop (for array objects), then the object must contain an indication of how far the elaboration has proceeded.

An implementation may also choose to protect against incomplete elaborations of objects by setting up regions around their elaboration where aborts are deferred. We expect that this is rather difficult to do efficiently, and do not recommend it for real-time systems, as it violates the requirement in the real-time annex that abort be immediate.

2.7 Implementation Strategies

2.7.1 Type/Subtype Declarations

2.7.1.1 Default Initialization

The symbol table must be enhanced to include a default initialization expression for each type and subtype. A similar mechanism may already exist to handle record components. When a type or subtype declaration with a default initialization expression is encountered the expression should be parsed, semantically analyzed, and saved along with the type or subtype definition. Subtype declarations which do not provide a default expression of their base type use the one associated with the type mark in the subtype indication. Derived types use the default expression associated with the parent subtype, if not overridden.

2.7.1.2 Finalization

The symbol table entry for a type should also indicate the 'FINALIZE, 'ASSIGN routines defined for that type (if one is provided.)

2.7.2 Object Declarations

When an object is declared (without an explicit initial value), and the type or subtype has an initialization expression, code should be emitted to compute that expression and assign it to the object.

If the type of the object has finalization, then certain actions must be taken to set up for the possibility of an asynchronous scope exit. These are discussed in Section 2.6.2.

2.7.3 Parameter Declarations

The default initialization for a type or subtype used in a parameter definition is not used to initialize parameters. **In** and **in out** parameters get their initial values from the actual parameters of the call. The initial value of an **out** parameter may be one of the following three values:

1. the actual parameter, if any part of the base type of the parameter requires default initialization
2. an unspecified (ie. stack junk) value in any other case.

2.7.4 Scope Exit

At scope exit the compiler must generate code to finalize all the relevant objects. If the PC map model is used this entails generating code to call each finalization routine, and recording the location the beginning of that finalization sequence in the PC map. This code should be placed so that it will be executed after exceptions are handled, and before they are propagated out of the frame. In the normal subprogram exit case, the code should be generated just prior to the subprogram's epilogue code. It would be best if the finalization can be arranged so that the code is not duplicated within the procedure. This should be simple for compilers which propagate exceptions by modifying a subprogram's return address.

When exiting a declare block by falling off the end, the finalization code should be inserted just after the last location of the block. Finalization code must also be inserted before any goto's, exits or returns which

transfer control outside of the of the block.

3. General Access Types

This chapter proposes changes to access types which allow them to point at declared data and subprograms as well as heap resident data.

We are interested in feedback on the issues regarding clarity and completeness of the semantic specification as well as any implementation difficulties, particularly in the areas of unchecked_deallocation, and collection reclamation.

This chapter proposes one of the few upwards incompatibilities in Ada 9X, scope-checking of limited types in return statements. There are two possible workarounds for existing programs encountering this problem.

1. If the type is limited only for the purpose of redefining "=", then the type can be made non-limited, since "=" can be redefined for any type in Ada 9X.
2. The subprogram (or the package containing it), which returns the limited type, can be made into a child unit of package where the type is declared. The full type is then visible, and may be non-limited.

We are very interested in feedback from the user teams on the effect that this change has on their programs, and the effectiveness of the workaround.

3.1 Basic Concepts

We begin by introducing several concepts from Ada 83, and then discuss new concepts and modifications of the old concepts for Ada 9X.

In Ada 83 a *limited* type is one which does not allow assignment or initialization, but does allow a user-defined equality. A limited type is either a task, a private type declared to be limited, in which case it is only limited outside the immediate scope where it is declared, or type with at least one limited component, in which case it is limited even within its immediate scope.

In Ada 9X the rules associated with limited types are modified. All types allow the re-definition of "=", so there is no distinction for limited types in that regard. In Ada 9X a limited private type has no operation which copies an instance of the type. Therefore, functions that return limited types must have a by-reference function return. Hence, functions which return objects of limited types must perform a *scope check* at compile time to ensure that the return expression does not reference data that is local to the function. In Ada 9X limited types may have user defined finalization.

A *scope check* is a compile time check performed at various places to ensure that dangling references will not be created.

An object which is to be pointed at by an access type must be declared with the "(aliased)" modifier. In this document we will refer to such objects as aliased objects, even though it is probably more correct to call them aliasable. Heap objects are considered aliased.

3.2 Access Types

The Mapping Document defines the new syntax and new rules for access types in sections 3.9, 3.9.2, and 4.6:(15,18,19). All of these rules should be implemented as part of this module, except those associated with limited access types, (**not null**) constraints, and other user-defined assertions. The relevant paragraphs of these sections are 3.9:(1,4,7,8,9), 3.9.2:(2,5,7-11), and 4.6:(15,18,19).

The new syntax for access type is:

```
access_type_definition ::=
      [in] access subtype_indication
      | access [protected] procedure [formal_part]
      | access [protected] function [formal_part]
            return subtype_indication
```

The **in** modifier associated with an access type implies that the designated object may not be modified through that access type. 'ACCESS applied to constant objects and mode **in** parameters is only overloaded on **in** access types.

'ACCESS may not be applied to array slices.

3.3 Limited Types

As part of this module we introduce a new upwards *in*compatible restriction on limited types. Values of a limited type must be scope-checked if they appear as return values. The effect of this is to disallow local variables of limited types to be returned from functions. The scope-check of a return value fails if the expression in the return value denotes a local object, is a function call where an actual parameter denotes a local object, or is a function call to a function declared at a more deeply nested scope than the function being returned from. Note that scope checks are not performed if the return statement has visibility on the full type which is not limited.

3.4 Implementation Models

The purpose of these changes is to allow safe and efficient low-level programming. The implementation model is designed to be as transparent as possible.

For the most part an access value should be equivalent to a machine address. There are, however, three cases where the model requires an access value to contain more than a simple address.

1. Access to a protected subprogram must designate both the protected record and the code of the subprogram.
2. Access to a nested subprogram may require a static link (or local display) as well as the address of the subprogram. The scope checks ensure that the static link is not a dangling reference, and in the case of a global display, the scope check ensures that the display is still valid wherever the subprogram is called.

3.4.1 Access Type Storage Pools

The model for access type storage pools in Ada 9X is that they define a boundary where finalization of the allocated objects must occur, but they do not necessarily represent distinct storage pools. Because of the new rules regarding conversion of access types, the model is that all access types without a specified STORAGE_SIZE at the same scope-level allocate their storage from a single pool.

3.5 Implementation Strategies

3.5.1 Subtype Declarations

Subtypes may have the modifier ‘‘(aliased)’’ in their declaration. This information must be retained in the symbol table.

The scope level of a declared access type must be retained in the symbol table for use in performing scope-checks.

3.5.2 Object Declarations

Any aliased object must yield a valid access value when suffixed by the ’ACCESS attribute. This implies that all such objects should be represented as if they were allocated on the heap. Dope and or type tags must be allocated just as for heap objects. The scope-level of an Aliased object must be remembered so that scope-checking can be performed when the object name is suffixed with ’ACCESS.

3.5.3 Parameter Declarations

The aliased modifier may appear in the definition of a formal parameter or the subtype of a formal parameter may indicate that the parameter is aliased. In either of these two cases the fact that the parameter is marked aliased must be remembered.

3.5.4 Subprogram Calls

At subprogram calls, an aliased formal parameter may only match an aliased actual parameter. The aliased parameter must be passed by reference. We are interested in feedback on how difficult it would be to implement an implicit dereference when an access value is passed opposite an aliased formal parameter.

A scope check is required on actual parameters passed opposite aliased formal parameters. The check fails if the object passed as an actual is declared in a more deeply nested scope than the subprogram being called.

3.5.5 Applications of ’ACCESS

When ’ACCESS is applied to an object, an access value designating the object must be generated.

When ’ACCESS is applied to a subprogram, an access value designating the subprogram is created. If

that subprogram is nested, then a static link or local display may be needed in the value of the access type.

For an object O of type T, O'ACCESS and "**new** O" are overloaded on the following types:

1. All access types which designate T.
2. All access types which designate T'CLASS or any P'CLASS where P is derived from T.

The type of the access value generated must be determinable from context when 'ACCESS is applied to an object. The scope-level of the access type should be compared to the scope-level of the object. If the object was declared at a deeper nesting level than the access type then the scope-check fails and the compilation must be rejected.

For the purposes of scope checking, an aliased formal parameter is assumed to be declared at the same level as the subprogram.

3.5.6 Function Return

As previously described, return values of limited type must be scope-checked.

3.5.7 Type Conversion

The rules for allowing conversion between access types are modified in Ada 9X.

For any combination of access types, the scope-level of the source must be no less nested than the scope-level of the target.

For any combination of access types, if the access subtype or designated subtype is constrained, then the designated objects must have matching or corresponding constraints. (This is a run-time check).

For any combination of types, if the part of source corresponding to target has a different representation than target, then the conversion is illegal.

If the source access type does not designate a tagged universal type then the target access type may have the following designated types (assume the source designated type is T1 or T1'CLASS):

1. T1
2. T1'CLASS
3. T0, an ancestor of T1
4. T0'CLASS, where T0 is an ancestor of T1

If the source access type has a tagged universal designated type (T1'CLASS), then the target access type may be any of the above or

1. T2'CLASS, where T1 is an ancestor of T2, checked at run-time to ensure that the designated object is convertible to T2'CLASS.

3.5.8 Finalization

Finalization of the collection associated with an access type must not affect the declared objects that are designated by access values.

3.5.9 Scope Checks

This section summarizes all the places in a program where scope checks must be performed.

1. When 'ACCESS is applied to an aliased object, the scope of the object must be checked against the scope of the access type implied by the context of the expression. The scope of the access type must be no less nested than the scope of the object.
2. A type conversion between two access types must check that the scope of the target access type is no less nested than the scope of the source.
3. If the expression in a return statement for a function *F* is of limited type, the value must be scope-checked. The expression may not depend, directly or indirectly, on limited objects whose scope level is more deeply nested than the scope level of *F*. Such objects are local variables, local functions which return a limited type, or global functions with local limited actual parameters.
4. At subprogram calls with an aliased formal parameter.

4. Interrupt Handlers

This module contains the necessary changes to support direct interrupt handlers. A new predefined package is proposed which contains primitives to associate interrupt sources with locking procedures as handlers. It also specifies the interaction between interrupts, priorities, and mutual-exclusion of protected records.

We are interested in feedback on the following issues:

- Clarity, completeness, and consistency of the semantic specification.
- The feasibility of efficient implementations, and possible adjustments to improve it.
- Ease of integration with the rest of the RTS, and interactions with other features of Ada83 tasking.

Since interrupt handlers are low-level mechanisms, they have to be very efficient. Therefore, we are particularly interested in issues such as implied interrupt latencies, interrupt response-time, length of non-interruptible sections, overhead in invoking and returning-from a handler, and possible conflicts with the rest of the RTS (especially in mutual-exclusion needs).

4.1 Proposed Change

4.1.1 Overview

A pre-defined language package is introduced; this package encapsulates all the services and operations needed to adequately handle interrupts. The mechanism uses non-interruptible/masking protected records as the means to synchronize the shared data between tasks and interrupt handlers. The interrupt handler code itself is encapsulated in a locked procedure. This procedure can be dynamically associated (and dissociated) with specific interrupts by calling the *Attach/Detach* procedures. When attached to an interrupt, such a procedure will be called by a "hardware task" at the priority of the interrupt (if the hardware system has such a notion), preventing other, "normal" tasks, from accessing the corresponding protected record instance.

4.1.2 Interrupt_Management Package

```
package Interrupt_Management is

   type Interrupt_ID is <implementation-defined>;
     -- How interrupts are named.

   package Interrupt_Names is

      -- Define named constants for all supported interrupts:

      A : constant Interrupt_ID := <implementation-defined>;
      -- ...
      Z : constant Interrupt_ID := <implementation-defined>;

   end Interrupt_Names;

   Handler_Already_Attached,
   Handler_Not_Attached,
   Reserved_Interrupt : exception;

   type Handler_Type is access protected procedure;

   Null_Handler : constant Handler_Type := null;

   procedure Attach_Interrupt_Handler (
      H : in Handler_Type;
      Interrupt : Interrupt_ID);
      -- Attach the procedure pointed to by H to interrupt Interrupt. The
      -- associated protected record instance is marked such that Interrupt
      -- will be masked each time the record instance is accessed.
      -- May raise Handler_Already_Attached.
      -- May raise Reserved_Interrupt.
      -- May raise PROGRAM_ERROR.

   procedure Detach_Interrupt_Handler (
      H : in Handler_Type;
      Interrupt : Interrupt_ID);
      -- Detach the handler from Interrupt.
      -- May raise Handler_Not_Attached.
      -- May raise Reserved_Interrupt.

   function Is_Attached (Interrupt : Interrupt_ID) return boolean;
     -- Return true if Interrupt is currently attached to a handler.

   function Interrupt_Handler (Interrupt : Interrupt_ID)
     return Handler_Type;
     -- Return the handler (or Null_Handler attached to the interrupt.)

   function Is_Reserved (Interrupt : Interrupt_ID) return boolean;
     -- Return true if Interrupt is reserved by the RTS.

end Interrupt_Management;
```

4.1.3 Notes on the Package

The *Attach_Interrupt_Handler* operation shall install the protected procedure with the entry point specified by *Handler_Type* as the handler for *Interrupt*. *PROGRAM_ERROR* will be raised if the indicated interrupt has a higher priority than the corresponding protected record object, or if the priority of the designated protected record is not in the range of *System.Interrupt_Priority*.

The *Detach_Interrupt_Handler* operation removes the specified *Handler_Type* from the specified *Interrupt_ID*, and restores the system default handler. The implementation shall document the actions of the default handlers for all interrupts.

The association of an interrupt with its handler occurs when the interrupt is actually delivered. (e.g. if it cannot be delivered immediately because the CPU executes in a higher priority, and the handler is replaced during the time the interrupt is pending, then the interrupt will be handled by the newer handler.)

The combinations of priorities and interrupts supported will depend on the hardware architecture. Any limitations shall be documented.

Note that multiple interrupts may be attached to the same handler.

Whenever an interrupt handler executes, it will have a CPU priority equal to that of the protected record associated with it. An interrupt handler is allowed to raise its priority while executing.

4.1.4 Static Binding

Since protected records do not provide for arbitrary initialization code, a new pre-defined pragma *Attach_Interrupt_Handler* is added. This pragma is used to indicate the interrupt association of a procedure within a protected record instance. Upon initialization of the object, the handler will be attached, and as part of the object finalization is will be detached automatically.

Proposed syntax:

pragma Attach_Interrupt_Handler(protected_procedure, interrupt_id);

Note that more than one *interrupt_id* may be attached to a particular handler, but no two handlers may be attached to the same *interrupt_id*. *PROGRAM_ERROR* will be raised upon an attempt to do the latter. *PROGRAM_ERROR* will also be raised if the indicated interrupt_id has a higher priority than the corresponding protected record object, or if the priority of the designated protected record is not in the range of *System.Interrupt_Priority*.

4.1.5 Handlers' Priority and Mutual Exclusion

Interrupt handlers have a priority which is in the range of *System.Interrupt_Priority*. Usually, tasks in the system will have lower priorities (in the range of *System.Priority*), but this is not mandated. Whenever the application needs are such, the priorities may overlap. Furthermore, on hardware systems where no prioritized levels of interrupt exist, all interrupts would be mapped to the same priority level, and masking of specific interrupts will be used to ensure mutual exclusion.

An interrupt handler will be disabled (in an implementation dependent way) when a task with a higher priority executes on the CPU where the interrupt may occur. In addition, the task which has been interrupted continues to be **runnable** (i.e. it may resume executing before all the activity that was initiated by the handler has been completed). Implementations will be required to define the maximum necessary stack space required for each task if nested interrupts are supported. This maximum may be dependent on the procedure being attached as an interrupt handler.

4.1.6 The Model of an Interrupt Handler

Conceptually, an interrupt handler may be viewed as a task which is being created, elaborated, and starts executing when the interrupt occurs, and terminates when the handler returns. This is in contrast with the model of a task looping while servicing one interrupt on each iteration. No state is saved from one invocation to another. Unhandled exceptions are lost and the handler code is abandoned (just like tasks). Raising the priority of a handler will remain in effect only for the current interrupt. Latter interrupts will have the original priority.

4.1.7 The Handler's Environment

The exact environment in which an interrupt handler runs should be fully documented by each implementation.

