Part Three

The Annexes

The first two parts should have given the reader a good understanding of the Core of Ada 95. This third part describes the material in the Annexes. This includes the predefined environment which is mandatory, as well as the various specialized annexes themselves. It should be noted as a general principle that the annexes contain no new syntax. They are hence largely a description of various packages, attributes and pragmas.

A Predefined Language Environment

One of the main objectives of Ada 95 is to supply a set of supplemental packages of general utility in order to promote portability and reusability. Several packages are essentially intrinsic to the language (such as `Ada.Finalization`) and are discussed in Part Two. This chapter explains the main design decisions behind the packages described in Annex A of [RM95]. It should be noted that input-output which appeared in chapter 14 of the Ada 83 reference manual [ANSI 83, ISO 87] now appears in Annex A. This move is designed to emphasize that input-output is just one of many facilities provided by the predefined environment and is not really an intrinsic part of the language.

As mentioned in II.13, the predefined library is structured into three packages, `Ada`, `Interfaces` and `System` which can be thought of as child packages of `Standard`. The main reason for the restructuring is to avoid contamination of the top level name space and consequent risk of clashes with library units defined by the user.

The package `System` concerns intrinsic facilities associated with the target machine and with storage management and is discussed in Chapter 13. The package `Interfaces` concerns communication with systems in other languages and also the interface to hardware numeric types. All other predefined packages including input-output are children of `Ada`.

The major additions to the predefined environment compared with Ada 83 are as follows:

- The packages `Ada.Characters` and `Ada.Strings` provide general facilities for the manipulation of characters and strings.

- The package `Ada.Numerics` provides elementary functions and random number generation.

- There are new packages for completely heterogeneous input-output streams.

- The additional mode `Append_File` is provided for `Ada.Sequential_IO` and `Ada.Text_IO`.

- Improvements to `Text_IO` include facilities for looking ahead at the next character, for getting the next character (from the keyboard) without buffering or blocking, and to flush an output buffer. In addition the procedure `Get` now accepts a wider variety of numeric formats.

- The package `Ada.Wide_Text_IO` provides text input-output based on the types `Wide_Character` and `Wide_String`. Both `Text_IO` and `Wide_Text_IO` also have internal packages for modular and decimal input-output.

- The concept of a *current error file* is introduced in `Text_IO` by analogy with the current output file. Additional subprograms are added to manipulate current input, output and error in a convenient manner.

- The package `Ada.Command_Line` enables a program to access any arguments of the command which invoked it and to return an exit status.

In order to avoid incompatibility problems, renamings are provided for packages existing in Ada 83 such as

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

These renamings are considered obsolescent and thus liable to be removed at the next revision of the language.

A.1 Character Handling

Ada 95 provides an empty parent package `Ada.Characters` with two children: a package of character categorization and conversion subprograms, `Characters.Handling`, and a child package of constants, `Characters.Latin_1`, corresponding to the values of type `Character`. The intent is to provide basic character handling facilities, similar in scope to the contents of the standard C library header <ctype.h> [Plauger 92].

The following were the major issues concerning the design of this package:

- Which kinds of classification functions to supply;
- What to provide for `Wide_Character` handling;
- Whether to have an analogue to the package `Standard.ASCII`, extended to account for characters in the "upper half" of the character set;
- What to do about localization.

We had considered declaring the character handling subprograms directly in the package `Ada.Characters`. However, with such an approach there was some concern that an application needing access only to the constants in `Characters.Latin_1` would incur a code-space penalty if the subprograms in the parent package were bound into the application. Placing the subprograms in a child package addresses this concern.

A.1.1 Classification Functions

A preliminary design of the character handling package was based heavily on C's <ctype.h>. Although some of the classification functions are directly applicable, such as testing if a character is a digit, or testing if it is a letter, it was soon apparent that the C model was not completely appropriate for Ada 95. The main issue is that Ada 95, unlike C, has Latin-1 as its default standard character type. Thus the <ctype.h> categorization functions such as `ispunct(c)` and `isspace(c)` would have no standard meaning in Latin-1. Moreover, <ctype.h> relies on the C approach to locale, which is rather complicated and has not been adopted by Ada 95.

The categorization functions in `Characters.Handling` are designed to reflect more the properties of Latin-1 than the heritage of <ctype.h>. Most of the categorizations form a hierarchy:

- Each character is either a control or graphic character
- Each graphic character is either an alphanumeric or a special graphic
- Each alphanumeric is either a letter or a decimal digit
- Each letter is either an upper- or lower-case letter

Supplementing these are further categories; for example a basic character (one without diacritical marks), a hexadecimal digit, and an ISO_646 character (whose position is in the range 0..127).

A.1.2 Wide_Character Handling

There is a single classification function for `Wide_Character`, namely a test if a value is within the `Character` subset of the type. We had considered providing additional classification functions for `Wide_Character`, but this would be premature since there is no widespread agreement on how such functions might be defined.

The `Characters.Handling` package provides a conversion from `Wide_Character` to `Character`, and from `Wide_String` to `String`, that leaves a `Character` value unchanged and that replaces a value outside the `Character` range with a (programmer-specifiable) value inside this range.

A.1.3 Package of Character Names

The Ada 83 package `Standard.ASCII` declares a set of constants for the control characters (those whose positions are in the range `0 .. 31`, and also the character at position `127`), the lower-case letters, and some of the other graphic characters. The contents of this package are a rather uneven mixture, and different motivations led to the inclusion of different parts. The constants corresponding to the control characters are needed, since otherwise references to such characters would have to be in terms of `Character'Val(number)`, which is not in the spirit of the language. It is accepted practice to use `ASCII.Nul`, `ASCII.CR`, and `ASCII.LF` in Ada programs.

On the other hand, the inclusion of constants for the lower-case letters is principally a concession to the fact that, in the early 1980's, the input devices used in some environments did not support upper-case characters. To simulate a string literal such as `"Abc"` the programmer can write `"A" & ASCII.LC_B & ASCII.LC_C`.

For Ada 95, the issues were what to do about the package `Standard.ASCII`, and what to do about names of the characters in the "upper half" (those whose positions are in the range `128 .. 255`).

Part of the problem surrounding `Standard.ASCII` is due to the fact that the name "ASCII" now no longer refers to a 7-bit character set, but rather to ISO 8859-1 (Latin-1). Thus perhaps the most consistent approach would be to add to `Standard.ASCII` the declarations of names for "upper half" characters, and to introduce renamings where relevant in order to be consistent with ISO nomenclature (for example, `Reverse_Solidus` as a renaming of `Back_Slash`). However, this would have the significant disadvantage of introducing a large number of declarations into `Standard`. Even though they would be in an inner package (and thus not pollute the user's namespace), there was concern that such a specialized package would be out of place if included in package `Standard`.

These considerations led to the declaration of the child package `Characters.Latin_1`. This package includes names of the control characters from ISO 646 (the same names as in `Standard.ASCII`), the graphic characters from ISO 646 excepting the upper case letters and the decimal digits, the control characters from ISO 6429, and the graphic characters in the "upper half". The names of the graphics are based on Latin-1; hence `Number_Sign` for `'#'`, as opposed to `Sharp` as in `Standard.ASCII`. Since `Characters.Latin_1` is in the predefined environment, it must be supported by all implementations.

Although there is some overlap between `Characters.Latin_1` and `Standard.ASCII`, we expect that new Ada 95 programs will refer to the former, whereas existing Ada 83 code that is being moved intact to Ada 95 will continue to use the latter. In fact, the main reason to retain `Standard.ASCII` at all is for upward compatibility; removing it from the language would have been a major incompatibility and was not a realistic design alternative. Instead, it is specified as an obsolescent feature and its declaration appears in [RM95 J].

We recognize that names such as `Ada.Characters.Latin_1.Nul` are notationally rather heavy. However, we expect that users will typically provide renamings (either at the library level or as local declarations) such as

```
package Latin_1 renames Ada.Characters.Latin_1;
```

and thus in practice the references will be of the more palatable form `Latin_1.Nul`.

A.1.4 Character Set Localization

Although the language standard dictates Latin-1 as the contents of type `Character`, an implementation has permission to supply an alternative set specific to the locale or environment. For example, an eastern European implementation may define the type `Character` based on ISO 8859, Part 2 (Latin-2); a personal computer implementation may define the type `Character` as the native PC character set. Of course with such adaptations an Ada program might no longer be portable, but in some environments the ability to exploit the characteristics of the local environment is more important than the ability to move a program between different environments. In fact the explicit permission for a validated compiler to perform such localizations is not new in Ada 95 but applies also to Ada 83 based on a non-binding interpretation of ISO/IEC JTC1/SC22 WG9 Ada [ISO WG9 93].

An implication of such localization is that the semantics of the classification and conversion functions in the `Characters.Handling` package depends on the definition of `Character`. For example, the result of `Is_Letter(Character'Val(16#F7#))` is false for Latin-1 (this character is the division sign) but is true for Latin/Cyrillic.

A.2 String Handling

Many languages support string handling either directly via data types and operations or through standard supplemental library functions. Ada 83 provided the framework for a solution, through discriminated record types and access types, but the absence of a standard set of string handling services has proved a barrier to portability. To solve this problem, Ada 95 includes a set of packages for string handling that need to be supplied by all implementations.

A.2.1 Categories of Strings

We can divide string data structures into three categories based on their flexibility and associated storage management:

- A *fixed-length string* has a length whose value is established at object creation and is invariant during the object's lifetime. There is no need to use the heap for the storage of a fixed-length string.

- A *bounded-length string*, also known in the literature as a varying-length string, can change in length during its lifetime, but the maximum length is established no later than when the object is created. Thus a bounded-length string has both a current length which can change, and a maximum length which does not change.

Since the maximum length is known on a per-object basis, a natural implementation is to reserve this maximum amount of storage when the object is created, rather than using the heap.

- An *unbounded-length string*, also known in the literature as a dynamic string, can change in length during its lifetime with no a priori maximum length other than that implied by the range of the index subtype. Unbounded-length strings need to be managed dynamically, either in the general heap or in some region reserved for strings, because of the wide range of possible sizes for any given object.