The following minimal behavior is required of each implementation:

- The interrupt should be acknowledged (if required by the hardware) prior to invoking the interrupt handler.
- The interrupt (and all lower priority interrupts, if applicable) should be masked/disabled prior and during the time the interrupt handler runs. They should be enabled upon the return from the interrupt handler (including the case when nested interrupts occur).
- A stack of an appropriate size should be available to the interrupt handler.
- The interrupted task should remain in a **runnable** state. i.e. if the interrupt handler returns "through **Dispatch**", another task may be resumed before the interrupted task. The task that was originally interrupted should be able to continue executing later (i.e. no LIFO order on return from handlers should be assumed).
- The RTS should be in such a state that all allowable operations from the handler will be supported, and all limitations will be enforced.

The following implementation-defined characteristics should be documented:

- The complete type definition of *Interrupt_ID* and the various operations and constants provided.
- Which interrupts are reserved by the implementation, so that they cannot be attached by the program.
- How nested interrupts are supported, and what limitations, if any, apply ?
- How lower priority tasks or interrupts handlers are prevented from running when a higher priority task or an interrupt handler executes ?
- How interrupts are handled (ignored, queued ?) when the priority level prevents them from being delivered.
- Which stack the interrupt handler uses, and any limitations on its size ?
- Any implementation or hardware specific activity which happens before the interrupt handler gets control (e.g. reading device registers, acknowledging devices, etc.).
- Any timing limitations imposed on the interrupt handler code to ensure proper behavior of the RTS.
- Any other implementation-dependent limitations, and/or special cases in the RTS state when the interrupt handler executes.

4.1.8 Limitations on Interrupt Handlers

Certain limitations are placed on the allowable constructs within the interrupt handling procedures. These limitations are needed in order to allow the procedure to be invoked directly by the hardware with minimal RTS support, and are similar in nature to the limitations placed on any locking subprogram. Violations of these limitations should be detected by the implementation. The constructs in the "immediate scope" of the interrupt handlers procedures will be checked statically. When this is not possible (i.e. when a separately compiled code is invoked by the interrupt handler), a *PROGRAM_ERROR* should be raised at run time.

These limitations are:

1. No potentially suspending operations are allowed to be called from within an interrupt handler. (e.g. I/O, queuing locks, task creation, synchronous rendezvous, accepts, blocking calls, delays, etc.)
2. Invoking locking subprograms on other protected records of lower priorities is disallowed (this limitation is not specific to interrupt handlers though).
3. Only procedures of library level protected records may be attached as interrupt handlers.

4.1.9 Miscellaneous Issues

4.1.9.1 Reserved Interrupts

Applications cannot expect to be able to handle all existing interrupts on a specific target. Implementations may reserve some of the interrupts for their correct functionality, and may reject an attempt to attach such an interrupt by the program. All reserved interrupts should be documented.

4.1.9.2 Selective Linking

Attaching a handler to an interrupt should indicate to the implementation that this procedure is in fact being called (it may be the only reference to the procedure), and code elimination is inappropriate in that case.

4.1.9.3 Vendor Provided Extensions

Some level of protection against misuse is provided by the *Interrupt_management* package. However, the final responsibility for the correct behavior of an application which uses interrupt handlers, on a particular hardware, is on the programmer himself.

Since we expect vendors to provide additional low-level primitives to control the machine's interrupt state (we bundle those under unchecked programming), we also allow an interrupt handler to lower its execution priority or even to enable such interrupts that otherwise would have to be disabled (to ensure the behavior required for protected records' mutual exclusion). This is done in order to allow for added flexibility in this very application and system dependent area. However such code will not be portable to other systems. It is therefore, the responsibility of the user to ensure that system-wide invariants are not affected by these actions. (The following are some of the expected invariants: the priority of the handler is the same as its invoking interrupt and lower priority interrupts never occur when an higher one runs.) For example, it might be necessary for an interrupt handler running at priority 7 to enable temporarily an interrupt of priority 2 (lower). The designer of the system must then ensure that no violation of locks

usage will result, since the RTS is not in a state to detect it. Violating these invariants is erroneous.

4.2 Implementation

4.2.1 General Notes

The model of interrupts and interrupt handlers is tied closely with protected records and priorities. Each protected record object will have one or more data words to contain interrupts related information. Since the *Attach* operation is performed after the creation of the object, this information cannot be initialized at creation time; enough space have to be allocated so it can later be updated by further *Attach*'s and *Detach*'s. The interrupt mask associated with a particular protected record object (and possibly other exclusion information) is used by the generated code when accessing this object from *normal* tasking code (to ensure that no conflicting interrupts can occur at the same time).

The type *Handler_Type* is an access type to a protected procedure. As part of the information carried in objects of this type is the identity of the protected record object, its type, and the code address of the handling procedure.

Implementing the pragma should be straightforward. It is only legal for a library level protected record. The pragma is provided since the user does not have a way to put arbitrary initialization code in a protected record. This code is necessary when the associated interrupt has to be attached immediately when the protected record is elaborated. When the pragma is encountered, the generated code should call the *Attach* procedure on behalf of the user. At object finalization time, the *Detach* procedure should be called to restore the default handler for that interrupt.

Each thread of control should have a *Suspendable* flag. This flag will be true whenever the task executes outside of any locking subprogram. It will be false otherwise, or when an interrupt handler is running. (We intentionally use the term "thread of control" here, and not "task" or **TCB**, since we do not want to limit the implementation technique regarding the way in which the interrupt handler's context is created and maintained.) This flag will be checked whenever a blocking operation is issued; if it is not set, this is an error condition, and a *PROGRAM_ERROR* should be raised.

If the particular hardware system determines the priority levels of the interrupts, those levels have to match the ones specified for the corresponding protected record objects.

Masked, or disabled interrupts should remain pending and be delivered when they are enabled again. Whether the pending interrupts are queued, acknowledged or lost, is an implementation and target specific issue and should be documented properly.

Exceptions could be raised from within an interrupt handler code. They can be handled, however, only by the handler code itself, and if no appropriate handler exists, the handler's code is abandoned, and the exception is lost. Since the mechanism to handle exceptions is beyond the scope of this module, the only requirement here is that this mechanism should support raising exceptions from interrupt handlers.

4.2.2 Interrupt Delivery

The interrupted task state has to be saved for a later resumption. This resumption may occur "much" later. For example, the interrupt handler may resume a higher priority task, or the handler may be interrupted by a chain of nested interrupts. In order to support these cases, the interrupted task state has to be saved "normally", like it was preempted by a higher priority task. This will enable it, later on, to be resumed asynchronously and not necessarily as the result of unwinding the nested interrupts stack. This requirement implies that there should be a separate stack for the interrupt handler(s). We leave it as an implementation choice whether there should be only one stack, one per interrupt, or some other combination. The stack(s) should be allocated before the interrupt is delivered (either statically or when the handler is attached). We would like to know about the specific decision, and the reasons behind it.

Determining the amount of state to save on each interrupt always involves trade-offs. Usually, the state is saved in two parts, the most elementary one is saved immediately after the interrupt occurs (in a global place), and only if a full context-switch is expected, then the full task state is saved (probably in or of the *TCB*). Often, when it is known that the interrupt handler will return immediately and no context-switch is expected (and this fact can be determined quite early), only a very small context (usually a couple of registers) need to be saved. Another critical decision is when to analyze the source of the interrupt (trap, RTS related, user's, spurious, etc.) and when to split the processing. The trade-offs are between simplicity, extensibility, interrupt latency, interrupt disabling period, etc. Since these decisions are highly architecture dependent (and sometimes even application dependent), we leave the details to the vendors, and again, we are interested in feedback regarding the decisions and the expected difficulties.

Depending on the exclusion mechanism chosen, and the priority of the interrupt, other interrupts may have to be masked or disabled when the particular interrupt occurs. On some systems, the CPU priority level may be raised to accomplish that, on others, specific interrupts may be masked, and yet on other targets, the most efficient way to achieve the exclusion is through the disabling of all interrupts (with all the known drawbacks). Like most of the rest of the implementation issues, this one is also left to the vendors to decide based on the particular target hardware. Since "normal" tasking code will also have to accomplish the same exclusion control (when accessing the protected record object), we are interested to know whether a switch to a privileged mode is required on each such access (on systems when the CPU has two modes).

After saving the necessary state, and potentially updating the RTS data structures (although we expect this book-keeping to be minimal), the appropriate handler (the previously attached protected procedure) should be called. The prologue of the procedure should not be any different than a normal locking procedure. The handler executes at the CPU priority of the corresponding protected record object. (Note that nested interrupts of higher priorities can still occur, but not at the same level.)

4.2.3 Return from Interrupt

As is the case with any locking procedure's epilogue, when the handler finishes its sequence of statements, all barriers of non-empty entry queues are evaluated (an optimization here would be to do it only if the state of the components has actually been changed), and all entry bodies of true barriers are executed and their respective callers are marked to be resumed. Like the common case, this activity is performed by the handler thread on behalf of the waiting callers.

It is not expected (nor desired) that the RTS data structures will be in a consistent state when an interrupt occurs. Therefore, most queuing operations (such as resuming other tasks) will not happen immediately but rather as part of the handler's epilogue. Since many handlers are executed without the need to resume other tasks, it is desired that the **return-from-interrupt** sequence be optimized away, when a simple return and resumption of the last task is enough. Less state is needed, and a simpler **return-from-interrupt** is possible when no scheduling is expected and the interrupted task is known to be resumed immediately. (Another approach is to postpone the "massive" state saving, and do it only when it is clear that a full context-switch is required.) In order to accomplish the above, some indication has to be maintained while the handler executes (see next).

Since actual resumption and queue manipulation is impossible from interrupt handler's code, some mechanism of deferring the work is necessary. A list of tasks to be resumed should be maintained by the interrupt handler's code. At the point of returning from the handler, the *Dispatcher* should be invoked if the list is non-empty. The thread that handles the actual resumption (after the *lock* has been released) could be a part of the *Scheduler/Dispatcher* or an auxiliary task. There are several trade-offs between these two techniques and we leave the decision to the implementors.

A generalization of task-list mechanism could be some form of a circular *work-item*s buffer which will be filled by interrupt handlers codes and emptied by the task-level scheduler. One advantage of this approach (as opposed to hard-coding the exact communication) is that it can serve as a more general-purpose mechanism of deferring work whenever the activity required by the handler is too long (or unbounded in time), and the processing needs to continue later after interrupts are enabled.

4.3 References to MI's

Note that MI's do not necessarily reflect the latest changes. The Mapping Document is the best source for an accurate description of the proposed changes. The MI's, where still accurate may provide more detail and a rationale.

1. MI-RT05.RWLocks
2. MI-RT09.DynPr
3. MI-RT11.IntHand
4. MI-OO02.GenlRef

5. Asynchronous Transfer of Control/Multi-Way Select Statement

5.1 Overview

This module contains the proposed changes in the area of tasking synchronization primitives. It builds on the previously delivered *Protected Records* module, and adds the changes regarding the support of asynchronous transfer of control and multi-way **select**. In also completes the proposed change regarding **requeue**, by adding the **with abort** option. Finally, it defines the concept of protected regions or *deferrable* aborts.

5.2 Purpose

As with the *Protected Records* module, we are interested in feedback on the following issues:

- Clarity, completeness, and consistency of the semantic specification.
- The feasibility of efficient implementations, and possible adjustments to improve it.
- Ease of integration with the rest of the RTS, and interactions with other features of Ada 83 tasking.
- We have removed the **and** option. We would like to know whether this simplifies the implementation, and whether the user teams feel that the eliminated functionality substantially reduces the expressive power of the proposed construct.
- The same is true regarding the elimination of guard expressions on the event alternatives. We would like to know whether this simplifies implementation and/or substantially reduces the functionality provided by the construct.
- There are two ways to define the behavior of the multi-way **select** with the **else** *final_part* when more than one event alternative exists. One is to require that all potential calls will be pending until the **else** is selected (and then all will be cancelled). The other is to check each alternative in a sequence, and if it cannot be executed, *NOT* to keep the call pending, but rather to continue with the next alternative, so that when the **else** is encountered, no cancellation of pending calls need occur. The difference in specification is not transparent to the program. We would like feedback both from the implementors and the users regarding the potential simplification of implementation, and which choice is less surprising to the user.
- What allocation technique is most suitable for QEL's? They could be allocated and initialized by the compiler, or allocated by the compiler and initialized by the RTS. They could be allocated and initialized by the RTS. In order to minimize dynamic allocation overhead, there could be some upper limit on QEL's per program or per task, with **STORAGE_ERROR** being raised when the limit is reached. We currently assume that the compiler will do the allocation and will initialize the QEL's together with the RTS. We would like to get feedback from the implementors regarding the various trade-offs involved.
- We currently allow parameters for entry calls in a multi-way **select** to be evaluated either at once before any call is queued, or one at a time. It is important that no locks will be held when parameters are evaluated since their evaluation might invoke user-provided code. We are interested in the various trade-offs associated with this choice.

The multi-way **select** with the *Protected Record* concepts constitute the main building blocks for user extensions to the synchronization primitives and parallel programming paradigms. Therefore, in addition, we are interested in feedback regarding the implementability of those changes as a whole, their interactions, and their usability. Finally, the feasibility of building the rest of the Ada RTS (tasking and rendezvous) on top of, and using the proposed new constructs, is of interest to us.

5.3 Proposed Change

5.3.1 Requeue Statements

A **requeue** statement may be used to complete the execution of the innermost enclosing entry body or **accept**.

requeue_statement ::= **requeue** *entry*_name [**with abort**];

A **requeue** statement is only allowed within an entry body or an accept statement and applies to the innermost such construct; a **requeue** statement is not allowed within a program unit body nested within the construct to which it applies.

In the rest of the discussion we do not address the **requeue** from **accept** anymore.

The entry named by the **requeue** statement must have no parameters, or have the same number of parameters as the innermost enclosing entry body, with the same parameter modes and statically matching subtypes for parameters in corresponding positions.

For the execution of a **requeue** statement, the entry name is first evaluated. Then the task which is actually performing the current entry begins an entry call on the newly identified entry. If the new entry is of the same protected record instance, the call is queued. Otherwise, if the barrier is FALSE, the call is queued; if it is TRUE, the entry call begins. If the new entry has formal parameters, then the values of these formal parameters will be initialized from the values of the corresponding formal parameters of the current entry at the time of the requeue. If the new entry does not have formal parameters, then the values of the formal parameters of mode **in out** or **out** of the current entry which were passed by copy are retained for later copy back and constraint check after completing the new entry call. Finally, the normal activities which happen prior to releasing a read-write lock are performed, and then the execution of the entry body enclosing the **requeue** is completed.

If the **requeue** statement includes the reserved words **with abort**, then the calling task becomes susceptible to abort or asynchronous transfer of control when calling the new entry. If an abort is pending, it will proceed. Otherwise, if the entry can be immediately executed, it will. If the call is queued, it stays susceptible until actually executing the body. Otherwise, existing or future aborts will be deferred until the task becomes susceptible.

5.3.2 Multi-Way Select

This section introduces a **select** statement with entry call alternatives, delay alternatives, and an optional *final_part* introduced by **else** or **in**.

```
selective_entry_call ::=
  select
     event_alternative
 {or
     event_alternative}
 [final_part]
  end select;

event_alternative ::=
  entry_call_alternative | delay_alternative

entry_call_alternative ::= entry_call [sequence_of_statements]

final_part ::=
    else sequence_of_statements
  | in sequence_of_statements
```

The event alternatives are considered in the order of their occurrence in the construct. For each alternative, the entry parameters (and entry index, if present) are evaluated and the entry is called. If the entry cannot be executed immediately, the call is queued, and the next alternative is considered. If the entry can be immediately executed, it is, and other alternatives are cancelled. At any time during the consideration of further alternatives, an earlier entry call may be accepted and processed. The first entry call to be accepted cancels all other pending calls atomically, ensuring that only one entry call will be processed. Committing to an alternative (or the else part) is done atomically; i.e. only one branch in the **select** may be chosen. However, parameters may be evaluated for more than one entry call.

An **else** part provides for a conditional call on one or more entries. If none of the entry calls is immediately selected, then the entry calls are cancelled and the sequence of statements of the **else** part is executed. Otherwise, the possible sequence of statements following the selected entry call is executed.

A **select** statement with an **in** part is discussed in the following subsection.

5.3.3 Asynchronous Transfer of Control

This section discusses the **select** statement with an **in** part. The sequence of statements of the **in** part executes while waiting for some event alternative (entry call or delay) to be ‘‘selected,’’ at which point the processing of the sequence of statements is aborted, and once the aborted sequence of statements becomes completed, the sequence of statements of the event alternative is executed. If the sequence of statements of the **in** completes prior to selection of an event alternative, all event alternatives are cancelled. If any event alternative is selected prior to beginning execution of the **in** part, then execution of the **in** part is not begun, the sequence of statements in the corresponding alternative is executed, and then the **select** is complete.

Note: For implementations directed to real-time domains, the sequence of statements must be aborted immediately. Any restriction or a possible latency, should be documented.

5.3.4 Aborting a Task and Aborting a Sequence of Statements

The **abort** statement allows an entire task to be aborted. A **select** construct with an **in** *final_part* allows a sequence of statements to be aborted. Each alternative in a **select** construct with an **in** *final_part* is an *aborting event*. An aborting event, associated with an outer scope, will cause all execution in more nested scopes to be abandoned (after exiting all protected regions). Any task dependent on a task being aborted, or dependent on a master within the sequence of statements being aborted, is itself aborted.

Abort is deferred when a task is executing within a *protected region*. The following are considered protected regions:

1. in a rendezvous, unless the acceptor task as a whole is aborted;
2. while performing a protected operation of a protected record;
3. while requeued on an entry queue if the reserved words **with abort** are not included in the **requeue** statement.
4. while in a handler part with a handler for **abort**;
5. while finalizing an object.

Once the task is no longer executing within a protected region, the **abort** begins or continues to propagate out of the frames in which the task is nested until the task completes (if the task as a whole is aborted), or until the aborted sequence of statements is completed.

All subtasks are marked **ABNORMAL** and finalization is then performed for all scopes, inner first and outermost last. Finally, the sequence of statements in the selected alternative is executed.

Objects requiring finalization are finalized even for an aborted task, as part of propagating **abort** out of the frame in which they are declared, directly, as a subcomponent, or as part of an object created in a collection associated with the frame.

5.3.5 User-Defined Timers

As part of the proposed multi-way **select**, we also are proposing a certain unification of entry calls with **delay** statements and delay alternatives with entry call alternatives. This in turn, allows us to provide for ‘‘user-defined’’ timers. Below, we describe the method in which user-defined timers can be programmed, and the transformation performed by the compiler in order to achieve the integration between the generated code, the RTS, and the user-provided code.

```
-- The following type will be predefined in 9X:

protected type Persistent_Signal is
   procedure Signal;
   entry Wait;
record
   Occurred: Boolean := False;
end Persistent_Signal;

-- The following will be defined (once) by the user
-- creating the special timer (i.e. not by every
-- one who uses it):

type My_Time_Type is ... ;
  -- e.g. time value 64-bits fixed-point

type DQEL_Type is record    -- delay queue element
   PR: Persistent_Signal;
   Time_Value: My_Time_Type;
   Links: ...;
   ... -- More components to be used by ENQUEUE/DEQUEUE
end record;

-- ENQUEUE and DEQUEUE are operations which may be
-- overridden by the user.  QUEUE_ELEMENT is a
-- new standard attribute.

procedure DQEL_Type'ENQUEUE (DQ: in out DQEL_Type;
                             Time_Value: My_Time_Type);

procedure DQEL_Type'DEQUEUE (DQ: in out DQEL_Type);

for My_Time_Type'QUEUE_ELEMENT use DQEL_Type;

   -- The issue here is how do we make sure that
   -- DQEL_Type has the appropriate Persistent_Signal
   -- component.  One way, is to say that DQEL_Type'QUEUE_ELEMENT
   -- is legal only if DQEL_Type is derived from some root type
   -- in System containing a component of type Persistent_Signal.


-- Example of a simple delay transformation:

delay My_Time_Type'(T);  --->

declare
   DQ: DQEL_Type;
begin
   DQEL_Type'ENQUEUE (DQ, T);
   DQEL_Type.PR.Wait;
   DQEL_Type'DEQUEUE (DQ);
end;


-- Example of a delay alternative transformation:

select
   delay My_Time_Type'(T);
or
   One_Pr.E1;
or
   Another_Pr.E2;
else
```

```
   ...
end select;  --->

declare
   DQ: DQEL_Type;
begin
   DQEL_Type'ENQUEUE (DQ, 500.0);
   select  -- "Normal" multi-way select
      DQEL_Type.PR.Wait;
   or
      One_Pr.E1;
   or
      Another_Pr.E2;
   else
      ...
   end select;
   DQEL_Type'DEQUEUE (DQ);
end;


-- The examples above assume the following user-defined timer
-- interrupt handler:

protected My_Timer is

   procedure Handle_Interrupt;
   pragma Attach_Interrupt_Handler(
     Handle_Interrupt,
     Interrupt_Management.Interrupts_Names.Special_Timer_ID
   );

   procedure Enqueue (DQ: in out DQEL_Type);

   procedure Dequeue (DQ: in out DQEL_Type);

record
   Next_Alarm: My_Time_Type := 0.0;
   Current_Value: My_Time_Type := 0.0;
   Q: Some_Queue_Type;
     -- Q is made of DQEL_Type records
end My_Timer;


protected body My_Timer is

   procedure Handle_Interrupt is
     -- Invoked by the interrupt and empties the Q
      Temp_DQ: DQEL_Type;
   begin
      -- This code is protected against
      -- the case when spurious interrupts occur
      -- (e.g., when a waiter has been removed
      -- from the Q prematurely.  This is NO-OP then.
      -- It may cause two alarms to be set to the
      -- same value, but this is not a big deal.

      Current_Value := Read_Or_Calculate_Value_From_Device;
      while Current_Value >= Top_Of_Q(Q).Time_Value loop
         Top_Of_Q(Q).Pr.Signal;  -- Resume the waiter
         Temp_DQ := Pop(Q);
      end loop;
      Next_Alarm := Top_Of_Q(Q).Time_Value;
      Set_Device_To_Interrupt(Next_Alarm);
```

```
      end Handle_Interrupt;

      procedure Enqueue (DQ: in out DQEL_Type) is
      begin
         Enter (Q, DQ);   -- Based on Time_Value, and using the links
         if DQ.Time_Value < Next_Alarm then
            Next_Alarm := DQ.Time_Value;
            Set_Device_To_Interrupt(Next_Alarm);
         end if;
      end Enqueue;

      procedure Dequeue (DQ: in out DQEL_Type) is
      begin
         Remove (Q, DQ);   -- Check links first to make sure
                           -- the entry is still on the Q
      end Dequeue;

   end My_Timer;

   procedure DQEL_Type'ENQUEUE (DQ: in out DQEL_Type;
                                Time_Value: My_Time_Type) is
   begin
      DQ.Time_Value := Time_Value;
      My_Timer.Enqueue (DQ, Time_Value);
   end DQEL_Type'ENQUEUE;

   procedure DQEL_Type'DEQUEUE (DQ: in out DQEL_Type) is
   begin
      My_Timer.Dequeue(DQ);
   end DQEL_Type'DEQUEUE;
```

5.4 Implementation

5.4.1 Introduction

In this section we describe the implementation model of the multi-way select, asynchronous transfer of control, and the **requeue** statement. The algorithm may be inlined by a compiler. For clarity, we have chosen to describe it as an out-of-line call from the generated code.

The two major data structures which are being discussed here are the *Select_Header* (SH) and the *Queue_Element* (QEL). Both are allocated on the caller's stack, initialized, and reclaimed by the compiler. There is one SH for each **select** construct and one QEL for each event alternative. A more detailed discussion of these data structures will be provided below.

5.4.2 Avoiding Races

Several mechanisms described below require protection against races when two threads of control are attempting to affect the same or conflicting results (e.g. committing to only one alternative of a **select** construct, or resuming a task that is about to suspend itself). These races only exist in multi-processor environments when the RTS is utilizing fine grain locking. We do not anticipate the need to use lower level locks in order to resolve those races. (In fact, if such locks become essential, we need to reevaluate other assumptions and decisions.) Instead, we assume the presence of an indivisible machine instruction

(such as test-and-set), and its use for that purpose. Since no spin-waiting is necessary in this case, and when one ‘‘contender’’ loses a race it simply gives up, we believe that this approach will suffice. (In the following descriptions, we do not repeat this discussion again, we simply caution the reader wherever it is applicable.)

5.4.2.1 The CLAIM Mechanism

On a multi-processor, more than one alternative may be selected at the same time. This may happen when two protected records change their state at the same time. A mechanism should be present to prevent such cases, and to ensure that when either an event alternative, an **else** part, or the completion of an **in** part is selected, all other alternatives are cancelled (the physical removal from the queues may be postponed, but an indication must exist to prevent them from being selected). We call this mechanism ‘‘committing to an alternative’’ and we use a *claim* bit in the *Select_Header* for this purpose. An indivisible operation, (*Claim*) on this bit allow us to avoid the use of a lower level lock.

```
function Claim (Sel_Head: SH_Access) return Boolean;
   -- By using an indivisible hardware instruction (such as
   -- test-and-set), the bit is "claimed". True is returned
   -- if the bit was not claimed (set) before, so the claim
   -- is successful, and False otherwise (i.e. the bit was
   -- previously set, somebody else claimed it already).
```

If the *Claim* operation is unsuccessful, normally nothing special needs to be done, and the processing of the QEL is completed. The reclamation of storage and other wrap-ups will be done later when the **select** is completed.

5.4.2.2 Suspend/Resume Race

We assume the presence of two simple signaling primitives: *Wait(SH.Op_Completed)* and *Signal(SH.Op_Completed)*. These primitives are used in various places (see below) to resolve races between the calling task suspending itself, waiting for an operation to complete, and another task just finishing the operation on its behalf. These primitives will also be used by the aborting mechanism. Note, that only one caller can be suspended on this flag, so efficient implementation is possible.

5.4.3 Data Structures

In the canonical implementation model described below, we assume the following data structures:

Entry_Desc_Rec: Each entry is identified by a record containing two fields: The *Entry_Name_Index* which identifies the entry body to be called, and *Entry_Family_Index* which identifies the particular entry family index, and can be used as the parameter for the barrier expression function.

Qel_Rec (queue element): Instances of this type are moved around on the various queues. All *Qel_Rec*’s can be allocated and often initialized by the compiler. A *Qel_Rec* contains: A pointer to the protected record instance on which it is queued, a pointer to the corresponding *Select_Header*, a pointer to the call parameters, the entry identification, and various links and flags.

PR_Rec: This is the data structure containing the actual components of the protected record instance in addition to all the fields needed for the RTS support such as: the PR priority and exclusion information,

a pointer to a type descriptor, and a queue head for waiting tasks.

Both barrier expressions and entry bodies' code are represented as pointers to the appropriate code segments. Arrays of such pointers are maintained in a descriptor for each distinct protected record type.

Queues: (We assume here a priority-based queuing discipline.) Each protected record instance will have a queue head associated with it which contains all waiting callers on entries of this instance. This queue is a two-dimensional priority queue of *Qel_Rec*'s. The row is a priority based list of callers for different entries. Each *Qel_Rec* in this row is the queue head of callers on the same entry, sorted in a priority order. The guard expression has always the same value for an entire column. Alternative implementations might use arrays and/or more sophisticated prioritized queues.

Select_Header: This data structure is created for each **select** construct. It contains an enumeration type to distinguish the various **select** kinds (based on the final part), flags such as *Claimed* and *Op_Completed*, an indication of the selected alternative, a pointer to the owning TCB, a nesting counter (see below), an array of *Qel_Rec*'s, one per each alternative (or a pointer to), and various links. The number of alternatives is statically known, therefore the array of *Qel_Rec*'s can be allocated by the compiler in the caller's stack frame. A simple entry call (i.e. one which is not a part of a **select** construct) will have a *Select_Header* as well pointed to by its *Qel_Rec*. This is done in order to avoid special-case code, since the RTS does not need to be aware of this fact. The rules regarding a simple entry call are the same as if the entry call is enclosed within an unconditional **select** construct with a single alternative.

Deferring_Abort_Info: a per task element. Since protected regions can be nested, the current nesting level of protection needs to be maintained. This element will be stored either in the TCB or in some task-private area (e.g. a dedicated global register). Several kinds of protected regions exist: when a task is holding a lock, it cannot be suspended or aborted; when it executes an **accept** statement, the corresponding region cannot be aborted, but the whole task can, and it may also be suspended; when a task is in its finalization code it cannot be aborted at all, but may be suspended. This data structure encapsulates all such information and is used to detect illegal suspension attempts or to defer abort requests. The various components of this element can be either counters which are incremented/decremented at the entry/exit to/from the particular region, or flags which will be saved on the stack frame on entry to the particular protected region and restored upon exit. The element is initially marked as allowing suspension and abortion. (Note that when an entry body is executed by another task, the element of the calling task is *NOT* updated).

Nested_Async_Level: a per-task counter (initialized to 0) of nested abortable regions (**select**'s with an **in** final part). This counter is incremented whenever an **in** sequence of statements begins execution and decremented when it completes. When a new *Select_Header* is created, if it is an asynchronous select, the TCB's *Nested_Async_Level* is copied to the corresponding field in the *Select_Header*.

Aborting_To_Level: another per-task component. It is initialized to *Integer'LAST* (i.e. no abort in progress). When an abort is in effect, the level to which it corresponds will be stored in this field. Another abort will override this field only if the value is smaller than currently exists (i.e. if we abort to a less nested level). Ada's **abort** statement will abort to level 0.

5.4.4 Generated Code and Transformations

5.4.4.1 Introduction

This section describes the canonical prologue and epilogue of protected operations, and the expected transformation of several constructs (using informal pseudo-code). The focus of the discussion is an efficient implementation for the simple and general cases. We would like to minimize the overhead implied by special cases (in particular, the abort), and the distributed overhead caused by the possibility of programming overly complex constructs. Whenever possible, we relegate special processing to the places where they are actually being used.

In the following description we suggest a specific division of work between the RTS and the generated code. We also separate between in-line code and out-of-line calls. Similarly, we assume a specific separation between the code that executes at the call-site, and that which executes as the prologue.

5.4.4.2 General

Because no new data or entries may be declared in a protected type body, the compiler has all necessary information to allocate the data structures for any protected record instance given visibility only on the protected type specification. Therefore allocation does not require an extra level of indirection, and can be made on the stack (or on a heap, if **new** is used). In this context, when we say ‘‘protected record data structure,’’ we refer to a structure containing both the user data components, and the necessary fields for the RTS support.

Similarly, *Select_Header*’s and *Qel_Rec*’s can be allocated by the compiler on the caller’s stack. Initialization can be done either by the generated code, or by the RTS. In this description we assume that most of the initialization is done by the generated code.

A separate proposed change discusses the association of protected records with interrupts. Usually (except when the association is static via a pragma), the *Attached Interrupt_ID*’s are not known at the time of object creation, so only a place-holder will be allocated, to be filled in later by the *Attach* operation.

Parameters for entries will be passed in memory rather than registers, so that they can be easily accessed after suspension by another task performing the entry body on their behalf, without requiring a complete context-switch.

The barrier expression will be transformed into a boolean function.

Locking subprograms, entry bodies, and the barrier functions will have an implicit parameter which is the protected record instance. When entry families are used, the entry bodies and barrier functions will have, in addition, the member index as a parameter.

All internal operations on protected records such as queuing, dequeuing, and evaluating barrier expressions, should be performed under the protection of the associated lock.

It is important to distinguish between single and multi-processor implementations. On a monoprocessor, locks need not exist physically; raising the priority, checking the ceiling rules, inhibiting preemption, and possibly masking certain interrupts, would be adequate to ensure exclusive access to

the protected record instance.

5.4.4.3 Wrappers, Macros, and RTS Calls

In the following sections we discuss ‘‘macros,’’ ‘‘wrappers,’’ and actual RTS calls. This separation, though not required, helps us to describe the canonical implementation model. In particular, the macros may be implemented as out-of-line calls, or as generated-code emitted sequences.

The code at the call-site invokes the appropriate wrappers (for protected procedures, entry calls, etc.) based on the compiled construct. Usually a result-code is returned by the wrappers which instructs the generated code how to proceed. If, as a result of an entry call or a **requeue**, the calling task needs to suspend itself, this is accomplished by the calling task issuing a *Wait* (see below).