In practice the storage allocation performed by the compiler may vary from the "natural" method mentioned. For example, if the length of a fixed-length string, or the maximum length of a bounded-length string, exceeds some threshold value then the compiler may choose to place the object on the heap, with automatic reclamation when the object becomes inaccessible. This may be done because of target machine addressing constraints or (for bounded-length strings with a prohibitively large maximum size) as a means to economize on storage usage.

Ada 95 supplies packages for each of these categories, for both the predefined types `String` and `Wide_String`. For fixed-length strings the type is the specific type `String` or `Wide_String`. For the other two categories, a private type is supplied (see below) since it is important for purposes of data abstraction to avoid exposing the representation. Each of the three packages supplies a set of string-handling subprograms. The bounded- and unbounded-length string packages also supply conversion and selection functions; these are needed because the type is private.

A.2.2 Operations on Strings

Operations on strings fall into several categories. This section summarizes the various operations that are provided, and notes the semantic issues that arise based on whether the strings are fixed-, bounded-, or unbounded-length.

Constructors

Literals are available for fixed-length strings; for bounded- and unbounded-length strings we need conversion functions (`String` to/from `Bounded_String` and also to/from `Unbounded_String`).

The conversion function from `String` to `Bounded_String` illustrates a point that comes up in other contexts when constructing a bounded string: suppose the length of the result exceeds the maximum length of the bounded string? We let the user control the behavior through a parameter to the constructor function. The default effect is to raise an exception (`Strings.Length_Error`), but the user can also establish truncation of extra characters either on the right or left.

Concatenation is available automatically for fixed-length strings, and explicit overloadings are provided for bounded and unbounded strings. Note that since the operator form for concatenation of bounded length strings does not offer a possibility for the user to control the behavior if the result length exceeds the bounded string type's maximum length, we provide also a set of `Append` functions taking an explicit parameter dictating truncation versus raising an exception. The operator form will raise an exception if the result length exceeds the type's maximum length.

For bounded and unbounded strings, there is the question of how many overloaded versions to supply for the concatenation functions. For convenience we allow concatenation of a `Bounded_String` with either a `Character`, a `String`, or another `Bounded_String`, returning a `Bounded_String` result, and analogously for `Unbounded_String`. We decided against allowing the concatenation of two fixed-length strings return a `Bounded_String` (or an `Unbounded_String`), since such an overloading would render ambiguous a statement such as

```
B := S1 & S2 & S3;
```

where `S1`, `S2` and `S3` are of type `String` and `B` is of type `Bounded_String`.

If it is necessary to convert between a bounded and an unbounded string, this can be done by producing a `String` as an intermediate result.

Replication operations are also provided to construct string values. For each of the three string categories a `"*"` operator is supplied with left operand of subtype `Natural` and right operand either a `Character`, a `String`, or (for bounded and unbounded strings) a value of the corresponding string type. The result is of the string type. For example:

```
declare
   Alpha, Beta : Unbounded_String;
begin
   Alpha := 3 * 'A';      -- To_String(Alpha) = "AAA"
   Alpha := 2 * Alpha;    -- To_String(Alpha) = "AAAAAA"
   Beta  := 2 * "Abc";    -- To_String(Beta)  = "AbcAbc"
end;
```

Copying

The issue in copying is what to do when the source and target lengths differ; this is only a concern in the fixed-length case. For bounded strings the `":="` operation always works: the source and target have identical maximum lengths, so assignment simply copies the source to the target. For unbounded strings the `":="` operation does the necessary storage management through `Adjust` and `Finalize` operations to allocate needed space for the new value of the target and to reclaim the space previously occupied by the object.

Our model, based on COBOL, is that a fixed-length string comprises significant contents together with padding. The pad characters may appear either on the right or left (or both); this is useful for report output fields. Parameters to the `Move` procedure allow the programmer to control the effect. When a shorter string is copied to a longer string, pad characters are supplied as filler, and the `Justify` parameter guides where the source characters are placed. When a longer string is copied to a shorter string, the programmer establishes whether the extra characters are to be dropped from the left or the right, or if an exception should be raised when a non-pad character is dropped.

Selection

Component selection is not an issue for fixed-length strings, since indexing and slicing are directly available. For both bounded and unbounded strings, we supply subprograms to select and replace a single element, and to select and replace a slice.

Ordering relations

For fixed-length strings the predefined ordering and equality operators are appropriate, but for both the bounded and unbounded string types we provide explicit overloadings. Note that if the implementation chooses to represent bounded strings with a maximum-length array and an index for the current length (see A.2.5 for further discussion), then predefined assignment has the desired effect but predefined equality does not, since it would check the "junk" characters in the string beyond the logical length.

The ordering operators return a result based on the values of corresponding characters; Thus for example the string "ZZZ" is less than the string "aa". Anything more sophisticated would have been out of scope and in any event is dependent on local cultural conventions.

Searching and pattern matching

Each of the string handling packages provides subprograms to scan a string for a pattern (`Index`, `Count`) or for characters inside or outside specified sets (`Index_Non_Blank`, `Index`, and `Find_Token`). The profiles for each of these subprograms is the same in the three packages, except for the type of the source string (`String`, `Bounded_String`, or `Unbounded_String`).

A design issue was how to arrange that pattern matches be case insensitive, or in general to reflect user-defined character equivalences. Our approach is to supply to each pattern matching function a parameter that specifies a character equivalence. By default the equivalence mapping is the identity relation, but the programmer can override this via an explicit parameter of type `Strings.Maps.Character_Mapping`.

Although `Index_Non_Blank` is redundant, it is included since searching for blanks is such a common operation.

`Find_Token` is at a somewhat higher level than the other subprograms. We have supplied this procedure since it is extremely useful for simple lexical analysis such as parsing a line of interactively supplied input text.

String transformation

As with the searching and pattern matching subprograms, we supply the same functionality for string transformations in each of the three string handling packages.

A common need is to translate a string via a character translation table. The `Translate` function satisfies this goal. The procedural form of `Translate` is included for efficiency, to avoid the extra copying that may be required for function returns.

The other string transformation subprograms are `Replace_Slice`, `Insert`, `Overwrite`, `Delete`, and `Trim`. These are not necessarily length preserving.

We had considered including a subprogram to replace all occurrences of a pattern with a given string but ultimately decided in the interest of simplicity to leave this out. It can be written in terms of the supplied operations if needed (see the example in A.2.8).

A.2.3 General Design Decisions

Independent of the functionality provided, several fundamental design questions arose: whether to make the packages generic (with respect to character and string type) or specific to the types in `Standard`; how to organize the packages (hierarchically or as siblings); and whether to define the string-returning operations as functions, procedures, or both.

Generic vs non-generic form of packages

String handling needs to be provided for the predefined `String` and `Wide_String` types, and it is also useful for strings of elements from user-supplied character types. For these reasons it seems desirable to have a generic version of the string handling packages, with language-defined instantiations for `Character` and `String` and also for `Wide_Character` and `Wide_String`. In fact, an earlier version of the packages adopted this approach, but we subsequently decided to provide non-generic forms instead.

There are several reasons for this decision. First, although the specifications for the packages for handling `Character` and `Wide_Character` strings might be the same, the implementations would be different. Second, the generic form would be rather complicated, a pedagogical issue for users and a practical issue for implementations.

Structure of packages

In order to minimize the number of language-defined names for immediate children of the root package `Ada`, the string handling packages form a hierarchy. The ancestor unit, `Ada.Strings`, declares the types and exceptions common to the other packages. The package `Strings.Maps` declares the types and related entities for the various data representations needed by the other packages. `Strings.Fixed`, `Strings.Bounded`, and `Strings.Unbounded` provide the entities for fixed-length, bounded-length, and unbounded-length strings, respectively. The package `Strings.Maps.Constants` declares `Character_Set` constants corresponding to the character classification functions in the package `Characters`, as well as `Character_Mapping` constants that can be used in pattern matching and string transformations. There are analogous packages `Strings.Wide_Maps`, `Strings.Wide_Fixed`, `Strings.Wide_Bounded`, `Strings.Wide_Unbounded`, and `Strings.Wide_Maps.Constants`, for `Wide_String` handling.

Procedures vs functions

The subprograms that deliver string results can be defined either as functions or as procedures. The functional notation is perhaps more pleasant stylistically but typically involves extra copying. The procedural form, with an in out parameter that is updated "in place", is generally more efficient but can lead to a heavy-looking style.

Our solution is to provide both forms for all three string-handling packages. Although this increases the size of the packages, the benefits are an increase in flexibility for the programmer, and a regularity in the structure of the packages that should make them easier to use.

A.2.4 Strings.Maps

The package `Strings.Maps` defines the types for representing sets of characters and character-to-character mappings, for the types `Character` and `String`. A corresponding package, `Strings.Wide_Maps`, provides the same functionality for the types `Wide_Character` and `Wide_String`.

The type `Character_Set` represents sets of `Character` values that are to be passed to the string handling subprograms. We considered several alternative declarations for `Character_Set`:

- A visible constrained array-of-Booleans type;
- A visible unconstrained array-of-Booleans type;
- A private type; and
- A private type with unknown discriminants.

A visible constrained array type is the traditional representation of a set of values from a discrete type; in the case of `Character` it would be:

```
type Character_Set is array (Character) of Boolean;
pragma Pack(Character_Set);
```

However, this has several disadvantages. First, it would differ from the choice of representations for a set of `Wide_Character` values, in the package `Strings.Wide_Maps`; in the latter package a constrained array type is not a realistic decision, since an overhead of 2^{16} (64K) bits for each set would be excessive. Second, even 256 bits may be more than is desirable for small sets, and a more compact representation might be useful.