Wrappers have an additional parameter which is the code address of the user code (procedure/function/entry body) to be executed. When the user code completes, control returns to the wrapper which, in turn, returns control to the call-site. Entry call parameters are placed in a special parameter area. The parameters for protected procedures/functions, however, are passed in the same manner as parameters to non-protected subprograms. Therefore, in the latter case, the stack has to be adjusted properly to allow the subprogram code to access the parameters normally, as in a non-protected operation. The current proposal allows a visible protected subprogram to be called from within the body of a protected record as well. In this case, no additional locking is necessary, and the call-site will invoke the subprogram directly without going through the wrappers. (This case can easily be distinguished based on whether the call uses the selector notation or just a simple name.)

If the implementation chooses to perform the locking as part of the prologue, then alternate prologues/epilogues will have to be generated by the compiler, and selected accordingly. Other techniques are possible of course.

Each wrapper will presumably be enclosed in exception and finalization handlers. These handlers will take care of exceptions raised by the RTS, and be responsible for the reclamation of all the objects that were created before the call (*Qel_Rec*’s, *Select_Header*’s, etc.).

We intend that by making its code sufficiently generic, only one version for each kind of a wrapper will be needed in the system.

The above description is further elaborated in the following sections where we describe the specific macros, wrappers, and RTS calls.

5.4.5 Get_Pr and Release_Pr Macros

A call on a protected operation from outside the body of the protected record needs to obtain a mutually-exclusive (or for functions shared (read-only)) access to the protected record instance. The code to accomplish this does not depend on the particular instance or type of the protected record, hence it can be shared for all PR’s. However, some variations may exist depending upon the mutual-exclusion mechanism being used, and whether an exclusive-write or read-only lock is required.

In this section we describe the macros as a single version common to all cases. Implementations may use multiple, optimized versions, or let the code-generator emit the appropriate code based on the

particular instance and the desired operation.

These macros will be used by all wrappers that need access to a protected record. Conceptually, they will have a parameter which is a pointer to the protected record instance, and a flag describing the mutual-exclusion mechanism used by this protected record. We assume that when protected operations are invoked using access objects, the access value of the operation contains, at least, a code address and a pointer to the protected record object. Note that these primitives will work quite differently in a single-processor environment or in a multi-processor one.

5.4.5.1 Get_Pr

```
Save original active priority.
Compare active priority of caller with that of the protected record instance.
  (Alternatively, check the interrupt level and the masking attribute, if any.)
Raise PROGRAM_ERROR if an illegal caller.
Increment PR nesting in Deferring_Abort_Info
Raise the priority of the calling task to that of the protected record instance.
  (Based on the Exclusion_Info, inhibit preemption/interrupts for the
  appropriate level.)

if single-processor then
   probably a no-op, the priority rules ensure the mutual-exclusion.
else
   if function then
      Get a read-only lock
   else
      Contend for the lock -- (e.g. by test-and-set) see: (*1)
   end if;
   Indicate that the lock is locked
end if;
```

Note:

(*1) There are various techniques to optimize this part depending on the characteristics of the system (number of CPU's, type of bus, etc.) Since failing to get the lock is usually quite rare, the contending task may continue with the busy-wait, or alternatively, lower its priority to the original value (enable preemption/interruption), and instruct the ''dispatcher'' to find somebody else to run.

5.4.5.2 Release_Pr

This macro is used by the various wrappers when all the protected processing is complete and a return to the call-site is about to occur. The result code is remembered and the QEL state is updated by now.

```
Unlock (may be no-op on mono-processor)
Clear locked indication of the lock
Restore active priority (potentially allow preemption/interruption)
Update Deferring_Abort_Info
if Deferring_Abort_Info is now zero then
   Check Aborting_To_Level
   Start aborting self if necessary
end if;
Call the ''dispatcher'' if necessary.
```

5.4.6 Wrappers

5.4.6.1 Procedures and Functions

This wrapper is invoked by the call-site when a protected procedure or function is called from outside of the protected record body (using a selector notation). The code address of the subprogram is passed as an additional parameter, and the stack is arranged to allow the body to access the parameters normally.

```
Get_Pr
ASSERT: PR now locked, priority saved.
Perform the body of the operation.
if an exception was raised, then remember it
if a function, then remember the return value
if a procedure then
   Scan_Queues  -- see: (*1) and (*2)
   ASSERT: No entries remain with true barriers and waiting callers
end if;
Release_Pr
Raise exception if occurred
Return result if a function
return
```

Notes:

(*1) See below for a description of *Scan_Queues*.

(*2) Possible optimizations:

1. If there are no changes which could affect any barriers, then skip call on Scan_Queues.
2. Evaluate all barriers now, and create vector of booleans. Can use **or** and **and** to set or clear individual booleans. Barriers not alterable by the procedure need not be reevaluated. Values already in registers can be used without reloading.

5.4.6.2 Simple Entry Calls

A QEL and a SH have been allocated and initialized by the caller. This wrapper is also invoked by the multi-way **select** and **requeue** transformation (see below). Parameters have been evaluated by now and placed in a parameter area pointed to by the QEL. In addition, the QEL contains a flag *Is_Cond_Call* which indicates whether it should be queued or not when the barrier is FALSE.

```
Get_Pr
Evaluate entry barrier (using QEL.Family_Index if necessary)
if any exception then remember it and goto <<fin>>
if barrier = FALSE then
  if QEL.Is_Cond_Call then
    -- For simple entry calls it is set to Uncond_Select
    QEL.State := not_queued, not_started
    goto <<fin>>
  end if;
  Add QEL to appropriate entry queue and increment 'COUNT
  QEL.state := queued

  if 'COUNT not used in a barrier then
    goto <<fin>>
  else
    goto <<Scanqueues>>
  end if;
else
  ASSERT: Barrier evaluates to TRUE
  if QEL.should_claim then
    Claim(QEL.SH.claim_bit)
    if claim fails then
      QEL.state := cannot_start
      goto <<fin>>
  end if;

  Mark selected alternative in SH
  Do entry body
  if exception then remember it in QEL
  QEL.state := done
end if;

<<Scanqueues>>
Scan_Queues  -- may be optimized
ASSERT: No entries remain with true barriers and waiting callers

<<fin>>
Release_Pr
if exception then raise it
return QEL.state
```

Discussion: After contending for the lock, the barrier expression is evaluated, and if TRUE, the entry body will be executed. In general, the barrier expression needs to be evaluated only if the corresponding queue is empty (this latter check is safe since the object is locked). It cannot be the case that the queue is non-empty and the barrier is TRUE. (This invariant has to be maintained in the presence of exceptions and aborts.) However, based on the proposed queue structure, it is faster to first evaluate the guard, and only if FALSE, to actually walk the queue and place the *Qel_Rec* in the appropriate location.

If an exception is raised while evaluating the barrier expression, the lock should be released, and the exception propagated to the caller. (It cannot be the case that other callers exist on that entry if the evaluation raises an exception. They must have been removed before.)

If an exception is raised while executing the entry body, the exception should be propagated to the caller, after scanning the queues, since the state of the protected record might have been changed.

If the QEL.State equals "queued", the task suspends itself using *Wait(SH.Op_Completed*).

5.4.6.3 Multi-Way Select

A multi-way **select** may have a final part (**else** or **in** part) which determines when the event alternatives are cancelled and what processing is done if none of the alternatives is selected.

An enumeration type *Select_Kind* is defined with the following three literals: *Async_Select* (**in**), *Cond_Select* (**else**), and *Uncond_Select* (no final part).

We assume that before calling multi-way **select**, the compiler does the following: Evaluates all actual parameters for all entry calls, and puts them in the corresponding parameter blocks. For each alternative, a *Qel_Rec* object is created and initialized. All *Qel_Rec*'s will be either in the *Select_Header* or pointed to by it. Alternatives are processed based on their lexical order.

The processing at the end of the **select** (we refer to it as the ''**select** completion'') is described separately. Note that after the **select** is complete, control is transferred to the sequence of statements of the selected alternative.

5.4.6.4 Processing Alternatives

Each alternative proceeds as follows:

```
Result := none
for each alternative loop
   QEL.Is_Cond_Call := FALSE
   Simple_Entry_Call
   case QEL.State is
      when not_queued not_started =>
         null
      when queued =>
         Increment pending callers counter
      when cannot_start =>
         Result := not_done
         exit
      when done =>
         Result := done
         exit
   end case;
end loop;

if done and exception then
   raise it
end if;
```

Discussion: If the loop exits with a *Done* indication, an alternative has been selected. However, since the entry body may be executed by another thread, or may end with a **requeue**, it is not guaranteed that on loop's exit the entry body was indeed finished. The calling task needs to suspend itself using *Wait(SH.Op_Completed)*.

Also, if the loop exits with a *Done* indication, all pending QEL's have to be removed from their corresponding queues.

Based on the *Select_Kind* and the loop exit indication, the processing proceeds as follows:

UNCOND_SELECT:

```
if Result = none or not_done then
   -- no alternative has been selected so far wait for Op_Completed
   -- if an alternative has been selected in the meantime,
   -- the wait will return immediately
end if;
ASSERT: SH claimed, Op_Completed set, An alternative was selected,
       or the task was aborted.
Remove all pending QEL's
if task aborted then
   start aborting
else
   return the index of the selected alternative
end if;
```

COND_SELECT:

```
ASSERT: if an alternative was chosen, the SH is claimed
attempt to claim;
if succeeded then
   ASSERT: no alternative was chosen
   Remove all pending QEL's
   Return the "else" indication
else
   Assert: An alternative was chosen or an abort requested
   wait for Op_Completed
   if abort was requested then
      start aborting self
   end if;
   Remove all pending QEL's
   Return the selected alternative index
end if;
```

Discussion: The reason for using the CLAIM mechanism is to ensure that the **else** part is not ‘‘selected’’ concurrently with another alternative.

An optimization of the above algorithm is for *Cond_Select* not to queue any call, but rather to try them in a sequence without leaving any *Qel_Rec* pending. This will save the need to cancel the pending *Qel_Rec*’s if none is selected by the time the **else** is encountered. This change is not entirely transparent to the user, since calls that would otherwise be selected might miss their chance. We are therefore interested in knowing how much simpler this change makes the implementation. Note that this will require special-case handling in the alternatives loop for the **else** case.

Also, since the selected alternative may use a **requeue** internally, there is another issue to consider. In the current approach (without the optional optimization), if an alternative is selected, the calling task must wait for the operation to complete, since it may be executed by another thread. Presumably, using the other approach would alleviate this need, since if an alternative is selected, it is guaranteed to be executed by the calling task itself, thus there is no need to wait until it completes. However, since the

corresponding entry may issue a **requeue** and then will not be able to proceed, there is always a need to wait for the operation to complete.

ASYNC_SELECT:

We refer to the sequence of statements after the **in** keyword as the *abortable region*. When the alternatives loop exits, the *Nested_Async_Level* is incremented in the TCB and the SH, and then the abortable region begins execution. (A check on the *Claim* bit may be done here to see if an alternative was chosen. However, this check is not safe; we can *NOT* set the bit yet. If no check is done, the abortable region will begin normally and, if an alternative was chosen, be immediately aborted.) The rest of the discussion addresses the needed processing of completing the abortable region.

When the abortable region completes:

```
Attempt to claim the SH
if succeeded then
  -- no other alternative was chosen so far
  -- (and none will be).
   Remove all pending QEL's
   Decrement Nested_Async_Level
   Return the appropriate indication
else
  -- An alternative was "just" selected
   Wait for Op_Completed
   if not already aborting to a higher level then
      record Aborting_To_Level
   end if;
   Remove all pending QEL's
   Decrement Nested_Async_Level
   abort self
end if;
```

5.4.6.5 Select Completion

A **select** statement can become completed in various ways depending on the *Select_Kind* and the reason for completion (see above). The *Select_Header* and QEL's should all be removed from their respective queues (if still on them), and their storage reclaimed. In the case of *Async_Select*, the finalization of the abortable region (if aborted) occurs *BEFORE* the **select** is completed. Also, the TCB.*Nested_Async_Level* should be decremented to reflect that the task is no longer abortable at that scope.

The selected alternative index (if any) should be saved and returned to the generated code. After the **select** is completed, control is transferred to the sequence of statements following the entry call of the selected alternative.

5.4.6.6 Requeue Handling

Two **requeue**s exist, with and without **abort**. The handling of both is mostly similar. In addition, two variations need to be addressed, a **requeue** on the same protected record (**requeue** on the same entry is just a normal case of this), and a **requeue** on another protected record. When the **requeue** is on another protected record the priority rules ensure that no deadlock can result; in addition, the current protected

record remains locked!

The original *Qel_Rec* is used for the **requeue**. (Most of the previously initialized information remains valid, and there is no need to reallocate a new *Qel_Rec*.) The indication about the target PR is updated, and the parameters are left in their original buffer (if the other entry is parameterless, they will simply be ignored). The family index is replaced, if needed. *Should_Claim* is set to FALSE, and the *Can_Abort* is set based on whether or not the **requeue** has **with abort**.

```
QEL.Should_Claim := FALSE
if not "with abort" then
  QEL.Can_Abort := FALSE;
else
  QEL.Can_Abort := FALSE;
end if;
if Target_PR is same as Current_PR then
  queue QEL
  goto <<Scanqueues>>
else
  QEL.Is_Cond_Call := FALSE
  Simple_Entry_Call
  if not done then
    Wait for Op_Completed
  end if;
  the SH is already updated accordingly
  remember exceptions if any
  if abort was requested then
    start aborting self
  end if;
<<Scanqueues>>
Scan_Queues (of enclosing PR)
```

5.4.7 Scan Queues

The following section describes the processing that occurs each time the state of a protected record instance changes (i.e. after a locking procedure or the execution of an entry body). Before the lock is released the implementation will scan all non-empty queues, evaluate corresponding barrier expressions based on the new state of the object, and perform all ‘‘ready’’ entry bodies on behalf of their original callers. Finally, each task whose entry call has completed is resumed (or aborted). Each queued call will be processed independently, i.e., after a candidate with a true barrier is chosen, its entry body is performed, and since the state of the object might have been changed, evaluation of barriers starts again.

```
Start from the top QEL in all queues
  -- Depending upon queuing order
loop
   Evaluate barrier;
   if FALSE goto <<next>>;
   if an exception then
      remember it
      remove qel
      op_done
      goto <<next>>
   end if;
   if needs to claim but cannot then
      -- When requeue, SH already claimed
      remove Qel;
      goto <<next>>
   else
      if can abort then
         if aborting to a higher level then
            remove qel
            op_done
            goto <<next>>
         end if;
      end if;
      ASSERT: SH is now claimed by this code
      remove qel
      mark alternative chosen in the SH
      perform entry body
      Save Exception if any;
      op_done
   end if;
   <<next>>
   get the next QEL from the queues
     -- Depending upon queuing order
end loop;
```

As mentioned above, we assume a model in which a task that modifies the state of a protected record instance is also responsible for performing the bodies of all waiting entry calls with a true barrier until no more such callers exist, before it releases the lock. We want to allow the program to rely on this model. However, other implementation techniques may be possible. In particular, for real-time implementations, it might be possible for the state-changing task to perform the operations of only higher priority callers, and to let the rest execute later, but without releasing the lock.

5.4.8 Op_Done, Abort and Finalization

The *Op_Done* procedure is called after an entry body or a protected procedure is completed. It should check whether it was called on itself (i.e., the executing task is the original caller) or on behalf of another task.

```
if SH is async then
   update TCB.Aborting_to_Level if lower
   for all nested SH's loop   -- see: (*1)
      attempt to claim SH
      SH.Should_Claim := TRUE
       -- to prevent further alternatives from
       -- being selected.  Each time a CLAIM
       -- fails, the abort request should be
       -- checked
      if succeeded then
         signal (Op_Completed)
      else
         do nothing
         -- as part of the corresponding select completion,
         -- the abort request will be "sensed"
      end if;
   end loop;
   if aborting self then
      if task deferring abort then
         do nothing; -- Will be handled when it becomes zero.
      else
         start aborting
      end if;
   else
      signal(Op_Completed)
      abort task  -- see: (*2)
   end if
else -- not an async SH
   if self then
      start finalizing
   else
      signal (Op_Completed)
   end if;
end if;
```

(*1) The **abort** statement will claim all SH's currently present.