An unconstrained array of Booleans addresses the second issue:

```
type Character_Set is array (Character range <>) of Boolean;
pragma Pack(Character_Set);
```

In this version, an object `CS` of type `Character_Set` represents the set comprising each character `C` in `CS'Range` such that `CS(C)` is true; any character outside `CS'Range` is implicitly regarded as not being in the set, and of course any character `C` in `CS'Range` such that `CS(C)` is false is regarded as not being in the set. Thus, for example, the empty set is represented by a null `Character_Set` array (as well as by many other `Character_Set` values).

The unconstrained array approach was used in earlier versions of the string handling packages, since it is more efficient in storage than the constrained array approach. However, we ultimately decided against this approach, for several reasons.

- Similar to the constrained array-of-Booleans approach, it is not always appropriate for `Wide_Character` sets. In particular, even if a set is small and has a compact representation, taking the complement of the set can yield a value requiring 64K bits.

- The effect of `":="` is unintuitive. Two `Character_Set` objects could represent the same set, yet since they might have different lengths, assigning one to another could raise `Constraint_Error`.

- The need to provide an explicit initialization for each `Character_Set` variable (since the type is unconstrained) is inconvenient.

The private type approach is much more in the spirit of Ada and allows the implementation, rather than requiring the language, to make the choice of representations for `Character_Set`. Note that a simple private type (i.e., one without unknown discriminants) is not allowed to have an unconstrained type as its full declaration. Thus if we want to allow some flexibility (rather than just imposing a private type interface on what is certain to be a constrained array type declaration as the full type declaration) we should allow the possibility of having the full declaration be an access type whose designated type is an unconstrained array of Booleans. To do this, we need to compromise the goal of having a pure package (since access types are not permitted in a pure package); instead, we simply make the package preelaborable.

A private type with an unknown discriminant part might seem like a more direct way to allow the unconstrained-array-of-Booleans as the full declaration, but it suffers from a major portability flaw. If `Set_1` and `Set_2` are objects of type `Character_Set`, and `Character_Set` is a private type with an unknown discriminant part, then the assignment `Set_1 := Set_2;` may or may not raise `Constraint_Error`, depending on what the implementation chooses for the full type declaration.

As a result of these considerations, we have declared `Character_Set` as a private type, without an unknown discriminant part, and have specified the package as just preelaborable rather than pure in order to allow the implementation to use an access type in the full declaration of `Character_Set`.

A consequence of declaring `Character_Set` as private is that constructor functions are needed for composing `Character_Set` values. We have provided several such functions, each named `To_Set`. Since it is often convenient to have a set containing a single character, or exactly those characters appearing in some array, we have overloaded To_Set to take either a parameter of type Character or of type Character_Sequence (the latter is in fact just a subtype with the effect of renaming String). It is also useful to compose a set out of one or more character ranges, and hence we have supplied the appropriate additional overloadings of To_Set. In the other direction, it is useful to get a "concrete" representation of a set as either a set of ranges or a character sequence, and hence we have provided the corresponding functions.

Although introducing the name Character_Sequence is not strictly necessary (the name String would be equivalent), the style of having a subtype as effectively a renaming of an existing (sub)type makes the intent explicit.

Other languages that supply string handling functions represent character sets directly as character sequences as opposed to boolean arrays; for example, the functions in the C standard header <strings.h>. This was considered for the Ada 95 packages but rejected in the interest of efficiency.

Another type declared by Strings.Maps is Character_Mapping, which represents a mapping from one Character value to another. For the same reasons underlying the choice of a private type for Character_Set, we have also declared Character_Mapping as private. A typical choice for a full type declaration would be:

```
type Character_Mapping is array (Character) of Character;
```

with the obvious interpretation; if CM is a Character_Mapping and C is a character, then CM(C) is the character to which C maps under the mapping CM.

Character mappings are used in two contexts:

- To define an equivalence function applicable during pattern matches (e.g., allowing the programmer to do searches where the distinction between upper and lower case letters does not matter); and

- To define a translation table used in string transformation subprograms.

As an example of the use of the Character_Mapping type, the constant Lower_Case_Map (declared in Strings.Maps.Constants) maps each letter to the corresponding lower case letter and maps each other character to itself. The following finds the first occurrence of the pattern string "gerbil" in a source string S, independent of case:

```
Index(Source  => S,
      Pattern => "gerbil",
      Going   => Forward,
      Mapping => Strings.Maps.Constants.Lower_Case_Map)
```

A character C matches a pattern character P with respect to the Character_Mapping value Map if Map(C)=P. Thus the user needs to ensure that a pattern string comprises only characters occurring in the range of the mapping. (Passing as a pattern the string "GERBIL" would always fail for the mapping Lower_Case_Map.) An earlier version of the string packages had a more symmetric definition for matching; namely C matched P if Map(C) = Map(P). However, this yielded some counterintuitive effects and has thus been changed.

There is another possible representation for mappings, namely an access value denoting a function whose domain and range are the character type in question. This would be useful where the domain and range are very large sets, and in fact is used in the string handling packages for Wide_Character and Wide_String. To avoid unnecessary differences between the String and Wide_String packages, we have supplied the analogous access-to-subprogram type in Strings.Maps:

```
type Character_Mapping_Function is
   access function (From : in Character) return Character;
```

Each subprogram that takes a `Character_Mapping` parameter is overloaded with a version that takes a `Character_Mapping_Function`. In an earlier version of the string handling packages, the access-to-subprogram type was provided for `Wide_String` handling but not for `String` handling, since we were striving to make the latter pure. However, since the package has had to compromise purity for other reasons as described above, there was no longer a compelling reason to leave out the character mapping function type.

A.2.5 Bounded-Length Strings

The major decisions for bounded-length strings were (1) whether the type should be private or not, and (2) whether to realize the maximum length as a discriminant or, instead, as a generic formal parameter.

There are two main reasons to declare a type as private as opposed to non-private:

- To hide irrelevant representational decisions, thus allowing implementation flexibility,
- To ensure that the programmer does not violate data consistency or otherwise abuse the intent of the type.

Both of these apply to `Bounded_String`; hence it is appropriate for the type to be declared as private.

There are two principal ways to represent a varying- (but bounded-) length string, assuming that access types are to be avoided. One is to supply the maximum length as a discriminant constraint, thus allowing different objects of the same type to have different maximum lengths. The other approach is to supply the maximum length at the instantiation of a generic package declaring a bounded string type, implying that objects with different maximum lengths must be of different types. We thus have the following basic approaches:

```
package Discriminated_Bounded_Length is
   type Bounded_String(Max_Length : Positive) is private;
   function Length(Item : Bounded_String) return Natural;
   ...
private
   type Bounded_String(Max_Length : Positive) is
      record
         Length : Natural;
         Data   : String(1 .. Max_Length);
      end record;
end Discriminated_Bounded_Length;
```

and also the alternative:

```
generic
   Max : Positive:
package Generic_Bounded_Length is
   Max_Length : constant Positive := Max;
   subtype Length_Range is Natural range 0 .. Max_Length;

       type Bounded_String is private;

       function Length(Item : Bounded_String)
          return Length_Range;
       ...
```

```
private
   type Bounded_String_Internals(Length : Length_Range := 0) is
      record
         Data : String(1 .. Length);
      end record;
   type Bounded_String is
      record
         Data : Bounded_String_Internals;
      end record;
end Generic_Bounded_Length;
```

Each of these approaches has advantages and disadvantages (the reason for the seeming redundancy in the private part of the generic package will be discussed below). If there is an operation that needs to deal with `Bounded_String` values with different maximum lengths, then the discriminated type approach is simpler. On the other hand, predefined assignment and equality for discriminated `Bounded_String` do not have the desired behavior. Assignment makes sense when the maximum lengths of source and target are different, as long as the source's current length is no greater than the target's maximum length, yet predefined `":="` would raise `Constraint_Error` on the discriminant mismatch. User-defined `Adjust` and `Finalize` operations do not solve this problem. It would be possible to avoid the difficulty by declaring the type as limited private, but this would result in a very clumsy programming style.

A variation is to declare a discriminated type with a default value for the `Max_Length` discriminant. An object declared unconstrained can thus be assigned a value with a different maximum length (and a different length). This approach, however, introduces other problems. First, if the object is allocated rather than declared, then its discriminant is in fact constrained (by its default initial value). Second, declaring an appropriate subtype for the discriminant — that is, establishing an appropriate bound for `Max_Length` — is difficult. If it is too small then the user might not be able to create needed objects. If it is too large, then there will either be a lot of wasted space or else the implementation may use dynamic storage allocation implicitly.

The solution is to let the user establish the maximum length as a parameter at generic instantiation. Such an approach avoids these complications, but has two main drawbacks. First, the programmer will need to perform as many instantiations as there are different maximum lengths to be supported. Second, operations involving varying-length strings of different maximum lengths cannot be defined as part of the same generic package. However, the programmer can get around the first difficulty by providing a small number of instantiations with sufficient maximum size (for example, max lengths of 20 and 80). Either explicit overloadings or generics with formal package instantiations serve to address the second issue. For these reasons we have adopted the generic approach, rather than the discriminant approach, to specifying the maximum length for a varying-length string.

Note that `Bounded_String` in the generic package is declared without discriminants. `Max_Length` is established at the generic instantiation, and the `Length` field is invisible to the user and is set implicitly as part of the effect of the various operations. An alternative would be to declare the type as follows:

```
type Bounded_String(Length : Length_Range := 0) is private;
```

However, this would allow the user to create constrained instances, which defeats the intent of the package. In order to prevent such abuses it is best to leave the `Length` component hidden from the user [Eachus 92].

A final point of rationale for the `Bounded_String` generic: the reason for declaring `Max_Length`, which is simply a constant reflecting the value supplied at the generic instantiation, is to allow the user to refer to the maximum length without keeping track manually of which values were supplied at which instantiations.

A.2.6 Unbounded-Length Strings

Unbounded-length strings need to be implemented via dynamic storage management. In Ada 83, in the absence of automatic garbage collection it was the programmer's responsibility to reclaim storage through unchecked deallocation. Ada 95's facilities for automatically invoked `Adjust` and `Finalize` plug this loophole, since the unbounded string type implementor can arrange that storage be reclaimed implicitly, with no need for the user to perform unchecked deallocation.

The main design issue for unbounded strings was whether to expose the type as derived from `Finalization.Controlled`. That is, the type could be declared either as

```
type Unbounded_String is private;
```

or

```
type Unbounded_String is new Finalization.Controlled with private;
```

An advantage of the latter approach is that users can further derive from `Unbounded_String` for richer kinds of data structures, and override the default `Finalize` and `Adjust`. However, we have chosen the simpler approach, just making `Unbounded_String` private. If a more complicated data structure is desired, this can be obtained by including an `Unbounded_String` as a component.

Besides providing the private type `Unbounded_String`, the package `Strings.Unbounded` declares a visible general access type `String_Access` whose designated type is `String`. The need for such a type arises often in practice, and so it is appropriate to have it declared in a language-defined package.

The following is a sample implementation of the private part of the package:

```
private
   use Finalization;

   Null_String : aliased String := "";

   type Unbounded_String is new Controlled with
      record
         Reference : String_Access := Null_String'Access;
      end record;

   -- No need for Initialize procedure
   procedure Finalize (Object : in out Unbounded_String);
   procedure Adjust   (Object : in out Unbounded_String);

   Null_Unbounded_String : constant Unbounded_String :=
     (Controlled with Reference => Null_String'Access);

end Ada.Strings.Unbounded;
```

The following skeletal package body illustrates how several of the subprograms might be implemented.

```
with Unchecked_Deallocation;
package body Strings.Unbounded is
   procedure Free is
      new Unchecked_Deallocation(String, String_Access);
```

```
function To_Unbounded_String(Source : String)
   return Unbounded_String is
   Result_Ref : constant String_Access :=
                        new String(1 .. Source'Length);
begin
   Result_Ref.all := Source;
   return (Finalization.Controlled with Reference => Result_Ref);
end To_Unbounded_String;

function To_String(Source : Unbounded_String) return String is
begin
   return Item.Reference.all;
     -- Note: Item.Reference is never null
end To_String;

--  In the following subprograms, the Reference component of each
--  Unbounded_String formal parameter is non-null, because of the
--  default initialization implied by the type's declaration

function Length(Source : Unbounded_String) return Natural is
begin
   return Source.Reference.all'Length;
end Length;

function "=" (Left, Right : Unbounded_String) return Boolean is
begin
   return Left.Reference.all = Right.Reference.all;
end "=";

procedure Finalize(Object : in out Unbounded_String) is
begin
   if Object.Reference /= Null_String'Access then
      Free(Object.Reference);
   end if;
end Finalize;

procedure Adjust(Object : in out Unbounded_String);
begin
   --  Copy Object if it is not Null_Unbounded_String
   if Object.Reference /= Null_String'Access then
      Object.Reference := new String'(Object.Reference.all);
   end if;
end Adjust;

function "&" (Left, Right : in Unbounded_String)
              return Unbounded_String is
   Left_Length   : constant Natural := Left.Reference.all'Length;
   Right_Length  : constant Natural := Right.Reference.all'Length;
   Result_Length : constant Natural := Left_Length + Right_Length;
   Result_Ref    : String_Access;
begin
   if Result_Length = 0 then
      return Null_Unbounded_String;
   else
      Result_Ref := new String(1 .. Result_Length);
      Result_Ref.all(1..Left_Length) := Left.Reference.all;
      Result_Ref.all(Left_Length+1..Result_Length) :=
         Right.Reference.all;
```

```
            return (Finalization.Controlled with
                       Reference => Result_Ref);
         end if;
      end "&";
      ...
   end Ada.Strings.Unbounded;
```

A.2.7 Wide String Handling

Since the same functionality is needed for `Wide_String` as for `String`, there are child packages of `Ada.Strings` with analogous contents to those discussed above, but for `Wide_Character` and `Wide_String`. The only difference is that some of the type and subprogram names have been adapted to reflect their application to `Wide_Character`.

As a consequence of providing equivalent functionality for the two cases, we have made it easier for a programmer to modify an application that deals with, say, `String` data, so that it can work with `Wide_String` data.

A.2.8 Examples

The function below, which replaces all occurrences of a pattern in a source string, is intended as an illustration of the various string handling operations rather than as a recommended style for solving the problem. A more efficient approach would be to defer creating the result string until after the pattern matches have been performed, thereby avoiding the overhead of allocating and deallocating the intermediate string data at each iteration.

```
with Ada.Strings.Maps, Ada.Strings.Unbounded, Ada.Strings.Fixed;
use Ada.Strings;
function Replace_All (Source  : in String;
                      Pattern : in String;
                      By      : in String;
                      Going   : in Direction := Forward;
                      Mapping : in Maps.Character_Mapping :=
                                     Maps.Identity)
                      return String is
   use type Unbounded.Unbounded_String;
   Pattern_Length : constant Natural := Pattern'Length;
   Start          : Natural          := Source'First;
   Result         : Unbounded.Unbounded_String;
   Index          : Natural;
begin
   loop
      Index :=
         Fixed.Index(Source(Start .. Source'Last),
                     Pattern, Going, Mapping);
      if Index/=0 then
         Result := Result & Source(Start .. Index-1) & By;
         Start  := Index + Pattern'Length;
      else
         Result := Result & Source(Start .. Source'Last);
         return Unbounded.To_String(Result);
      end if;
   end loop;
end Replace_All;
```

The following program fragments show how the string handling subprograms may be used to get the effect of several `COBOL INSPECT` statement forms.

```
COBOL:   INSPECT ALPHA
         TALLYING NUM FOR ALL "Z" BEFORE "A".


Ada 95:  Alpha : String( ... );
         A_Index, Num: Natural;
         ...
         A_Index := Index(Alpha, 'A');
         Num     := Count(Alpha(Alpha'First .. A_Index-1), "Z");


COBOL:   INSPECT ALPHA
         REPLACING ALL "A" BY "G", "B" BY "H"
         BEFORE INITIAL "X".


Ada 95:  Alpha   : String( ... );
         X_Index : Natural;
         My_Map  : Character_Mapping :=
                       To_Mapping(From => "AB", To=>"GH");
         ...
         X_Index := Index(Alpha, 'X');
         Translate(Source  => Alpha(Alpha'First .. X_Index -1),
                   Mapping => My_Map);
```

A.3 Numerics Packages and Attributes

Ada 95 includes in the predefined environment several child packages of `Ada.Numerics`, and the language also provides a comprehensive set of representation-oriented, model-oriented, and primitive-function attributes for real types.

The package `Ada.Numerics` itself defines the named numbers `Pi` and `e`, as well as an exception (`Argument_Error`) shared by several of its children.

The constants `Pi` and `e` are defined for the convenience of mathematical applications. The WG9 Numerics Rapporteur Group did not define these constants in the secondary numeric standards for Ada 83 [ISO 94a], primarily because it could not decide whether to define a minimal set (as has now been done in Ada 95) or a much larger set of mathematical and physical constants. Ada 95 implementations are required to provide `Pi` and `e` to at least 50 decimal places; this exceeds by a comfortable margin the highest precision available on present-day computers.

The `Argument_Error` exception is raised when a function in a child of `Numerics` is given an actual parameter whose value is outside the domain of the corresponding mathematical function.

The child packages of `Ada.Numerics` are `Generic_Elementary_Functions` and its non-generic equivalents, `Float_Random` and `Discrete_Random` (see A.3.2); `Generic_Complex_Types` and its non-generic equivalents (see G.1.1); `Generic_Complex_Elementary_Functions` and its non-generic equivalents (see G.1.2).

A.3.1 Elementary Functions

The elementary functions are critical to a wide variety of scientific and engineering applications written in Ada. They have been widely provided in the past as vendor extensions, but the lack of a standardized interface, variations in the use or avoidance of generics, differences in the set of functions provided, and absence of guaranteed accuracy have hindered the portability and the analysis of programs. These impediments are removed by including the elementary functions in the predefined language environment.

The elementary functions are provided in Ada 95 by a generic package, `Numerics.Generic_Elementary_Functions`, which is a very slight variation of the generic package, `Generic_Elementary_Functions`, defined in [ISO 94a] for Ada 83.

In addition, Ada 95 provides non-generic equivalent packages for each of the predefined floating point types, so as to facilitate the writing of scientific applications by programmers whose experience in other languages leads them to select the precision they desire by choosing an appropriate predefined floating point type. The non-generic equivalent packages have names as follows

```
Numerics.Elementary_Functions         -- for Float
Numerics.Long_Elementary_Functions    -- for Long_Float
```

and so on.

These nongeneric equivalents behave just like instances of the generic packages except that they may not be used as actual package parameters as in the example in 12.6.

A vendor may, in fact, provide the non-generic equivalent packages by instantiating the generic, but more likely they will be obtained by hand-tailoring and optimizing the text of the generic package for each of the predefined floating point types, resulting in better performance.