(*2) If the task is running this call will cause it to stop its current processing and transfer control to a special abort handler. The abort handler (executing in the context of the task itself) should check if the task is still in a protected region and if so, should exit (the task will sense the abort request when it exits the last protected region). Otherwise, finalization to the appropriate scope should commence. The finalization code checks the *Aborting_To_Level* field in its own TCB, and either calls the normal finalization or the **abort**'s exception handler.

When a task aborts itself to a specific level, all nested SH's have already been claimed. It then marks the entire sub-tree of dependent tasks as **ABNORMAL** and then starts finalization from the deepest not-yet-finalized scope up. In each scope, the **abort**'s exception handler (if it exists) will be invoked.

5.4.9 User-Defined Timers

5.4.9.1 Delay Statement

Delay statements are transformed as discussed in 5.3.5. The allocation and deallocation of DQEL's are always done by the compiler, including the creation of the PR in the DQEL (this PR has canonical *Wait* entry and *Signal* procedure). The bodies of the *ENQUEUE* and *DEQUEUE* procedures are implemented either by the RTS (for the predefined delays) or by the program.

5.4.9.2 Delay Alternatives

The basic transformation is the same. The compiler calls *ENQUEUE* before calling the RTS' multi-way **select**, and *DEQUEUE* afterwards. The *ENQUEUE* is always called for each delay alternative without checking whether previous delays have expired. Similarly, all the *DEQUEUE* procedures are called at the end of the **select** construct, regardless of which one was selected. (The *DEQUEUE* may be implemented as a ''no-op'' if the corresponding timer has expired). The *DEQUEUE* subprogram and the code (user/RTS) that actually signals the timer expiration, should coordinate regarding the actual removal of a DQEL from its corresponding queue.

After this transformation, a delay alternative is treated and processed as any other entry call alternative.

5.4.10 Ada Types

This section includes the Ada specification of the types that would be created by the compiler for implementing protected records (in the canonical implementation model).

```
package Common_Types is

   -- The following are created by the compiler, for each protected type.

      type Storage_Element is range 0..2**System.Storage_Unit-1;
      type Storage_Array is array(Positive range <>)
        of Storage_Element;

      type Entry_Name_Index is range 0..Integer'LAST;

      type Entry_Family_Index is range 0..Integer'LAST;

      type Alt_Index is range 0..Integer'LAST;

      type Entry_Desc_Rec is
        -- Unique identification of a single entry
        -- or member of entry family
          record
              Entry_ID: Entry_Name_Index;
              Member_ID: Entry_Family_Index;
                -- "0" means not in a family
          end record;

      type PR_Rec (Num_Entries: Entry_Name_Index;
                   Component_Size: Natural);
        -- Protected record structure
      type PR_Access is access PR_Rec;

      type Param_Access is access ...;
        -- A pointer to the parameter area

      type Barrier_Function is access function (
        Pr_Data: PR_Access;
        Family_Index: Entry_Family_Index
      ) return Boolean;
        -- This is a general access pointer to a Boolean
        -- function.  The two formal parameters are the
        -- address of the protected record and the family
        -- member index.  The function body itself is
        -- generated by the compiler.

      type Operation_Procedure is access procedure (
        Pr_Data: PR_Access;
        Family_Index: Entry_Family_Index;
        Parameters: Param_Access
      );
        -- This is a general access pointer to a procedure.
        -- The formal parameters are the address of the
        -- protected record, the family member index, and
        -- a pointer to the parameters area.

      type Barrier_Pointers_Array is array
        (Entry_Name_Index range <>) of Barrier_Function;

      type Operation_Pointers_Array is array
        (Entry_Name_Index range <>) of Operation_Procedure;

      type Pr_Type_Desc (Num_Entries: Entry_Name_Index) is
        -- A descriptor for each unique protected
        -- record type.  Filled in by the compiler
          record
              Barriers:
                Barrier_Pointers_Array (1..Num_Entries);
```

```
        Operations:
          Operation_Pointers_Array (1..Num_Entries);
    end record;
type P_Pr_Type_Desc is access Pr_Type_Desc;

type Exclusion_Enum is (
  None,
  Non_Interruptible,
  Maskable,
  Non_Preemptable
);
  -- This is an enumeration of the various ways to
  -- achieve the mutual exclusion of access to the
  -- protected record instance (derived from
  -- defaults, pragmas, and priorities).

type Exclusion_Info is
    record
        Exclusion: Exclusion_Enum;
        Interrupt: Interrupt_ID;
          -- If a specific interrupt has to be
          -- disabled/masked when the protected record
          -- instance is accessed (determined when the
          -- object is created), the Interrupt_ID will
          -- be recorded here.
        Test_And_Set_Word: Integer;
        Lock_Pr: System.Priority;
          -- The lock's ceiling priority
        Owner: TCB_ID;   -- The owning task of this QEL.
    end record;

type Qel_Rec;
type Qel_Access is access Qel_Rec;

type Double_Link_Rec is
    record
        For_Link: Qel_Access;
        Back_Link: Qel_Access;
    end record;

type PR_Rec (
  Num_Entries: Entry_Name_Index;
  Component_Size: Natural
) is
  -- This is the data structure created by the compiler
  -- per each protected record instance
    record
        Type_Desc: P_Pr_Type_Desc;
          -- A pointer to the protected record's
          -- type descriptor
        Ex: Exclusion_Info;
        Queue_Head: Qel_Access;
          -- This queue may be FIFO or priority-based,
          -- depending on the queuing policy in effect.
        Components: Storage_Array(Component_Size);
          -- The actual declared components of the
          -- protected record.
    end record;

type Select_Header (Alt: Alt_Index);
type SH_Access is access Select_Header;

type Qel_Rec is
```

```
        record
            Header_Ptr: SH_Access;
            Which_Pr: PR_Access;   -- Which PR is called
            Entry_Desc: Entry_Desc_Rec;
            Parameters: Param_Access;
            Should_Claim: Boolean;
               -- True for entry calls, False for requeue
               -- since already claimed.
            Can_Abort: Boolean;
               -- Set for entry call and abortable requeue
               -- False otherwise.
            Select_Alt_Index: Alt_Index;
            Exception_Raised: Exception_ID := No_Exception;
               -- If exception was raised during the
               -- barrier evaluation or OP (entry body)
               --  execution.
            Next_Entry: Double_Link_Rec;
               -- link across row of entries
            Next_Caller_Same_Entry: Double_Link_Rec;
               -- link down column of callers
        end record;

    type Qel_Array is array (Alt_Index range <>)
      of Qel_Rec;

    type Select_Kind is (
      Uncond_Select, Cond_Select, Async_Select);

    type Select_Header (Alt: Alt_Index) is

        -- All parameters have been evaluated by now,
        -- and are pointed to by the corresponding
        -- Param_Area.  Alternatives are processed
        -- based on their lexical order.

        record
            Kind: Select_Kind;
            Claimed: Boolean;
               -- To resolve the race among multiple
               -- alternatives attempting to start at
               -- the same time.
            Nested_Async_Level: Natural := 0;
               -- A nested "select-in" counter
            Op_Completed: Boolean;
               -- Will be set after the OP
               -- May result in a race too
            Alt_Chosen: Alt_Index;
            Owner: TCB_ID;
            No_Of_Pending_Qels: Alt_Index := 0;
               -- Needed for clean-up if interrupted
            For_Link: SH_Access;
            Back_Link: SH_Access;
            Alternatives: Qel_Array (1..Alt);
        end record;

    type Deferring_Type is ...;
    type TCB is
        record
            Deferring_Abort_Info: Deferring_Type;
            Aborting_To_Level: Natural := Integer'LAST;
            Exception_Raised: Exception_ID := No_Exception;
            Nested_Async_Level: Natural := 0;
            Select_Headers_Chain: P_Select_Header;
```

```
            -- Other stuff
        end record;

    type TCB_ID is access TCB;

end Common_Types;
```

I. Suggested Code for the Implementation Prototype

The following code represents a slightly earlier version of the implementation model. We include it here for reference purposes only. It does not exactly match the description above.

```
package Callable_By_GC is

   -- Only the subprograms in this package are
   -- visible to the compiler.

    procedure Selective_Entry_Call (Sel_Desc: Select_Header);

    procedure Complete_Abortable_Region (
     Sel_Desc: SH_Access; Succeeded: out Boolean);
        -- Called by the compiler before the IN-Sequence
        -- is about to finish. Must Claim the SH.

    procedure Requeue_Same (Qel: Qel_Access);
        -- Called by the generated code when the
        -- requeue is on the same PR.

    procedure Requeue_Another (Qel: Qel_Access);
        -- Called by the generated code when the
        -- requeue is on another PR.

    procedure Abort_Handler;
       -- Not actually callable by the GC but rather
       -- invoked in the context of the program.

end Callable_By_GC;

package Operations is

     function Do_Or_Queue (Qel: Qel_Access) return Boolean;
        -- Based on the info in Qel (PR, entry, parameters,
        -- etc.), either does the appropriate OP, or just
        -- queues the QEL on the corresponding queue.
        -- Returns True if queued, False otherwise.
        -- Called by Selective_Entry_Call and Requeue_Same.

     procedure Scan_Queues (P: in out PR_Access);
        -- This is called after a state change of the
        -- PR (i.e. the epilogue of a "writing" OP).
        -- Called by Do_Or_Queue.

     procedure Op_Done (Qel: Qel_Access);
        -- Called by Scan_Queues and Do_Or_Queue to
        -- indicate that the operation of that Qel has
        -- been done, and the owner task has to be resumed.

     function Barrier_Is_True (Qel: Qel_Access) return Boolean;
        -- Evaluates the barrier of the corresponding
        -- PR/entry.  May raise an exception (if occurs
        -- during evaluation).  Called by Do_Or_Queue and
        -- Scan_Queues.

     procedure Perform_Entry_Body (Qel: Qel_Access);
        -- Executes the body of the corresponding PR/entry.
        -- May raise exception (if occurs in the body).
        -- Called By Do_Or_Queue and Scan_Queues.

     function Claim (Header: SH_Access) return Boolean;
        -- Attempts to "claim" the select header, returns
        -- True if successful (i.e. not claimed before),
        -- False otherwise.

     function Find_Queue (
       PR: PR_Access;
```

```
        Index: Entry_Desc_Rec
      ) return Double_Link_Rec;
        -- Follow the "Next_Entry" list and returns
        -- the corresponding head of queue.

      procedure Complete_Select (Sel_Desc: SH_Access);
        -- Remove QEL's from their queues, adjust TCB.

  end Operations;

  package body Operations is

      function Do_Or_Queue (Qel: Qel_Access) return Boolean is

          Sel_Head: SH_Access renames Qel.Header_Ptr;
          TCB: TCB_ID renames Sel_Head.Owner;

      begin

          Lock (Qel.Which_Pr);

          if Barrier_Is_True (Qel) and then
           Find_Queue(Qel.Which_Pr, Qel.Entry_Desc).For_Link =
            Find_Queue(Qel.Which_Pr, Qel.Entry_Desc).Back_Link then
              -- The above checks for an empty queue.
              -- Should be optimized!
              if not Qel.Should_Claim or Claim(Sel_Head) then
                -- No one of the previous QEL's was satisfied
                -- or the QEL "belongs" to a requeue.

                   Sel_Head.Alt_Chosen := Qel.Select_Alt_Index;
                   -- We would like to optimize away this
                   -- check for the "normal" (i.e. no abort
                   -- case).
                   if Qel.Can_Abort and then TCB.Aborting_To_Level /=
                    Integer'LAST and then TCB.Aborting_To_Level <=
                     Sel_Head.Nested_Async_Level then
                       Sel_Head.Op_Completed := True;
                       if TCB /= My_TCB then
                          Op_Done(Qel);
                       else
                          -- Op_Done stuff for self will be
                          -- performed at the caller after
                          -- return.
                           null;
                       end if;
                       -- No need to Scan_Queues since no
                       -- components have been touched yet.
                       Unlock(Qel.Which_Pr);
                       -- Start finalizing(abort).
                       return True;
                   end if;
                   begin
                       Perform_Entry_Body (Qel);
                   exception
                     -- Exception while performing OP.
                       when others =>
                           Qel.Exception_Raised := Current_Exception;
                   end;

                   -- Note that the state of the PR may have changed
                   -- even when exception was raised.
                   Sel_Head.Op_Completed := True;
```

```
                if TCB /= My_TCB then
                    Op_Done(Qel);
                else
                    -- Op_Done stuff for self will be
                    -- performed at the caller after return.
                    null;
                end if;
                Scan_Queues (Qel.Which_Pr);

            else
               -- Already claimed, do nothing
               null;
            end if;
            Unlock (Qel.Which_Pr);
            return False;

        else -- The condition is False or callers exist
             -- on this entry, which means the same.

            -- The following checks is similar to the one
            -- above (before doing the OP).  We may want
            -- to combine the two.
            if Qel.Can_Abort and then
             TCB.Aborting_To_Level /= Integer'LAST and then
              TCB.Aborting_To_Level <=
               Sel_Head.Nested_Async_Level then
                Sel_Head.Op_Completed := True;
            else
                Add(Qel, Qel.Which_Pr.Entry_Desc);
            end if;
            Unlock (Qel.Which_Pr);
            return True;
        end if;

    exception
       -- Exception when evaluating the barrier, we need to reraise the
       -- exception in the caller.

        when others =>
            Qel.Exception_Raised := Current_Exception;
            Unlock(Qel.Which_Pr);
            return True;

    end Do_Or_Queue;

    -- ================================================

    procedure Scan_Queues (P: in out PR_Access) is

       -- This is called after a state change of the PR
       -- (i.e. the epilogue of a "writing" OP) with QEL
       -- not-yet claimed).

       -- P is locked throughout this code !
       -- (Note that this code may raise exceptions in the caller.)

        Temp_Qel: Qel_Access;

    begin

       loop

           -- Find the caller at the head of the queue
```

```
-- with a True condition.
-- We iterate through the queue(s) which are
-- ordered according to the queuing policy
-- in effect.  Since we check barriers as
-- well, we need to continuously advance the
-- iterator.  We don't show all of it here.

Initialize_Iterator(P.Queue_head);
Temp_Qel := Top(P.Queue_Head);

while Temp_Qel /= null loop
   begin
      exit when Barrier_Is_True (Temp_Qel);
   exception
     -- Exception was raised while evaluating
     -- the barrier.  Multiple exceptions per
     -- TCB are not covered here.

      when others =>
         if Claim(Temp_Qel.Header_Ptr) then
           -- Not claimed before
            Temp_Qel.Exception_Raised :=
              Current_Exception;
            Temp_Qel.Header_Ptr.Alt_Chosen :=
              Temp_Qel.Select_Alt_Index;
            Temp_Qel.Header_Ptr.Op_Completed := True;
            Op_Done (Temp_Qel);
         end if;
   end;
   Temp_Qel := Next(P.Queue_Head);
end loop;

if Temp_Qel = null then
   -- No waiters
   return;
end if;

if not Temp_Qel.Should_Claim or else
   Claim (Temp_Qel.Header_Ptr) then
     -- Not claimed before
   Remove_Qel;
   Temp_Qel.Header_Ptr.Alt_Chosen :=
    Temp_Qel.Select_Alt_Index;
   declare
      Sel_Head: SH_Access renames Temp_Qel.Header_Ptr;
   begin
      if Temp_Qel.Can_Abort or else
       Sel_Head.Owner.Aborting_To_Level = Integer'LAST
        or else Sel_Head.Owner.Aborting_To_Level >
          Sel_Head.Nested_Async_Level then
           -- Can't be equal
           begin
              Perform_Entry_Body(Temp_Qel);
           exception
             -- Exception was raised while performing
             -- the OP on behalf of a task.
              when others =>
                 Temp_Qel.Exception_Raised :=
                  Current_Exception;
           end;
      end if;
      Sel_Head.Op_Completed := True;
      Op_Done (Temp_Qel);
```

```
            end;

         end if;

      end loop;

      -- When we come here eventually, the protected
      -- record is left with all of its queues either
      -- empty or their conditions False.

   end Scan_Queues;

   -- ================================================

   procedure Op_Done (Qel: Qel_Access) is
     -- Note that the PR pointed to by the Qel is locked here !

      Other_TCB: TCB_ID renames Qel.Header_Ptr.Owner;

   begin

      Other_TCB.Exception_Raised := Qel.Exception_Raised;

      if Qel.Header_Ptr.Kind = Async_Select and then
       Qel.Header_Ptr.Nested_Async_Level >
        Other_TCB.Aborting_To_Level then
         Other_TCB.Aborting_To_Level :=
          Qel.Header_Ptr.Nested_Async_Level;
         -- Claim all deeper SH's, if not yet claimed
         declare
            Temp_SH: SH_Access := Qel.Header_Ptr.Back_Link;
         begin
            while Temp_SH /= null loop
               if Claim(Temp_SH) then
                  Temp_SH.Should_Claim := True;
                    -- To prevent requeue's from proceeding
                  Temp_SH.Op_Completed := True;
               end if;
               Temp_SH := Temp_SH.Back_Link;
            end loop;
         end;
      end if;

      if Other_TCB.Deferring_Abort_Info /= 0 then
         return;
      end if;

     if Other_TCB = My_TCB then
        Start_Finalize(Qel.Header_Ptr);
        return;
     end if;

     if Other_TCB.Suspending then
        Signal(Qel.Header_Ptr.Op_Completed);
     elsif Qel.Header_Ptr.Kind = Async_Select and then
       Qel.Header_Ptr.Nested_Async_Level <
        Other_TCB.Nested_Async_Level then
        -- The level is already recorded in the TCB
        Abort_Task(Other_TCB); -- The low-level stuff
     end if;
   end Op_Done;

   -- ================================================
```

```
        function Barrier_Is_True (Qel: Qel_Access) return Boolean is
        begin
            return Qel.Which_Pr.Type_Desc.Barriers
             (Qel.Entry_Desc.Entry_ID)
              (Qel.Which_Pr.Components,
               Qel.Entry_Desc.Member_ID);
        end Barrier_Is_True;

        procedure Perform_Entry_Body (Qel: Qel_Access) is
        begin
            Qel.Which_Pr.Type_Desc.Operations(Qel.Entry_Desc.Entry_ID)
              (Qel.Which_Pr.Components,
               Qel.Entry_Desc.Member_ID,
                Qel.Parameters);
        end Perform_Entry_Body;

        function Claim (Header: SH_Access) return Boolean is
        begin
            if TAS(Header.Claimed) then
                return False;  -- Can't claim
            else
                return True; -- Successfully claimed
            end if;
        end Claim;

        procedure Complete_Select (Sel_Desc: SH_Access) is
        begin
           for J in 1..Sel_Desc.No_Of_Pending_Qels loop
              Lock (Sel_Desc.Alternatives(J).Which_Pr);
              -- Will check if the QEL has not been
              -- removed already (by comparing the
              -- two pointers).
              Delete (Sel_Desc.Alternatives(J),
                Sel_Desc.Alternatives(J).Which_Pr.Entry_Desc);
              Unlock (Sel_Desc.Alternatives(J).Which_Pr);
           end loop;
           My_TCB.Select_Headers_Chain := Sel_Desc.For_Link;
           My_TCB.Select_Headers_Chain.For_Link.Back_Link := null;
           if Sel_Desc.Kind = Async_Select then
              My_TCB.Nested_Async_Level := My_TCB.Nested_Async_Level - 1;
           end if;
           -- Finalization will take care of normal scope exit
           -- or abort
        end Complete_Select;

    end Operations;

    -- ================================================================

    package body Callable_By_GC is

       procedure Selective_Entry_Call (Sel_Desc: Select_Header) is

           Queued: Boolean;

       begin

           Sel_Desc.Owner := My_TCB;
           Sel_Desc.For_link := My_TCB.Select_Headers_Chain;
           Sel_Desc.Back_Link := null;
           My_TCB.Select_Headers_Chain := Sel_Desc;
           if Sel_Desc.Kind = Async_Select then
              My_TCB.Nested_Async_Level := My_TCB.Nested_Async_Level + 1;
```

```
            Sel_Desc.Nested_Async_Level := My_TCB.Nested_Async_Level;
        end if;

        For I in Sel_Desc.Alternatives'RANGE loop
            begin
                Queued := Do_Or_Queue (Sel_Desc.Alternatives(I));
            exception
                -- Exception while evaluating the condition
                -- or performing the OP
                when others =>
                    Complete_Select(Sel_Desc);
                    raise;
            end;

            exit when not Queued; -- Assert Claimed = True

            Sel_Desc.No_Of_Pending_Qels :=
             Sel_Desc.No_Of_Pending_Qels + 1;

        end loop;

        if Sel_Desc.Kind = Uncond_Select then
           if Queued then
              Wait(Sel_Desc.Op_Completed);
           end if;
           Complete_Select(Sel_Desc);
        elsif Sel_Desc.Kind = Cond_Select then
           if Claim(Sel_Desc) then
              Sel_Desc.Alt_Chosen := The_Else_Alternative;
           else
              -- Assert: Queued = True
              Wait(Sel_Desc.Op_Completed);
           end if;
           Complete_Select(Sel_Desc);
        else -- i.e. Kind = Async_Select

        -- We don't care if an alternative was chosen (in fact
        -- we cannot guarantee that we'll catch it on time).
        -- Furthermore, there is no special constant
        -- The_In_Alternative since writing it in Sel_Desc
        -- may override an existing indication.  Returning
        -- with no indication will tell the compiler that
        -- an IN sequence is in progress, and Op_Done will
        -- take care of races.

           null; -- !!
        end if;

    end Selective_Entry_Call;

    procedure Complete_Abortable_Region (
      Sel_Desc: SH_Access; Succeeded: out Boolean) is
      Temp: Boolean := Claim(Sel_Desc);
    begin
      Succeeded := Temp; -- True = no event had
                         -- previously interrupted
      if not Temp then
         Wait(Sel_Desc.Op_Completed);
      end if;
      Complete_Select(Sel_Desc);
    end Complete_Abortable_Region;

    procedure Requeue_Same (Qel: Qel_Access) is
```

```
        begin
            -- The PR is already locked
            Add(Qel, Qel.Which_Pr.Queue(Qel.Entry_ID));
        exception
            when others =>
                raise;
        end Requeue_Same;

        procedure Requeue_Another (Qel: Qel_Access) is
        begin
            if not Do_Or_Queue (Qel) and then
             Qel.Header_Ptr /= Async_Select then
               Suspend_Self;
            end if;
        exception
            when others =>
                raise;
        end Requeue_Another;

        procedure Abort_Handler is
        begin

            if My_TCB.Deferring_Abort_Info /= 0 then
               return;
            end if;

            -- Perform other state checks

            Complete_Select(How_Do_We_Get_The_SH);
            Start_Finalize(Self);   -- All deeper scopes
              -- finalization will stop when the first
              -- select in the TCB's list is exited.

        end Abort_Handler;

    end Callable_By_GC;
```