The `Argument_Error` exception is raised, for example, when the `Sqrt` function in `Numerics.Generic_Elementary_Functions` is given a negative actual parameter. In [ISO 94a] and related draft secondary standards for Ada 83, `Argument_Error` was declared in each generic package as a renaming of an exception of the same name defined in a (non-generic) package called `Elementary_Functions_Exceptions`; in Ada 95, the children of `Numerics` do not declare `Argument_Error`, even as a renaming. In Ada 83, simple applications that declare problem-dependent floating point types might look like this:

```
with Generic_Elementary_Functions;
procedure Application is
   type My_Type is digits ...;
   package My_Elementary_Functions is
      new Generic_Elementary_Functions(My_Type);
   use My_Elementary_Functions;
   X : My_Type;
begin
   ... Sqrt(X) ...
exception
   when Argument_Error =>
   ...
end Application;
```

In Ada 95, they will look almost the same, the essential difference being the addition of context clauses for `Ada.Numerics`:

```
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Elementary_Functions;
procedure Application is
   type My_Type is digits ...;
   package My_Elementary_Functions is
      new Generic_Elementary_Functions(My_Type);
   use My_Elementary_Functions;
   X : My_Type;
begin
   ... Sqrt(X) ...
```

```
exception
   when Argument_Error =>
   ...
end Application;
```

The benefit of the Ada 95 approach can be appreciated when one contemplates what happens when a second problem-dependent type and a second instantiation of `Numerics.Generic_Elementary_Functions` are added to the application. There are no surprises in Ada 95, where one would write the following:

```
with Ada.Numerics; use Ada.Numerics;
with Ada.Numerics.Generic_Elementary_Functions;
procedure Application is
   type My_Type_1 is digits ...;
   type My_Type_2 is digits ...;
   package My_Elementary_Functions_1 is
      new Generic_Elementary_Functions(My_Type_1);
   package My_Elementary_Functions_2 is
      new Generic_Elementary_Functions(My_Type_2);
   use My_Elementary_Functions_1, My_Elementary_Functions_2;
   X : My_Type_1;
   Y : My_Type_2;
begin
   ... Sqrt(X) ...
   ... Sqrt(Y) ...
exception
   when Argument_Error =>
   ...
end Application;
```

If one were to extend the Ada 83 example with a second problem-dependent type and a second instantiation, one would be surprised to discover that direct visibility of `Argument_Error` is lost (because both instances declare that name, and the declarations are not overloadable [RM95 8.4(11)]). To regain direct visibility, one would have to add to the application a renaming declaration for `Argument_Error`.

The functions provided in `Numerics.Generic_Elementary_Functions` are the standard square root function (`Sqrt`), the exponential function (`Exp`), the logarithm function (`Log`), the forward trigonometric functions (`Sin`, `Cos`, `Tan`, and `Arctan`), the inverse trigonometric functions (`Arcsin`, `Arccos`, `Arctan`, and `Arccot`), the forward hyperbolic functions (`Sinh`, `Cosh`, `Tanh`, and `Coth`), and the inverse hyperbolic functions (`Arcsinh`, `Arccosh`, `Arctanh`, and `Arccoth`). In addition, an overloading of the exponentiation operator is provided for a pair of floating point operands.

Two overloadings of the `Log` function are provided. Without a `Base` parameter, this function computes the natural (or Napierian) logarithm, i.e. the logarithm to the base `e`, which is the inverse of the exponential function. By specifying the `Base` parameter, which is the second parameter, one can compute logarithms to an arbitrary base. For example,

```
Log(U)                    -- natural logarithm of U
Log(U, 10.0)              -- common (base 10) logarithm of U
Log(U, 2.0)               -- log of U to the base 2
```

Two overloadings of each of the trigonometric functions are also provided. Without a `Cycle` parameter, the functions all imply a natural cycle of 2p, which means that angles are measured in radians. By specifying the `Cycle` parameter, one can measure angles in other units. For example,

```
Sin(U)                        -- sine of U (U measured in radians)
Cos(U, 360.0)                 -- cosine of U (U measured in degrees)
Arctan(U, Cycle => 6400.0) -- angle (in mils) whose tangent is U
Arccot(U, Cycle => 400.0)  -- angle (in grads) whose cotangent is U
```

`Cycle` is the second parameter of all the trigonometric functions except `Arctan` and `Arccot`, for which it is the third. The first two parameters of `Arctan` are named `Y` and `X`, respectively; for `Arccot`, they are named `X` and `Y`. The first parameter of each of the remaining trigonometric functions is named `X`. A ratio whose arctangent or arccotangent is to be found is specified by giving its numerator and denominator separately, except that the denominator can be omitted, in which case it defaults to `1.0`. The separate specification of numerator and denominator, which of course is motivated by the Fortran ATAN2 function, allows infinite ratios (i.e., those having a denominator of zero) to be expressed; these, of course, have a perfectly well-defined and finite arctangent or arccotangent, which lies on one of the axes. Thus,

```
Arctan(U, V)              -- angle (in radians) whose tangent is U/V
Arccot(U, V)              -- angle (in radians) whose cotangent is U/V
Arctan(U)                 -- angle (in radians) whose tangent is U
Arctan(U, V, 360.0)       -- angle (in degrees) whose tangent is U/V)
Arctan(1.0, 0.0, 360.0) -- 90.0 (degrees)
```

The result of `Arctan` or `Arccot` is always in the quadrant (or on the axis) containing the point `(X, Y)`, even when the defaultable formal parameter takes its default value; that of `Arcsin` is always in the quadrant (or on the axis) containing the point `(1.0, X)`, while that of `Arccos` is always in the quadrant (or on the axis) containing the point `(X, 1.0)`.

Given that the constant `Pi` is defined in `Numerics`, one might wonder why the two overloadings of each trigonometric function have not been combined into a single version, with a `Cycle` parameter having a default value of `2.0*Numerics.Pi`. The reason is that computing the functions with natural cycle by using the value of `Numerics.Pi` cannot provide the accuracy required of implementations conforming to the Numerics Annex, as discussed below. Since `Numerics.Pi` is necessarily a finite approximation of an irrational (nay, transcendental) value, such an implementation would actually compute the functions for a slightly different cycle, with the result that cumulative "phase shift" errors many cycles from the origin would be intolerable. Even relatively near the origin, the relative error near zeros of the functions would be excessive. An implementation that conforms to the accuracy requirements of the Numerics Annex will use rather different strategies to compute the functions relative to the implicit, natural cycle of 2p as opposed to an explicit cycle given exactly by the user. (In particular, an implementation of the former that simply invokes the latter with a cycle of `2.0*Numerics.Pi` will not conform to the Numerics Annex.)

Similar considerations form the basis for providing the natural logarithm function as a separate overloading, with an implicit base, rather than relying on the version with a base parameter and a default value of `Numerics.e` for that parameter.

In an early draft of Ada 95, the overloading of the exponentiation operator for a pair of floating point operands had parameter names of `X` and `Y`, following the style adopted for the other subprograms in `Numerics.Generic_Elementary_Functions`. It was subsequently deemed important for new overloadings of existing arithmetic operators to follow the precedent of using `Left` and `Right` for the names of their parameters, as in Ada 83.

The exponentiation operator is noteworthy in another respect. Instead of delivering `1.0` as one might expect by analogy with `0.0**0`, the expression `0.0**0.0` is defined to raise `Numerics.Argument_Error`. This is because $0.0^{0.0}$ is mathematically undefined, and indeed x^y can approach any value as x and y approach zero, depending on precisely *how* x and y approach zero. If `X` and `Y` could both be zero when an application evaluates `X**Y`, it seems best to require the application to decide in advance what it means and what the result should be. An application can do that by defining its own exponentiation operator, which would

- invoke the one obtained by instantiating the elementary functions package, and
- handle an `Argument_Error` exception raised by the latter, delivering from the handler the appropriate application-dependent value.

The local exponentiation operator can be inlined, if the extra level of subprogram linkage would be of concern.

The Ada 95 version uses `Float_Type'Base` as a type mark in declarations; this was not available in Ada 83. Thus the formal parameter types and result types of the functions are of the unconstrained (base) subtype of the generic formal type `Float_Type`, eliminating the possibility of range violations at the interface. The same feature can be used for local variables in implementations of `Numerics.Generic_Elementary_Functions` (if it is programmed in Ada) to avoid spurious exceptions caused by range violations on assignments to local variables having the precision of `Float_Type`. Thus, in contrast to [ISO 94a] there is no need to allow implementations to impose the restriction that the generic actual subtype must be an unconstrained subtype; implementations must allow any floating point subtype as the generic actual subtype, and they must be immune to the potential effects of any range constraint of that subtype.

Implementations in hardware sometimes do not meet the desired accuracy requirements [Tang 91] because the representation of p contained on the hardware chip has insufficient precision. To allow users to choose between fast (but sometimes inaccurate) versions of the elementary functions implemented in hardware and slightly slower versions fully conforming to realistic accuracy requirements, we introduced the concept of a pair of modes, "strict" and "relaxed". No accuracy requirements apply in the relaxed mode, or if the Numerics Annex is not supported. (These modes govern all numeric accuracy issues, not just those connected with the elementary functions.)

The accuracy requirements of the strict mode are not trivial to meet, but neither are they particularly burdensome; their feasibility has been demonstrated in a public-domain implementation using table-driven techniques. However, it should be noted that most vendors of serious mathematical libraries, including the hardware vendors, are now committing themselves to implementations that are fully accurate throughout the domain, since practical software techniques for achieving that accuracy are becoming more widely known. The accuracy requirements in Ada 95 are not as stringent as those which vendors are now striving to achieve.

Certain results (for example, the exponential of zero) are prescribed to be exact, even in the relaxed mode, because of the frequent occurrence of the corresponding degenerate cases in calculations and because they are inexpensively provided. Also, although the accuracy is implementation-defined in relaxed mode, nothing gives an implementation license to raise a spurious exception when an intermediate result overflows but the final result does not. Thus, implementations of the forward hyperbolic functions need to be somewhat more sophisticated than is suggested by the usual textbook formulae that compute them in terms of exponentials.

An implementation that accommodates signed zeros, such as one on IEEE hardware (where `Float_Type'Signed_Zeros` is true), is required to exploit them in several important contexts, in particular the signs of the zero results from the "odd" functions `Sin`, `Tan`, and their inverses and hyperbolic analogs, at the origin, and the sign of the half-cycle result from `Arctan` and `Arccot`; this follows a recommendation [Kahan 87] that provides important benefits for complex elementary functions built upon the real elementary functions, and for applications in conformal mapping. Exploitation of signed zeros at the many other places where the elementary functions can return zero results is left implementation-defined, since no obvious guidelines exist for these cases.

A.3.2 Random Number Generation

The capability of generating random numbers is required for many applications. It is especially common in simulations, even when other aspects of floating point computation are not heavily stressed. Indeed, some applications of random numbers have no need at all for floating point computation. For these reasons, Ada 95 provides in the predefined language environment a package, `Numerics.Float_Random`, that defines types and operations for generating random floating point numbers uniformly distributed over the range `0.0 .. 1.0` and a generic package, `Numerics.Discrete_Random`, that defines types and operations for generating uniformly distributed random values of a discrete subtype specified by the user.

As a simple example, various values of a simulated uniform risk could be generated by writing

```
use Ada.Numerics.Float_Random;
Risk: Float range 0.0 .. 1.0;
G: Generator;
...
loop
   Risk := Random(G);    -- a new value for risk
   ...
end loop;
...
```

It has been the custom in other languages (for example, Fortran 90) to provide only a generator of uniformly distributed random floating point numbers and to standardize the range to `0.0 .. 1.0`. Usually it is also stated that the value `1.0` is never generated, although values as close to `1.0` as the hardware permits may be generated. Sometimes the value `0.0` is excluded from the range instead, or in addition. The user who requires random floating point numbers uniformly distributed in some other range or having some other distribution, or who requires uniformly distributed random integers, is required to figure out and implement a conversion of what the language provides to the type and range desired. Although some conversion techniques are robust with respect to whether `0.0` or `1.0` can occur, others might fail to stay within the desired range, or might even raise an exception, should these extreme values be generated; with a user-designed conversion, there is also a risk of introducing bias into the distribution.

The random number facility designed for Ada 95 initially followed the same custom. However, concerns about the potential difficulties of user-designed post-generation conversion, coupled with the assertion that the majority of applications for random numbers actually need random integers, led to the inclusion of a capability for generating uniformly distributed random integers directly. The provision of that capability also allows for potentially more efficient implementations of integer generators, because it gives designs that can stay in the integer domain the freedom to do so.

Thus a random integer in the range 1 to 49 inclusive can be generated by

```
subtype Lotto is Integer range 1 .. 49;
package Lottery is new Ada.Numerics.Discrete_Random(Lotto);
use Lottery;
G: Generator;
Number: Lotto;
...
loop
   Number := Random(G);    -- next number for lottery ticket
   ...
end loop;
...
```

The use of generics to parameterize the integer range desired seemed obvious and appropriate, because most applications for random integers need a sequence of values in some fixed, problem-dependent subtype. As an alternative or a potential addition, we considered specifying the range dynamically on each call for a random number; this would have been convenient for those applications that require a random integer from a different range on each call. Reasoning that such applications are rare, we left their special needs to be addressed by using the floating point generator, coupled with post-generation conversion to the dynamically varying integer range. Help for the occasional user who faces the need to perform such a conversion is provided by a note in the reference manual, which describes a robust conversion technique [RM95 A.5.2(50..52)].

Note that the parameter of `Discrete_Random` can be of any discrete subtype and so one can easily obtain random Boolean values, random characters, random days of the week and so on.

Once the potential conversion problems had been solved by the combination of providing a generic discrete generator and documenting a robust conversion technique for the small number of applications that cannot use the generic generator, some of the pressure on the floating point generator was relieved. In particular, it was no longer necessary to specify that it must avoid generating `0.0` or `1.0`. The floating point generator is allowed to yield any value in its range, which can be described as the range `0.0 .. 1.0` without further qualification. Of course, some implementations may be incapable of generating `0.0` or `1.0`, but the user does not need to know that and would be better off not knowing it (portability could be compromised by exploiting knowledge that a particular implementation of the random number generator cannot deliver one or both bounds of the range). A note in the reference manual [RM95 A.5.2(52..54)] discusses ways of transforming the result of the floating point generator, using the `Log` function, into exponentially distributed random floating point numbers, illustrating a technique that avoids the `Argument_Error` exception that `Log` would raise when the value of its parameter is zero.

Generators

With the obvious exception of the result subtype, the two predefined packages declare the same types and operations, thereby simplifying their description and use. In the remainder of this section, we therefore discuss the contents of the packages without (in most cases) naming one or the other.

Applications vary widely in their requirements for random number generation. Global floating point and discrete random number generators would suffice for most applications, but more demanding applications require multiple generators (either one in each of several tasks or several in one task), with each generator giving rise to a different sequence of random numbers. For this reason, we provide in both packages a type called `Generator`, each of whose objects is associated with a distinct sequence of random numbers.

Operations on generators, such as obtaining the "next" random number from the associated sequence, are provided by subprograms that take an object of type `Generator` as a parameter. Applications requiring multiple generators can declare the required number of objects of type `Generator` in the tasks where they are needed. The mechanism is simple enough, however, not to be burdensome for applications requiring only a single global generator, which can be declared in the main program or in a library package.

(We entertained the idea of also having an implicit generator, which would be used when the `Generator` parameter is omitted from an operation on generators. This idea was abandoned, however, when agreement could not be reached on the question of whether the implicit generator should be local to each task or global to the tasks in a partition, and, in the latter case, whether serialization should be automatically provided for concurrent operations on the default generator performed in different tasks. To do so would be likely to impose an unnecessary overhead on applications that do no tasking and require only a single generator. The mechanisms provided in the random number packages and elsewhere in the language, particularly protected types and the generic package `Ada.Task_Attributes`, are sufficient to allow the developer of an advanced application, or the designer of a secondary library, to provide these capabilities, if desired.)

State Information

A generator obviously has state information associated with it, which reflects the current position in the associated sequence of random numbers and provides the basis for the computation of the next random number in the sequence. To allow the implementation wide latitude in choosing appropriate algorithms for generating random numbers, and to enforce the abstraction of a generator, the `Generator` type is a private type; furthermore, to enforce the distinctness of different generators, the type is limited. The full type is implementation defined.

For convenience of use, we chose to make `Random` a function. Since its parameter must therefore be of mode in, the `Generator` type in practice has to be realized either as an access type or (if its storage is to be reclaimed through finalization on exit from the generator's scope) as a controlled type containing an access type.

Applications that use random numbers vary also in their requirements for repeatability as opposed to uniqueness of the sequence of random numbers. Repeatability is desired during development and testing but often not desired in operational mode when a unique sequence of random numbers is required in each run. To meet both of these needs, we have specified that each generator always starts in the same, fixed (but implementation-defined) state, providing repeatable sequences by default, and we have provided several operations on generators that can be used to alter the state of a generator.

Calling the `Reset` procedure on a generator without specifying any other parameter sets the state to a time-dependent value in an implementation-dependent way.

The `Reset` procedure can also be used to ensure that task-local generators yield different, but repeatable, sequences. Note that by default, the fixed initial state of generators will result in all such generators yielding the same sequence. This is probably not what is desired. We considered specifying that each generator should have a unique initial state, but there is no realistic way to provide for the desired repeatability across different runs, given that the nondeterministic nature of task interactions could result in the "same" tasks (in some logical sense) being created in a different order in different runs.

Assuming that each task has a generator, different-but-repeatable sequences in different tasks are achieved by invoking the `Reset` procedure with an integer `Initiator` parameter on each generator prior to generating random numbers. The programmer typically must provide integer values uniquely associated with each task's logical function, independent of the order in which the tasks are created. The specified semantics of `Reset` are such that each distinct integer initiator value will initiate a sequence that will not overlap any other sequence in a practical sense, if the period of a generator is long enough to permit that. At the very least, consecutive integers should result in very different states, so that the resulting sequences will not simply be offset in time by one element or a small number of elements.

Saving the State

Most applications will have no need for capabilities beyond those already described. A small number of applications may have the need to save the current state of a generator and restore it at a later time, perhaps in a different run. This can be done by calling the `Save` procedure and another overloading of the procedure `Reset`. The state is saved in a variable of the private type `State`.

As was said earlier, the realization of the internal state of a generator is implementation defined, so as to foster the widest possible innovation in the design of generators and generation algorithms. The state is thus private and might be represented by a single integer or floating point value, or it might be represented by an array of integer or floating point values together with a few auxiliary values, such as indices into the array.

Internal generator states can be exported to a variable of the type `State`, saved in a file, and restored in a later run, all without knowing the representation of the type.

We also provide an `Image` function, which reversibly converts a value of type `State` to one of type `String` in an implementation-defined way, perhaps as a concatenation of one or more integer images separated by appropriate delimiters. The maximum length of the string obtained from the `Image` function is given by the named number `Max_Image_Width`; images of states can be manipulated conveniently in strings of this maximum length obtained by the use of `Ada.Strings.Bounded`. Using `Save` and `Image`, one can examine (a representation of) the current internal state for debugging purposes; one might use these subprograms in an interactive debugger, with no advanced planning, to make a pencilled note of the current state with the intention of typing it back in later. This does not require knowledge of the mapping between states and strings.

The inverse operation, `Value`, converts the string representation of an internal state back into a value of type `State`, which can then be imported into a generator by calling the `Reset` procedure. This pair of subprograms supports, without knowledge of the mapping between states and strings, the restoration of a state saved in the form of a string. Of couse if one does know the implementation's mapping of strings to states, then one can use `Value` and `Reset` to create arbitrary internal states for experimentation purposes. If passed a string that cannot be interpreted as the image of a state, `Value` raises `Constraint_Error`. This is the only time that the possibly expensive operation of state validation is required; it is not required every time `Random` is called, nor even when resetting a generator from a state.

We considered an alternative design, perhaps closer to the norm for random number generators, in which `Random` is a procedure that acts directly on the state, the latter being held in storage provided by the user. There would be no need for the `Save` and `Reset` (from-saved-state) procedures in this design, since the generator and state types would effectively be one and the same. The only real problem with this design is that it necessitates making `Random` a procedure, which would interfere with the programmer's ability to compose clear and meaningful expressions.

Of course, most simple applications will have no need to concern themselves with the `State` type: no need to declare variables of type `State`, and no need to call `Save` or `Reset` with a state parameter.

Statistical Considerations

The result subtype of the `Random` function in `Float_Random` is a subtype of `Float` with a range of `0.0 .. 1.0`. The subtype is called `Uniformly_Distributed` to emphasize that a large set of random numbers obtained from this function will exhibit an (approximately) uniform distribution. It is the only distribution provided because it is the most frequently required distribution and because other distributions can be built on top of a uniform distribution using well-known techniques. In the case of `Discrete_Random`, it does not really make sense to consider other than uniform distributions.