6. Exceptions and Frame Finalization

This Implementation module describes a model for implementing the changes proposed for chapter 11 of MD-3.0. That chapter is attached as an appendix to this module.

We are interested in feedback on the clarity and completeness of the semantic specification, as well as the implementation difficulty and feasibility of the proposed implementation models.

6.1 Basic Concepts

The new features which must be implemented as part of this module are

- The new types in SYSTEM — EXCEPTION_OCCURRENCE, EXCEPTION_TAG — and the operations on those types.
- Choice parameters on exception handlers.
- Exit handlers for abort, exit, and exception.

In order to explain our implementation model we introduce the concept of a finalization level. Each nested frame with finalization introduces a new finalization level. When finalization for a normal exit begins it is usually parameterized by the level out to which one must finalize before continuing.

Example:

```
procedure OUTER is
begin                   -- Level 0
    loop
        begin           -- Level 1
            INNER;
            exit;       -- Exit to Level 0.
        at end          -- Handler A
            when exit =>
                ...;
        end;
        MORE_STUFF;
    end loop;
    raise FOO_ERROR;
    ...
when others =>
    ...
at end              -- Handler B
    when exit | exception =>
        ... ;
end OUTER;
```

```
procedure INNER is
begin                   -- Level 0
    select
        P.E1;
        return;
    in
        begin           -- Level 1
        ... ;
        at end
            when abort =>
                ... ;     -- Handler C
        end ;
    end;
at end
    when exit =>
    ... ;           -- Handler D
end INNER;
```

In the program fragment, assume that the barrier for P.E1 becomes true while the final-part of the select statement is executing. The finalization handlers will be invoked as follows:

1. Handler C is invoked for an asynchronous exit.
2. The return statement following the call to P.E1 is executed.
3. Handler D is invoked for a normal exit.
4. The exit statement following the call to INNER is executed.
5. Handler A is invoked for a normal exit.
6. MORE_STUFF is *not* executed, and control is transferred to the **raise** statement.
7. The exception handler is invoked as in Ada 83.
8. Handler B is invoked for an exceptional exit if the exception handler propagates the exception.
9. Otherwise Handler B is invoked for normal exit if the exception handler does not propagate the exception.

6.1.1 New Types and Operations

SYSTEM.EXCEPTION_OCCURRENCE is an implementation defined private type, which must support the 'TAG attribute and the raise operation.

Implementations are free to support other operations on these types if it is convenient to do so, and the user team needs the functionality.

SYSTEM.EXCEPTION_TAG is an implementation defined private type, which must support the raise operation.

```
function EXCEPTION_NAME(E : in EXCEPTION_TAG) return STRING ;
```

is an operation defined in SYSTEM which returns the simple name of the exception whose tag corresponds with E. Exceptions renamings should be ignored by this function. In other words, the following program prints "TRUE".

```
procedure TEST is
    FOO_ERROR : exception renames CONSTRAINT_ERROR;
    -- FOO_ERROR'TAG = CONSTRAINT_ERROR'TAG
begin
    raise FOO_ERROR;
exception
    when X:FOO_ERROR =>
        if SYSTEM.EXCEPTION_NAME(X'TAG) = "CONSTRAINT_ERROR" then
            TEXT_IO.PUT("TRUE");
        end if;
end TEST;
```

6.2 Implementation Models for Exceptions

Regardless of whether an implementation uses a PC map or a list of active handlers to determine the location of an exception handler, the implementation model for the EXCEPTION_OCCURRENCE and EXCEPTION_TAG types is as follows:

An EXCEPTION_TAG is the address of a descriptor that provides information about the exception. At the very least the descriptor must contain the simple_name of the exception.

An EXCEPTION_OCCURRENCE is a structure which at the very least contains the EXCEPTION_TAG corresponding to the exception being raised. Other components of the structure are implementation dependent. Whatever is required to support the implementation defined operations of EXCEPTION_OCCURRENCE can be placed in the structure. For example, one might store the values of the machine registers at the time when the original exception was raised.

The model for allocating an exception occurrence depends on how an implementation cuts back the stack during exception propagation. If the stack is cut back before a handler is entered, then space for the currently propagating exception occurrence must be allocated in a per task data area (unless it fits in registers). If the implementation doesn't cut back the stack until after the EXCEPTION_OCCURRENCE is copied to the handler's frame, then the occurrence can be allocated on the stack at the point where the exception is raised.

Instances of EXCEPTION_OCCURRENCE are initialized when an exception is raised. Storage for such instances is also statically allocated within every frame containing an exception handler with a choice parameter or re-raise. You can think of this as allocating space for the choice parameter. When control is transferred to the handler in a frame, the exception_occurrence should be copied into that space. A possible option is to set aside registers for an EXCEPTION_OCCURRENCE (if they are small enough to fit in the set of registers which are not changed or restored during a subprogram's epilogue.

6.3 Implementation Model for Exit Handlers

This model for exit handlers assumes that there is one copy of the code for the handler. Control is transferred to this code when a frame with such a handler is abandoned. The type of exit, as well as some auxiliary information associated with the exit must be passed along to the exit handlers. The specific handler chosen is based on the type of exit. The propagation performed after the handler has executed is based on the other information passed along.

An exit handler for a normal exit receives control directly from the statement which caused the abandonment of the frame. Such an exit is parameterized by the address where the program should continue after all necessary finalizations are complete at the finalization-level where that address resides. At the end of a handler for normal-exit, the finalization-level parameter is compared with the finalization-level of the frame, and if they are equal, then control is transferred to the continuation address. If there are more levels to finalize before continuing, then the exit is propagated to the next level of exit-handler in a manner similar to exception propagation.

An exit handler for exceptional exits receives control similar to how an exception handler receives control. The implementation model is that such a handler is equivalent to a "when others" exception handler.

An exit handler for an asynchronous exit receives control from the run-time system. These handlers are parameterized by an indication of the outermost select-in construct being exited. Each select-in has an implicit finalization handler for asynchronous exits which checks the level and causes execution to continue at the appropriate select alternative when the desired select construct is reached.

We recommend that the compiler provide tables mapping the PC location to the address of the exit handler for that location. The RTS searches these tables, and transfers control to the appropriate handler. Many implementations already use a similar approach to exceptions. This was discussed in more detail in *Implementation Modules II*, in the description of type finalization.

Aborts are deferred within exit handlers if the frame has an abort handler. Therefore the first thing which occurs within a non-abortable exit handler is a call to inform the RTS that the current task has deferred aborts. This is probably done by bumping a counter in the TCB.

The sequence of statements in the handler are executed, followed by finalization of objects whose types have finalization declared in the frame being finalized. The RTS should then be informed that the task is again abortable (if it was on entrance to the handler), and finally the transfer of control that was interrupted by the handler is continued. The continuation depends on the type of exit being handled as discussed above.

6.3.1 Implementation Model For Aborts

Since finalization applies to **abort**, our model is that task aborts propagate like exceptions, and that an aborted task finalizes itself. Below we describe an implementation that generalizes the EXCEPTION_OCCURRENCE type into an EXIT_OCCURRENCE type. Handlers and propagation for all forms of exit are similar, and parameterized by an EXIT_OCCURRENCE. Other models are possible. In particular, finalization thunks could be created, and strung together as handlers are encountered. This would allow an aborted task to be finalized by the RTS or another task.

6.4 Implementation Strategies

6.4.1 Exception Declarations

We expect very few compiler modifications will be required in the area of exceptions. For the most part all implementations must be assigning some unique-id to every declared exception, and this need not change. Most systems can already determine the simple name of an exception given the unique-id, since that is a useful debugging tool, so the only work involved is making that function available to the user.

6.4.2 Raise Statements

There are two new forms of raise statements. One takes an exception tag, and the other an exception occurrence. The first is essentially identical to the existing raise routine. The only difference is that the value of the exception is a variable, instead of a constant. If exception raising is done by calling a "raise" routine in the Ada RTS with a parameter giving the unique-id of the exception being raised, then the same routine will work for raising an exception tag.

The "raise" routine most likely can be decomposed into two parts, one which allocates and initializes an exception occurrence, and one which finds the appropriate handler, and copies the exception occurrence into the registers or memory where the handler expects to find it. Raising an exception_occurrence corresponds to the second part of the "raise" routine. It is possible that some code re-organization will be needed to get this second part callable separately from the first.

6.4.3 Exception Handlers

The front-end must be modified to accept choice parameters. In order to make re-raising of an exception work properly compilers must already have some location where the current exception is stashed during a handler. The choice parameter should be viewed as a name for that (read-only) location. How this is accomplished will probably differ for every compiler, depending on where in the compiler the exception handling semantics are implemented.

6.4.4 Frames and exits

When compiling a frame with an exit handler, the compiler must increase the current finalization-level. When compiling an exit from a frame via a goto,exit, or return, the compiler must note the finalization-level of the exited construct, compared with the current finalization-level. If they are equal, then no finalization handlers apply, and the exit occurs as in Ada 83. If the current level is deeper than the level of the construct being exited, then an exit-occurrence should be constructed (see below), and control should be transferred to the innermost relevant finalization handler.

6.4.4.1 Exit Occurrences

An EXIT_OCCURRENCE is the parameter to an exit handler. An EXCEPTION_OCCURRENCE can be implemented as a constrained subtype of EXIT_OCCURRENCE.

```
type EXIT_KIND is (NORMAL_EXIT, EXCEPTION_EXIT, ASYNC_EXIT);

type EXIT_OCCURRENCE(KIND : EXIT_KIND) is
  record
    case KIND is
      when NORMAL_EXIT =>
        TARGET_LEVEL : FINALIZATION_LEVEL;
        FINAL_DESTINATION  : INSTRUCTION_ADDRESS;
      when ASYNC_EXIT =>
         -- identifies outermost "select"
         -- whose final part is abandoned.
         TARGET_SELECT: SELECT_HANDLE;


      when EXCEPTION_EXIT =>
        TAG : SYSTEM.EXCEPTION_TAG;
        ... ; -- Other Stuff;
    end case;
  end record;

subtype EXCEPTION_OCCURRENCE is EXIT_OCCURRENCE(EXCEPTION);
```

6.4.5 Exit Handlers

Within an exit handler, the compiler generates code similar to the following:

```
declare
   CURRENT_LEVEL : constant := Finalization Level
   CURRENT_EXIT_OCCURRENCE: EXIT_OCCURRENCE; -- initialized by RTS
   LAST_EXCEPTION : SYSTEM.EXCEPTION_OCCURRENCE := NO_EXCEPTION;
begin
   if ABORTS_ARE_HANDLED_IN_THIS_HANDLER then  -- known statically
      DEFER_ABORTS;
   end if;
   begin
       case CURRENT_EXIT_OCCURRENCE.KIND is
       -- These cases can be combined if the user has combined
       -- them in the source.
          when NORMAL_EXIT =>
            user's Code for handling normal exits
          when ASYNC_EXIT =>
            user's code for handling async exits
          when EXCEPTION_EXIT =>
            user's code for handling exception exits
       end case;
   exception
       when x:others =>
           LAST_EXCEPTION := x;
   end;
   if OBJECTS_NEEDING_FINALIZATION_EXIST then
       -- possibly known statically
       DEFER_ABORTS;
       -- for each OBJ needing finalization
       loop
           begin
              finalize object
           exception
              when x:others =>
                  last_exception := x;
           end;
       end loop;
       ALLOW_ABORTS;
   end if;
   if ABORTS_ARE_HANDLED_IN_THIS_HANDLER then  -- known statically
      ALLOW_ABORTS;
   end if;
   if LAST_EXCEPTION /= NO_EXCEPTION then
       if CURRENT_EXIT_OCCURRENCE.KIND /= ASYNC_EXIT then
           -- propagate LAST_EXCEPTION
       end if;
   end if;
   if CURRENT_EXIT_OCCURRENCE.KIND = NORMAL_EXIT and then
        CURRENT_EXIT_OCCURRENCE.TARGET_LEVEL = CURRENT_LEVEL then
          -- statically known
          jump to CURRENT_EXIT_OCCURRENCE.FINAL_DESTINATION
    else
       propagate CURRENT_EXIT_OCCURRENCE
   end if;
end ;
```

7. Generics

This Implementation module describes a model for implementing the changes proposed for chapter 12 of MD-3.0. That chapter is attached as an appendix to this module.

We are interested in feedback on the clarity and completeness of the semantic specification, as well as the implementation difficulty and feasibility of the proposed implementation models.

Generic formal unconstrained types strengthen the contract model, and hence, are not upward compatible. The workaround is to add (<>) as a formal discriminant part on generics whose instantiations no longer compile. We are interested in an analysis from the user teams of the severity of the upwards incompatibility and the difficulty in applying the workaround.

This chapter is less detailed than others, since the actual implementation strategies will most likely depend heavily on the compiler's particular approach to implementing generics.

7.1 Basic Concepts

The new features which must be implemented as part of this module:

Generic formal unconstrained types

> These types are specified by a (<>) as the formal_discriminant_part. A formal [limited] private or a formal derived type may be specified this way. Such types match any type which satisfies the other relevant matching rules. Within the generic it is illegal to derive from a such a type. If (<>) is not specified, then only actuals which can appear in the declaration of a variable/component (without an initial value) match the formal type.

Generic formal derived types

> These are the formals with definitions in the form "**is new** type_mark". We will call the type denoted by the type_mark the parent type of the formal. These formals match the parent type and any derivative of the parent type, including all the universal types. If the type is tagged, then the universal types only match against a formal unconstrained type. Complete matching rules are found in MD 12.3.6.
>
> Within the generic all the operations on the parent type are available for the formal. If the formal is tagged then in the instantiation the primitive operations of the formal are implemented by the corresponding primitive operations of the actual. If the formal is untagged, then the primitive operations of the formal are implemented with the primitive operations of the parent type. These rules are not discussed in the mapping document. We are interested in feedback regarding the applicability of the implementation model for these rules (described below) in the various implementations of generics.

Generic formal package instantiations

> Generic formal package instantiations are specified by "**with package** identifier **is new** generic_package_name (<>)". The actual to these types must name a package which is an instantiation of the generic package denoted in the formal definition (which we call the parent package.)
>
> Within the generic, the entities within the spec of the parent package, including the formal parameters, are available via selected name notation. In the instantiation, these entities are implemented by the corresponding entities in the actual generic instantiation.

Generic formal tagged types

Generic formal tagged types are specified as either "tagged [limited] private", or are generic formal derived types where the parent type is a tagged type. Such types only match actuals which are tagged types. Of formal types, only formal tagged types may be extended within a generic.

Extensions declared in the spec of an instantiation are considered to occur for scope-checking purposes at the level of the instantiation. Extensions within the body are considered to occur in a more deeply nested scope, thereby disallowing conversion to the universal type of the (tagged) parent.

7.2 Implementation Model for Generics

7.2.1 Formal Derived Types

Our model is that formal derived types act similarly to universal types. For tagged types the primitive operations within the instantiation are implemented by the primitive operations of the actual. In a universally shared generic, this could be implemented exactly as the dispatching operations are for a tagged universal type. In a non-shared generic, the dispatched-to operation is statically known, so the dispatch can be optimized away. For untagged types the primitive operations in the instantiation match the primitive operations of the untagged universal type, that is, those of the parent type.

Given this model, shared generics are simple to implement. The formal parent type can be substituted by the corresponding universal type, and the subsequent code will dispatch appropriately. This model may be appropriate for non-shared generics as well.

For non-shared generics, an alternative implementation would treat the formal as the parent type during semantic analysis on the generic, It would then substitute the actual for the formal during instantiation. For untagged types the operations identified during semantic analysis of the generic are correct for the instantiation. For tagged types, a substitution of an overridden primitive operation is required.

7.2.2 Formal Package Instantiations

Our model for implementing formal package instantiations is to create an instantiation of the parent package (with arbitrary actuals) and use that to perform semantic analysis of the generic. For shared generics, this should do the right thing, since all instantiations of the parent are implemented identically. For unshared generics, the subsequent instantiation of that generic must substitute the entities in the actual instantiation for the entities in the arbitrary instantiation used to compile the generic. We would expect this to be very similar to what must happen during the expansion of Ada 83 unshared generic bodies when the generic spec contains an instantiation.

7.3 Other Comments

7.3.1 Exceptions and Generics

In order to make it easier to share generic bodies, we propose to make it optional whether exceptions declared in the generic body are replicated in instantiations. It is very difficult to construct a situation where a replicated exception can be distinguished from a shared exception, we therefore believe the upwards incompatibility is unimportant.

7.3.2 Generic Contract Model

In order to provide a strong contract model without sacrificing useful features within a generic we propose that some checks done in the spec of the generic occur at instantiation time. Checks done in the body of the generic would occur when compiling the generic, and must assume a worst-case scenario. For instance, as we proposed above, scope-checks required on conversions to universal types (or using 'ACCESS) would depend on the actual scope of the instantiation when performed in the spec of the generic, but such checks must always assume that the instantiation is nested when performed in the body of the generic. A similar situation is that extension of a formal private type in the spec must check for component name clashes based on the actuals of the instantiation, but in the body the parent of a type extension can always be assumed to be a fully private type.

In general we favor this sort of differentiation, and are interested in UI feedback on the concept.

8. Distribution of an Ada Program

8.1 Overview

This module describes the proposed implementation technique for the 9X changes regarding the support of distribution. The approach of 9X is to define package categorization pragmas to facilitating compiler support for the construction of program partitions. The suggested model of communication among partitions is that of an RPC with compiler help for generating remote procedure calls messages, and reconstructing these messages back into parameter profiles. Finally an interface for a user-provided-communication-system (UPCS) is defined with the assumption that such a system will be built by the vendor or the user and be implementable in pure Ada.

8.2 Purpose

The purpose of this module is to check the implementability and usability of the 9x proposed changes regarding distribution. Specifically, we are interested in feed-back regarding the following topics:

1. Missing features or UPCS primitives.
2. The capabilities of the proposed changes with regard to building distributed system.
3. The ability to compose higher-level mechanisms, protocols, and distributed synchronization primitives.
4. The separation of the language changes from the UPCS and the (re)configuration tool. Its appropriateness, flexibility, efficiency, and completeness.
5. Since this area contains a relatively large number of implementation-defined areas, we expect this module to be developed with the user part of team feeding into the design phase of the changes.
6. We would like to receive a description of the target architecture for which the system is being prototyped: Interesting details include: the ISA, the mapping of the active/passive partition into the actual hardware, the communication system topology and capabilities, and limitations imposed on passive partitions (i.e. whether the CPU's have any limitations in performing the necessary instructions on memory space included in the passive partition).

As with previous modules we would also like to hear about:

1. The completeness of the changes and their description.
2. Problem areas and unclear semantics.
3. Unnecessary features.

8.3 Proposed Change

We will not repeat the proposed changes here, please refer to the Mapping Document version 2.0 (May 3, 1991) for the description of the changes. In particular, 10.1.2, 10.5, G.1.7, and G.3 contain relevant information.

8.4 Implementation

8.4.1 Introduction: A Distributed Application

A distributed application typically consists of several active partitions, at least one per physical node, and, if global memory exists in the system, one or more passive partitions representing the global memory. Partitions are the unit of (re)configuration, and hence there may more than one per physical node. The assumed model of a passive partition is that of a globally shared memory.

Passive packages may include code, so passive partitions may have code in them as well. However, we do not specify how code will be distributed (or replicated) in a system. In particular, we do not specify whether code of passive packages will physically reside in the corresponding passive partition or in the active partitions accessing its data (e.g. access functions for private types). This decision is left to the implementation-dependent configuration tools.

All communication between active partitions is assumed to be through either RPC messages, shared global memory represented by passive partitions, or additional implementation-defined interfaces included in the UPCS. We assume that there is no direct addressability from one active partition to data declared in another active partition.

8.4.2 Implementation-Defined Limitations on Passive Partitions

All operations on objects defined within a passive partition (e.g. atomic load and store of composite objects, locking protected records(test-and-set) across passive partition boundaries, and executing code) are assumed to be available to the program. However, since there is a wide variance of distributed architectures, and passive partitions can be used to describe different configurations, we assume that certain limitations will be imposed by implementations on the allowable operations on objects within these partitions (in addition to restriction on the residence of such objects). These limitations must be documented.

8.4.3 Dynamic (Re)Configuration

Since there is no specific language support for the actual (re)configuration of a system, and this is assumed to be handled by a user-provided configuration subsystem, we are interested to know what are the additional capabilities which are provided by the implementation to:

1. Group packages into partitions and name them.
2. Load and reload a partition into a node.
3. Dynamic addition or deletion of partitions.
4. Initiate or re-initiate the elaboration of a partition’s load module or an entire node.
5. Methods for replicating partitions among multiple nodes.
6. Mechanisms to abort partitions.
7. Under which conditions COMMUNICATION_ERROR will be raised.
8. Other errors which are detected by the UPCS.
9. Recovery actions for the various errors.

8.4.4 Compile-Time Checking

The compile-time checking required for supporting the specified package categories is relatively straightforward. The restrictions generally only limit the categories of other packages which may be "with"ed, and whether library-level variables may be declared. Remote call interface packages also disallow the declaration and use of limited types in the visible part. Given these restrictions, there is no special processing required when analyzing statements or expressions, or the bodies of subprograms.

8.4.5 RPC Stub Generation

Calls to subprograms defined in a REMOTE_CALL_INTERFACE (RCI) will be generated normally, i.e. the call-site will look like any other procedure call. Subprograms in RCI package bodies, will similarly be generated normally, i.e. their prologue and epilogue will be the same as if placed locally.

When compiling the specification of an RCI package, the compiler will generate the appropriate stubs per each visible subprogram, in addition to a dispatcher. Each stub will be called as a normal subprogram; it will allocate a message object (using UPCS interfaces, see below), "flatten" all the IN [OUT] parameters, write the byte stream into the message object, and then call DO_RPC (or DO_DATAGRAM, if supported), to send the message and wait for a reply. When a reply arrives, the stub will unpack the [IN] OUT parameters from the incoming message buffer into the formals and return. If an exception is raised by the remotely called subprogram, it will be propagated by DO_RPC within the UPCS after doing any necessary clean-ups (i.e. the UPCS will extract the exception ID from the message and raise the exception. It will not be necessary for each stub to contain an exception handler.)

Function results are treated as OUT parameters in the above discussion.

When compiling the body of an RCI package, the compiler will build a dispatcher which will be called from the UPCS when an RPC is received for this package. The dispatcher will look at the subprogram index encoded in the message (by the matching stub), and dispatch to code specific to the subprogram which will extract the parameters, pass them to the corresponding subprogram, gather up the results and return a reply to the UPCS. If an exception is raised by the subprogram, it will be caught by the UPCS on the way back. The UPCS will then 'WRITE the exception name with the associated information into the reply message buffer and send it back to the original caller, instead of the [IN] OUT parameters.

8.4.5.1 A Canonical Stub at the Caller side

We assume that each partition will use the following values:

1. *Partition_ID*: One per partition, assigned by the linker (or the configuration tool). Used for the *'PARTITION_ID* attribute.
2. *Package_ID*: A per package value, assigned in by the linker, and used for the *'PACKAGE_ID* attribute.
3. *Subp_ID*: Assigned by the compiler to uniquely identify each subprogram in a given RCI package.

```
procedure Example (X : in X_Type; Y : in out Y_Type; Z : out Z_Type);

   Out_Message : (aliased) UPCS.RPC_Stream_Type (
     Size => UPCS.Max_Out_Message);
     -- Or specific size or "0" for unlimited

   In_Message : Stream_Support.Stream_Access;
     -- Just a pointer; The UPCS will allocate the
     -- appropriate sized message when the reply
     -- arrives.

begin
   -- IN parameters
   X_Type'WRITE(Out_Message'ACCESS, X);

   -- IN OUT parameters
   if not Y_Type'CONSTRAINED and then
    Y_Type has defaults for all discriminants then
      -- Send the Y'CONSTRAINED attribute
      Boolean'WRITE(Out_Message'ACCESS, Y'CONSTRAINED);
   end if;
   Y_Type'WRITE(Out_Message'ACCESS, Y);

   -- OUT parameters
   if not Z_Type'CONSTRAINED and then
    Z_Type has defaults for all discriminants then
      -- Send the Z'CONSTRAINED attribute
      Boolean'WRITE(Out_Message'ACCESS, Z'CONSTRAINED);
   end if;
   if not Z_Type'CONSTRAINED then
      -- Assume Z_Type has discriminants ZD1, ZD2, ..., ZDn of types
      -- ZD1_Type, ZD2_Type, ..., ZDn_Type respectively
      ZD1_Type'WRITE(Out_Message'ACCESS, ZD1);
      ZD2_Type'WRITE(Out_Message'ACCESS, ZD2);
      ...
      ZDn_Type'WRITE(Out_Message'ACCESS, ZDn);
   end if;

   UPCS.DO_RPC (Out_Message'ACCESS, In_Message,
     PTN_ID => THIS_PKG'PARTITION_ID,
     PKG_ID => THIS_PKG'PACKAGE_ID,
     Subp_ID => Example'Subprogram_ID
   );
     -- If exception is raised here, Out_Message has been
     -- finalized already by UPCS.  The exception will
     -- simply be propagated to the caller.

   Y := Y_Type'READ(In_Message);
   Z := Z_Type'READ(In_Message);
   -- Necessary finalization happens here
end Example;
```

8.4.5.2 On the Receiving Side

On the receiving side, the dispatching to the appropriate subprogram will be done in two steps. First, the linker will build a dispatch table (or case statement) in each partition which contains an RCI package. The UPCS will index into this table using the *Target_Pkg* component in the incoming message. For each RCI package, the compiler will build another dispatch table to reach the appropriate subprogram wrapper based on the *Target_Subp* in the message. (These two dispatching tables may of course be combined.)