No provision is made for obtaining floating point random numbers with a precision other than that of `Float`. One reason is that applications typically do not have a need either for extremely precise random floating point numbers (those with a very fine granularity) or for random floating point numbers with several different precisions. Assuming that they are to be used as real numbers, and not converted to integers, the precision of a set of random floating point numbers generally does not matter unless an immense quantity of them are to be consumed. High precision random floating point numbers would be needed if they were to be converted to integers in some very wide range, but the provision of `Discrete_Random` makes that unnecessary. A second reason for providing random floating point numbers only with the precision of `Float`, and especially for not providing them with a precision of the user's choice, is that algorithms for random number generation are often tied to the use of particular hardware representations, which essentially dictates the precision obtained.

Nothing is said about the number of distinct values between `0.0` and `1.0` that must be (capable of being) delivered by the `Random` function in `Float_Random`. Indeed, in the spirit of not requiring guaranteed numerical performance unless the Numerics Annex is implemented, the specification of `Float_Random` says nothing about the quality of the result obtained from `Random`, except that a large number of such results must appear to be approximately uniformly distributed. On the other hand, the Numerics Annex specifies the minimum period of the generation algorithm, a wide range of statistical tests that must be satisfied by that algorithm, and the resolution of the time-dependent `Reset` function. In implementations in which `Float` corresponds to the hardware's double-precision type, the floating point random number algorithm can be based on the use of single-precision hardware, and can coerce the single-precision results to double precision at the final step, provided that the statistical tests are satisfied, which is perfectly feasible.

Details of the statistical tests, which are adapted from [Knuth 81] and other sources, are provided in an annotation in the [AARM]. The tests applicable to the floating point random number generator facility all exploit the floating point nature of the random numbers directly; they do not convert the numbers to integers. Different tests are applicable to the discrete random number generator.

In the rare case that random floating point numbers of higher precision (finer granularity) than that of `Float` are needed, the user should obtain them by suitably combining two or more successive results from `Random`. For example, two successive values might be used to provide the high-order and the low-order parts of a higher-precision result.

Guaranteeing that all the values in a wide integer range will eventually be generated is, in general, rather difficult and so is not required for the discrete generator. Nevertheless, some guarantee of this nature is desirable for more modest ranges. We thus require that if the range of the subtype has 2^{15} or fewer values then each value of the range will be delivered in a finite number of calls. This coverage requirement is in the specification of `Discrete_Random` in the Predefined Language Environment Annex; because it so directly affects the usability of the discrete random number generator facility, it was not thought appropriate to relegate the coverage requirement to the (optional) Numerics Annex. It is practical to verify by testing that the coverage requirement is satisfied for ranges up to this size, but it is not practical to verify the same for significantly wider ranges; for that matter, only a very long-running application could detect that a wide integer range is not being completely covered by the random numbers that are generated. Satisfying the coverage requirement is easily achieved by an underlying floating point algorithm, even one implemented in single precision, that converts its intermediate floating point result to the integer result subtype by appropriate use of scaling and type conversion.

The modest requirement discussed above does not completely eliminate all the difficulty in implementing `Discrete_Random`. Even the straightforward scaling and conversion technique faces mundane problems when the size of the integer range exceeds `Integer'Last`. Note that the size of the range of the predefined subtype `Integer` exceeds `Integer'Last` by about a factor of two, so that an instantiation of `Discrete_Random` for that predefined subtype will have to confront certain mundane problems, even if it does not purport to cover that range completely. These implementation burdens could have been eliminated by imposing restrictions on the (size of the ranges of the) subtypes with which `Discrete_Random` could be instantiated, but such restrictions are inimical to the spirit of Ada.

Of course, implementations of `Discrete_Random` need not be based on an underlying floating point algorithm, and indeed, as has already been said, part of the justification for providing this package separately from `Float_Random` has to do with the efficiency gains that can be realized when the former is implemented in terms of an underlying integer algorithm, with no use of floating point at all. Nevertheless, it may be convenient and sufficiently efficient for the discrete generator facility to be implemented in terms of a floating point algorithm. There are implementations of the venerable multiplicative linear congruential generator with multiplier 7^5 and modulus 2^{31}-1 of [Lewis 69] and both the add-with-carry and subtract-with-borrow Fibonacci generators of [Marsaglia 91] that remain entirely within the floating point domain, and which therefore pay no premium for conversion from integer to floating point. These algorithms have been verified to pass the statistical requirements of the Numerics Annex. (Other algorithms that might be expected to pass, but that have not been explicitly tested, include the combination generators of [Wichmann 82] and [L'Ecuyer 88] and the x^2 mod N generators of [Blum 86]; each of the algorithms mentioned here has much to recommend it.)

A.3.3 Attributes of Real Types

Most of the attributes of floating and fixed point types are defined in the Predefined Language Environment Annex. These attributes are discussed elsewhere in this Rationale (see 6.2).

A.4 Input-Output

Enhancements to input-output include a facility for heterogeneous streams, additional flexibility for `Text_IO`, and further file manipulation capabilities.

A.4.1 Stream Input and Output

The packages `Sequential_IO` and `Direct_IO` have not proved to be sufficiently flexible for some applications because they only process homogeneous files. Even so this is fairly liberal in the case of `Sequential_IO` which now has the form

```
generic
   type Element_Type(<>) is private;
package Ada.Sequential_IO is ...
```

since the actual parameter can be any indefinite type and hence can be a class-wide type. This does not apply to `Direct_IO` which can only take a definite type as a parameter because of the need to index individual elements.

In order to provide greater flexibility, totally heterogeneous streams can be processed using the new package `Streams` [RM95 13.13] and several child packages [RM95 A.12].

The general idea is that there is a stream associated with any file declared using the package `Ada.Streams.Stream_IO`. Such a file may be processed sequentially using the stream mechanism and also in a positional manner similar to `Direct_IO`. We will consider the stream process first and return to positional use later.

The package `Streams.Stream_IO` enables a file to be created, opened and closed in the usual manner. Moreover, there is also a function `Stream` which takes a stream file and returns (an access to) the stream associated with the file. In outline the first part of the package is

```
package Ada.Streams.Stream_IO is
   type Stream_Access is access all Root_Stream_Type'Class;
   type File_Type is limited private;
   -- Create, Open, ...
   function Stream(File: in File_Type) return Stream_Access;
   ...
end Ada.Streams.Stream_IO;
```

Observe that all streams are derived from the abstract type `Streams.Root_Stream_Type` and access to a stream is typically through an access parameter designating an object of the type `Streams.Root_Stream_Type'Class`. We will return to the package `Streams` and the abstract type `Root_Stream_Type` in a moment.

Sequential processing of streams is performed using attributes `T'Read`, `T'Write`, `T'Input` and `T'Output`. These attributes are predefined for all nonlimited types. The user can replace them by providing an attribute definition clause and can also define such attributes explicitly for limited types. This gives the user fine control over the processing when necessary. The attributes `T'Read` and `T'Write` will be considered first; `T'Input` and `T'Output` (which are especially relevant to indefinite subtypes) will be considered later.

The attributes `Read` and `Write` take parameters denoting the stream and the element of type `T` thus

```
procedure T'Write(Stream : access Streams.Root_Stream_Type'Class;
                  Item   : in T);

procedure T'Read(Stream : access Streams.Root_Stream_Type'Class;
                 Item   : out T);
```

As a simple example, suppose we wish to write a mixture of integers, month names and dates where type `Date` might be

```
type Date is
   record
      Day   : Integer;
      Month : Month_Name;
      Year  : Integer;
   end record;
```

We first create a file using the normal techniques and then obtain an access to the associated stream. We can then invoke the `Write` attribute procedure on the values to be written to the stream. We have

```
use Streams.Stream_IO;
Mixed_File : File_Type;
S          : Stream_Access;
...
Create(Mixed_File);
S := Stream(Mixed_File);
...
Date'Write(S, Some_Date);
Integer'Write(S, Some_Integer);
Month_Name'Write(S, This_Month);
...
```

Note that `Streams.Stream_IO` is not a generic package and so does not have to be instantiated; all such heterogeneous files are of the same type. Note also that they are binary files. A file written in this way can be read back in a similar manner, but of course if we attempt to read things with the inappropriate subprogram then we will get a funny value or `Data_Error`.

In the case of a simple record such as `Date` the predefined `Write` attribute simply calls the attributes for the components in order. So conceptually we have

```
procedure Date'Write(Stream : access Streams.Root_Stream_Type'Class;
                     Item   : in Date) is
begin
   Integer'Write(Stream, Item.Day);
   Month_Name'Write(Stream, Item.Month);
   Integer'Write(Stream, Item.Year);
end;
```

We can supply our own version of `Write`. Suppose for some reason that we wished to output the month name in a date as the corresponding integer; we could write

```
procedure Date_Write(Stream : access Streams.Root_Stream_Type'Class;
                     Item   : in Date) is
begin
   Integer'Write(Stream, Item.Day);
   Integer'Write(Stream, Month_Name'Pos(Item.Month) + 1);
   Integer'Write(Stream, Item.Year);
end Date_Write;
```

```
for Date'Write use Date_Write;
```

and then the statement

```
Date'Write(S, Some_Date);
```

will use the new format for the output of dates. Similar facilities apply to input and indeed if we wish to read the dates back in we would need to declare the complementary version of `Date'Read` to read the month as an integer and convert to the appropriate value of `Month_Name`.

Note that we have only changed the output of months in dates, if we wish to change the format of all months then rather than redefining `Date'Write` we could simply redefine `Month_Name'Write` and this would naturally have the indirect effect of also changing the output of dates.

Note carefully that the predefined attributes `T'Read` and `T'Write` can only be overridden by an attribute definition clause in the same package specification or declarative part where T is declared (just like any representation item). As a consequence these predefined attributes cannot be changed for the predefined types. But they can be changed for types derived from them.