Such a wrapper will look like the following: (we use the same *Example* procedure as above.)

```
Dispatch_Table : array (Package_Ids) of Package_Dispatch_Access :=
  (..., This_Package'PACKAGE_ID => This_Package_Dispatch'ACCESS, ...);
-- A per Partition array

type Wrapper_Access is access procedure (
  In_Message : Stream_Support.Stream_Access;
  Out_Message : out Stream_Support.Stream_Access
);

This_Package_Dispatch : array (Subp_Ids) of Wrapper_Access :=
  (..., Example_Wrapper'SUBP_ID = > Example_Wrapper'ACCESS, ...);
-- A per package array

procedure This_Partition_Dispatcher (Package_ID : Natural;
                      In_Message : Stream_Support.Stream_Access;
                      Out_Message: out Stream_Support.Stream_Access
                     ) is
begin
    Dispatch_Table(Package_ID)(In_Message.Target_Subp)
      (In_Message, Out_Message);

end This_Partition_Dispatcher;

procedure Example_Wrapper (
  In_Message : Stream_Support.Stream_Access;
  Out_Message : out Stream_Support.Stream_Access
) is

    X : constant X_Type := X_Type'READ(In_Message);
    Y : Y_Type := ('CONSTRAINED => Boolean'READ(In_Message),
      data => Y_Type'READ(In_Message));
    Z : Z_Type('CONSTRAINED => Boolean'READ(In_Message),
      ZD1 => ZD1_Type'READ(In_Message),
      ZD2 => ZD2_Type'READ(In_Message),
        . . .
      ZDn => ZDn_Type'READ(In_Message)
    );
begin
    Example (X, Y, Z);
    Y_Type'WRITE(Out_Message, Y);
    Z_Type'WRITE(Out_Message, Z);
end Example_Wrapper;
```

8.4.6 Link-Time Partitioning

At link time (or earlier), each remote-call-interface package and each passive package will be associated with some active or passive partition, respectively. Each active partition will be assigned a partition identifier by the configuration tool, and each remote-call-interface package within the partition will be assigned a package identifier. These will be accessible from within a program via the attributes pkg_name'PARTITION_ID and pkg_name'PACKAGE_ID. These identifiers will be included in the header of the messages built by the stubs to identify the target partition and package for the RPC. The partition identifier will be used by the UPCS to route the message to the correct partition.

The code associated with an RCI package body is always included in the remote partition to which it is directed. The code for the RPC stubs (generated by compiling the RCI package specification) is included

in the load-module of each partition which calls the remote subprogram. (Obviously, the code associated with the calling-site is only included in the calling partition.)

When both the body and spec of a remote-call-interface package are included in the same partition, the RPC stubs are bypassed, and the call goes directly to the associated body. When only the spec of a remote-call-interface package is included in a partition, the RPC stubs take the place of the package body.

8.4.7 Dispatching Tables

Each active partition which contains one or more RCI packages will have a two-level dispatching table. The first level, the "package-level", will be built by the linker when it combines RCI packages into active partitions. When a message arrives, the *Package_ID* encoded in the message header, will be used as an index to this table. The second level, the "subprogram-level", will be built by the compiler for each package and it will be indexed by the *Subp_ID* of the visible subprograms. Note that at this point, the *Partition_ID* is no longer relevant (it is a constant) since the network will deliver messages to the appropriate partitions only. The latter dispatch table may be implemented as a **case** statement.

8.4.8 Run-Time Support

We assume the presence of a network layer that supports asynchronous send and receive primitives, a way to notify a caller when a message (or a reply) is received via an interrupt, and an identification of partitions. We also assume that interrupts will be handled using protected record procedures.

8.4.8.1 On the Calling Side

After the message is created and filled-in as described above, it will be sent over the network to the appropriate partition address. The calling task then will be suspended waiting for a reply. When a reply comes in, an interrupt will be generated, thus invoking the interrupt handler which will release the waiting task. If the reply indicates that an exception is to be raised, then an exception is raised in the calling task at the point of the call. Otherwise it returns to the stub, which should return the function result or [IN]-OUT parameters.

If the calling task is aborted while waiting for a reply, then an abort handler must send an RPC cancel message to the receiving partition.

Timed services may be implemented using the asynchronous select construct. Expiration of the delay will invoke the finalization code of the interruptible code, and this in turn will send the cancel message.

8.4.8.2 On the Receiving Side

In order to allow for parallelism in servicing RPC requests, we assume that each active partition which is capable of receiving RPC requests will have a pool of agent tasks. The size of this pool will either be configurable by the user or be implementation-defined. We also assume that each task will service requests in a loop (i.e. one iteration per each request) and will not be recreated/aborted for each incoming message.

What we show below, is the protected record to control the incoming messages and a typical agent

task which processes these messages, and provides the necessary context and thread to run a remotely-called subprogram.

```
type Local_Messages_Ids is natural range ...;
type Bool_array is array (natural <>) of boolean;

protected Incoming_Messages is
     -- A protected record to manage the
     -- incoming messages and the allocation
     -- of messages to agent tasks.

     entry Wait(Msg : out RPC_Stream_Access;
                ID : out Local_Messages_Ids);
        -- Wait for an incoming message (an RPC call).
        -- When one arrives, it will be pointed to by Msg,
        -- And ID will contain a partition-local
        -- index of this message.

     Entry Cancel (Local_Messages_Ids);
        -- The barrier for this entry will become true
        -- if a cancel request has arrived for this
        -- message.

     procedure Deallocate_ID (ID : Local_Messages_Ids);
        -- Return this ID to the free pool.

     procedure Cancel_Message (ID : Local_Messages_Ids);
        -- Will be called as a result of receiving a
        -- cancel message from the network and will
        -- mark the local id as cancelled.

     procedure Message_Arrived;
        -- This routine will be invoked by an interrupt.
        -- It will then read the message from the network
        -- device.
     pragma Attach_Interrupt_Handler (Message_Arrived,
                                      Network_Interrupt_Id);

record

   New_Messages : natural := 0;
   Free_Ids : Pool_Type;
   Cancelled : Bool_Array(Local_Messages_Ids) := (others => false;)
   -- Other fields to maintain the incoming
   -- messages queue

end Incoming_Messages;
```

```
task body Agent is
   Reply_Msg: (aliased) RPC_Message_Type;
     -- For now, allocate the maximum
   In_Msg : RPC_Stream_Access;
   Local_Msg_ID: natural;
     -- A reusable index of the message, local to the partition.
begin
   loop
      Incoming_Messages.Wait(In_Msg, Local_Msg_ID);

      select
         Incoming_Messages.Cancel(Local_Msg_ID);
         -- Do any necessary cleanup
      in
         begin
            This_Partition_Dispatcher(In_Msg.Target_pkg,
             In_Msg'ACCESS, Reply_Msg'ACCESS);
         exception
            when Exp: others =>
               System.Exception_Occurrence'WRITE
                 (Reply_Msg, Exp);
               Reply_Msg.Exception_Raised := true;
         end;
         if not In_Msg.Datagram then
            Send_Reply(In_Msg, Reply_Msg);
              -- The Partition_ID and Sender_ID to respond to,
              -- will be determined from In_Msg.
              -- Send_Reply will dispose of the two message buffers
              -- after sending the reply.
          end if;

      end select;
   end loop;
end Agent;
```

8.4.9 UPCS Interface Specification

The UPCS is a high-level interface intended to support the RPC requirements of partitions. We assume that this interface is above the message-passing layer of the network and uses it.

```
with System;
with Stream_Support;
package UPCS is

   Communication_Error : exception;

   type Message_Ids_Type is range ...;

   Fixed_Message_Length : Natural := ... impl-defined ...;

   type Dynamic_Buffer;
   type Dynamic_Buffer_Access is access Dynamic_Buffer;
```

```
-- The following type will probably be a private one,
-- exporting only the necessary access operations.
-- We show it here in its entirety for simplicity
-- only.

type RPC_Stream_Type (
 Length : Natural := Fixed_Message_Length) is
  new Stream_Support.Root_Stream_Type with
    record
       Message_Size : Natural := Length;
       Message_ID : Message_Ids_Type;
         -- Unique within Sending_Ptn
       Target_Ptn: Partition_ID_Type;
       Sending_Ptn : Partition_ID_Type := This_Partition;
       Target_Pkg : Package_ID_Type;
       Target_Subp : Subp_ID_Type;
       Sender_ID : System.Any_Task;
       Exception_Raised : Boolean := False;
       Datagram : boolean := false;
       case Length is
          when 0 =>
             Buffer_Ptr : Dynamic_Buffer_Access;
          when others =>
             Buffer : Stream_Support.Simple_Stream(Length);
    end record;

 type RPC_Stream_Access is access RPC_Stream_Type;

 procedure RPC_Stream_Type'WRITE (
     Stream : Stream_Support.Stream_Access;
     Obj : RPC_Stream_Type);

 function RPC_Stream_Type'READ (
     Stream : Stream_Support.Stream_Access)
     return RPC_Stream_Type;

 procedure DO_RPC (
   Out_Going :  RPC_Stream_Access;
   In_Coming : out RPC_Stream_Type;
   PTN_ID : natural;
   PKG_ID : natural;
   Subp_ID : natural
 );
   -- Send a message to perform an RPC on the specified
   -- partition, and wait for a reply or an exception.
   -- Out_Going will be deallocated, and if a reply
   -- exists, a message buffer will be allocated and
   -- pointed to by In_Coming upon return.

 procedure DO_Datagram (
   Out_Going :  RPC_Stream_Access;
   PTN_ID : natural;
   PKG_ID : natural;
   Subp_ID : natural
 );
   -- Sends the message and does not wait for reply.
   -- Can be used if all parameters of the RPC are
   -- IN, and no exception indication is important.
   -- There will probably a pragma to indicate the
   -- above properties.

private
```

```
      type Dynamic_Buffer is ... impl-defined ...;

   end UPCS;

   package UPCS.Extensions is

      -- This package will contain vendor-specific
      -- additions to the UPCS layer.

   end UPCS.Extensions;
```

8.4.9.1 Pseudo-Code of DO_RPC

The outgoing message is allocated and filled with the IN [OUT] parameters before calling *DO_RPC*, see 8.4.5.1.

```
Send the message to the appropriate partition
  using the network and the existing name-server.
Deallocate the message.

-- Assume the presence of a protected record
-- and an interrupt handler that will be
-- invoked when a reply arrives (just like
-- on the receiving node.)

Wait for reply on a protected record entry.
Allocate a buffer for the reply message.
If an exception indication then
   'READ exception string from message buffer
   Deallocate buffer
   raise exception
else
   Copy reply message to the buffer
   Return a pointer to the reply message buffer.
end if;

at end -- An abort handler for the DO_RPC
   when abort =>
      Send a cancel RPC message to the target
      partition, for the same PKG_ID and SUBP_ID.
end;
```

8.4.10 Elaboration and Termination of Partitions

8.4.10.1 Pre-Elaboration of Passive Partitions

All packages in a passive partition must be pure or preelaboratable, since there is no specific thread of control (in one of the other active partitions) to perform the elaboration. However, if link-time preelaboration is not supported, an implementation may designate a thread of control, in an active partition with an access to the passive one, to be responsible for the elaboration of that passive partition. This should happen though before the elaboration of any other active partition and in a way which is transparent to the program (as if elaboration occurred at link-time).

8.4.10.2 Elaboration of Active Partitions

See also G.3 in the MD 2.0. The relative order of elaborating active partitions is not specified by the language. Furthermore, it is not specified what happens to an RPC if its serving partition is not elaborated yet at the time the message arrives. Since the UPCS must maintain some information regarding the state of the active partitions, we recommend that such an "early" RPC will be delayed and kept for some amount of time before returning an error.

8.4.10.3 Termination of Active Partitions

As described in G.3 in the MD 2.0, active partitions terminate when their environment task can terminate.

8.4.11 Message Layout

In order to support the flattening of instances of a data types into messages, and reconstructing them, standard operations are defined. These operations will use a "stream" class of types. All types in the stream class must provide primitives for reading and/or writing an array of "Storage_Elements" (normally bytes). A stream can be associated with an external file, with an internal buffer, with a network channel, etc.

The compiler will automatically provide implementations of these operations for all non-limited types. The default implementations may be overridden. T'READ and T'WRITE will generally be implemented in terms of 'READ and 'WRITE operations of simpler component types, or in terms of the primitive stream operations defined in package Stream_Support.

```
package Stream_Support is

    type Storage_Element is range 0 .. 2**Storage_Unit-1;
    type Storage_Array is array(Positive range <>) of Storage_Element;

    -- For each different STREAM kind a new type will be derived
    -- from Root_Stream_Type to represent the particular structure and
    -- requirements of that STREAM.

    type Root_Stream_Type is tagged record null; end record;
    type Stream_Access is access Root_Stream_Type'CLASS;
    -- T'WRITE and T'READ of non-limited types for the default
    -- stream will use the following operations.  When called
    -- on different tags without a corresponding operations,
    -- PROGRAM_ERROR will be raised.  User provided T'WRITE and
    -- T'READ will used these operations as well when used in
    -- conjunction with the default Root_Stream_Type.
    -- Read array of storage elements from stream
    procedure Read(
      Stream : in out Root_Stream_Type;
      Item : out Storage_Array;
      Last : out Natural
      -- Last < Item'LAST only if end-of-stream reached
    ) is <>;
    -- Write array of storage elements into stream
    procedure Write(
      Stream : in out Root_Stream_Type;
      Item : in Storage_Array
    ) is <>;
```

```
        -- Align stream cursor on multiple of alignment
        procedure Align(
          Stream : in out Root_Stream_Type;
          Alignment : in Positive
        ) is <>;

    end Stream_Support;
    package Stream_Support.Simple_Streams is

        -- The following is the default Stream_Type which will be
        -- provided by each implementation.

        type Simple_Stream (Length : Positive) is new
          Root_Stream_Type with record
            Cursor : Natural := 0;
            Buffer : Storage_Array(1..Length);
        end record;

        -- Read array of storage elements from stream
        procedure Read(
          Stream : in out Simple_Stream;
          Item : out Storage_Array;
          Last : out Natural
          -- Last < Item'LAST only if end-of-stream reached
        );

        -- Write array of storage elements into stream
        procedure Write(
          Stream : in out Simple_Stream;
          Item : in Storage_Array
        );

        -- Align stream cursor on multiple of alignment
        procedure Align(
          Stream : in out Simple_Stream;
          Alignment : in Positive
        );

    end Stream_Support.Simple_Streams;
```

8.4.11.1 Predefined T'WRITE and T'READ

The predefined read/write operations for scalar or access type T transfer the number of storage elements which would be used by a one-element unpacked array of T. (This definition is meant to bypass concerns about register size, optimal size of stand-alone objects, etc.)

For a constrained array or record which is packed or has a rep clause, the predefined read/write operations transfer sufficient storage elements to represent the Item parameter, using the specified in-memory representation.

For a constrained array or record which is unpacked and has no rep clause, the predefined read/write operations consist of repeated reads/writes of the components in the canonical order (last dimension varying fastest for an array, order of component declaration for a record). Calls on the Align operation are inserted where necessary to maintain appropriate alignment.

For an unconstrained array, the 'LENGTH of each dimension is first written to the stream, followed by

the storage element sequence defined above for constrained arrays.

8.4.11.2 Validation on T'READ

The predefined T'READ operation for a record or an array may omit certain range checks on scalar subcomponents (other than discriminants), if the base type of the subcomponent does not have a default initialization. If the base type of the scalar subcomponent does have a default initialization, then full range checking must be performed as part of the read. This requirement preserves the desirable property that instances of scalar types with default type initialization are always within known bounds.

8.4.11.3 T'WRITE and T'READ for Tagged Universal Types

The predefined 'READ and 'WRITE operations for a tagged universal type T use a unique String ID in place of the type tag which appears in the in-memory representation. Each type derived from T has a distinct String ID, assigned at compile time so that its value does not vary from program to program which uses the type. The T'READ operation looks up the String ID in a table at run-time to find the appropriate type tag, or raises an exception if the String ID is not found (implying the type was not linked into the reading program).

8.4.11.4 Stream Representation of Exceptions

Exception tags use the same string ID mechanism as is used to represent a type tag.

8.4.12 Asynchronous Calls and Fault Tolerance

By default, the remote-call-interface subprograms will provide at-most-once semantics, with a standard exception (COMMUNICATION_ERROR) if the UPCS determines the call cannot be completed.

However, an additional pragma will be provided to specify that calls on a particular procedure should be treated as a datagram, with fire-and-forget semantics and no success or failure indication. This pragma would be appropriate for procedures with **in** parameters only.

The UPCS may provide additional interfaces to query the state of some remote partition (given its partition id), of the UPCS itself, to set timeout and retry parameters, etc.

Table of Contents