The situation is slightly more complex in the case of arrays, and also records with discriminants, since we have to take account of the "dope" information represented by the bounds and discriminants. (In the case of a discriminant with defaults, the discriminant is treated as an ordinary component.) This is done using the additional attributes `Input` and `Output`. The general idea is that `Input` and `Output` process dope information (if any) and then call `Read` and `Write` to process the rest of the value. Their profiles are

```
procedure T'Output(Stream : access Streams.Root_Stream_Type'Class;
                   Item   : in T);

function T'Input(Stream: access Streams.Root_Stream_Type'Class)
    return T;
```

Note that `Input` is a function since `T` may be indefinite and we may not know the constraints for a particular call.

Thus in the case of an array the procedure `Output` outputs the bounds of the value and then calls `Write` to output the value itself.

In the case of a record type with discriminants, if it has defaults (is definite) then `Output` simply calls `Write` which treats the discriminants as just other components. If there are no defaults then `Output` first outputs the discriminants and then calls `Write` to process the remainder of the record. As an example consider the case of a definite subtype of a type whose first subtype is indefinite such as

```
subtype String_6 is String(1 .. 6);
S: String_6 := "String";
...
String_6'Output(S);        -- outputs bounds
String_6'Write(S);         -- does not output bounds
```

Note that the attributes `Output` and `Write` belong to the types and so it is immaterial whether we write `String_6'Write` or `String'Write`.

The above description of `T'Input` and `T'Output` applies to the default attributes. They could be redefined to do anything and not necessarily call `T'Read` and `T'Write`. Note moreover that `Input` and `Output` also exist for definite subtypes; their defaults just call `Read` and `Write`.

There are also attributes `T'Class'Output` and `T'Class'Input` for dealing with class-wide types. For output, the external representation of the tag (see [RM95 3.9]) is output and then the procedure `Output` for the specific type is called (by dispatching) in order to output the specific value (which in turn will call Write). Similarly on input, the tag is first read and then, according to its value, the corresponding function Input is called by dispatching. For completeness, T'Class'Read (T'Class'Write) is defined to dispatch to the subprogram denoted by the Read (respectively, Write) attribute of the specific type identified by the tag.

The general principle is, of course, that whatever is written can then be read back in again by the appropriate reverse operation.

We now return to a consideration of the underlying structure. All streams are derived from the abstract type Streams.Root_Stream_Type which has two abstract operations, Read and Write thus

```
procedure Read(Stream : in out Root_Stream_Type;
               Item   : out Stream_Element_Array;
               Last   : out Stream_Element_Offset) is abstract;

procedure Write(Stream : in out Root_Stream_Type;
                Item   : in Stream_Element_Array) is abstract;
```

These work in terms of stream elements rather than individual typed values. Note the difference between stream elements and storage elements (the latter being used for the control of storage pools which was discussed in 13.4). Storage elements concern internal storage whereas stream elements concern external information and are thus appropriate across a distributed system.

The predefined Read and Write attributes use the operations Read and Write of the associated stream, and the user could define new values for the attributes in the same way. Note, however, that the parameter Stream of the root type is of the type Root_Stream_Type whereas that of the attribute is an access type denoting the corresponding class. So any such user-defined attribute will have to do an appropriate dereference thus

```
procedure My_Write(Stream : access Streams.Root_Stream_Type'Class;
                   Item   : T) is
begin
   ... -- convert value into stream elements
   Streams.Write(Stream.all, ...);  -- dispatches
end My_Write;
```

We conclude by remarking that Stream_IO can also be used for indexed access. This is possible because the file is structured as a sequence of stream elements. Indexing then works in terms of stream elements much as Direct_IO works in terms of the typed elements. Thus the index can be read and reset. The procedures Read and Write process from the current value of the index and there is also an alternative Read that starts at a specified value of the index. The procedures Read and Write (which take a file as parameter) correspond precisely to the dispatching operations of the associated stream.

A.4.2 Text_IO

The main changes to Ada.Text_IO are the addition of internal generic packages Modular_IO (similar to Integer_IO) and Decimal_IO (similar to Fixed_IO).

There is also a completely distinct package Ada.Wide_Text_IO which provides identical facilities to Ada.Text_IO except that it works in terms of the types Wide_Character and Wide_String rather than Character and String. Text_IO and Wide_Text_IO declare distinct file types.

Both Text_IO and Wide_Text_IO have a child package Editing defined in the Information Systems Annex. This provides specialized facilities for the output of decimal values controlled by picture formats; for details see F.1. Similarly both packages have a child Complex_IO defined in the Numerics Annex; see G.1.3.

Small but important changes to `Text_IO` are the addition of subprograms `Look_Ahead`, `Get_Immediate` and `Flush`. The procedure `Look_Ahead` enables the next character to be determined without removing it and thereby enables the user to write procedures with similar behavior to predefined `Get` on numeric and enumeration types. The procedure `Get_Immediate` removes a single character from the file and bypasses any buffering that might otherwise be used; it is designed for interactive use. A call of `Flush` causes the remainder of any partly processed output buffer to be output.

A minor point is that the procedures `Get` for real types accept a literal in more liberal formats than in Ada 83. Leading and trailing zeros before or after the point are no longer required and indeed the point itself can be omitted. Thus the following are all acceptable forms for input for real types:

```
0.567
123.0
.567
123.
123
```

whereas in Ada 83 only the first two were acceptable. This is in some respects an incompatibility since a form such as `.567` would cause `Data_Error` to be raised in Ada 83. However, the main advantage is interoperability with other languages; data produced by Fortran programs can then be processed directly. Furthermore, the allowed formats are in accordance with ISO 6093:1985 which defines language independent formats for the textual representation of floating point numbers.

There are also nongeneric equivalents to `Integer_IO` and `Float_IO` for each of the predefined types `Integer`, `Long_Integer`, `Float`, `Long_Float` and so on. These have names such as `Ada.Integer_Text_IO`, `Ada.Long_Integer_Text_IO`, and `Ada.Float_Text_IO`. Observe that they are not child packages of `Ada.Text_IO` but direct children of `Ada`, thus allowing the names to be kept reasonably short.

A major reason for introducing these nongeneric equivalents was to facilitate teaching Ada to new users. Experience with teaching Ada 83 has shown that fundamental input-output was unnecessarily complicated by the reliance on generics, which gave the language an air of difficulty. So rather than writing

```
with Ada.Text_IO;
procedure Example is
   package Int_IO is new Ada.Text_IO.Integer_IO(Integer);
   use Int_IO;
   N: Integer;
begin
   ...
   Put(N);
   ...
end Example;
```

one can now perform simple output without needing to instantiate a generic

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Example is
   N: Integer;
begin
   ...
   Put(N);
   ...
end Example;
```

Another advantage of the nongeneric equivalents is that the user does not have to worry about an appropriate name for the instantiated version (and indeed fret over whether it might also be called Integer_IO without confusion with the generic version, or some other name such as we chose above). Having standard names also promotes portability since many vendors had provided such nongeneric equivalents but with different names.

Note carefully that these packages are said to be nongeneric equivalents rather than preinstantiated versions. This is so that implementations can use special efficient techniques not possible in the generic versions. A minor consequence is that the nongeneric equivalents cannot be used as actual package parameters corresponding to the generic package. Thus we cannot use Ada.Integer_Text_IO as an actual parameter to

```
generic
   with package P is new Ada.Text_IO.Integer_IO;
package Q is ...
```

Similar nongeneric equivalents apply to the generic packages for elementary functions, complex types and complex elementary functions, see A.3.1 and G.1.

Finally, it is possible to treat a Text_IO file as a stream and hence to use the stream facilities of the previous section with text files. This is done by calling the function Stream in the child package Text_IO.Text_Streams. This function takes a Text_IO file as parameter and returns an access to the corresponding stream. It is then possible to intermix binary and text input-output and to use the current file mechanism with streams.

A.4.3 File Manipulation

The Ada 83 package Sequential_IO did not make provision for appending data to the end of an existing file. As a consequence implementations provided a variety of solutions using pragmas and the form parameter and so on. In Ada 95 we have overcome this lack of portability by adding a further literal Append_File to the type File_Mode for Sequential_IO and Text_IO. It also exists for Stream_IO but not for Direct_IO.

The concept of a current error file for Text_IO is introduced, plus subprograms Standard_-Error, Current_Error and Set_Error by analogy with the similar subprograms for current input and current output. The function Standard_Error returns the standard error file for the system. On some systems standard error and standard output might be the same.

Error files are a convenience for the user; the ability to switch error files in a similar manner to the default output file enables the user to keep the real output distinct from error messages in a portable manner.

A problem with the Ada 83 subprograms for manipulating the current files is that it is not possible to store the current value for later use because the file type is limited private. As mentioned in 7.3, it is possible to temporarily "hang on" to the current value by the use of renaming thus

```
Old_File: File_Type renames Current_Output;
... -- set and use a different file
Set_Output(Old_File);
```

and thus permits some other file to be used and then the preexisting environment to be restored afterwards. This works because the result of a function call is treated like an object and can then be renamed. However, this technique does not permit a file value to be stored in an arbitrary way.

In order to overcome this difficulty, further overloadings of the various functions are introduced which manipulate an access value which can then be stored. Thus

```
type File_Access is access constant File_Type;
function Current_Input  return File_Access;
function Current_Output return File_Access;
function Current_Error  return File_Access;
```

and similarly for `Standard_Input` and so on. Additional procedures for setting the values are not required. We can then write

```
procedure P(...) is
   New_File     : File_Type;
   Old_File_Ref : constant File_Access := Current_Output;
begin
   Open(New_File, ...);
   Set_Output(New_File);
   -- use the new file
   Set_Output(Old_File_Ref.all);
   Close(New_File);
end P;
```

More sophisticated file manipulation is also possible. We could for example have an array or linked list of input files and then concatenate them for output. As another example, a utility program for pre-processing text files could handle nested "include"s by maintaining a stack of File_Access values.

Making the access type an access to constant prevents passing the reference to subprograms with **in out** parameters and thus prevents problems such as might arise from calling `Close` on `Current_Input`.

A.5 Command Line

The package `Ada.Command_Line` provides an Ada program with a simple means of accessing any arguments of the command which invoked it. The package also enables the program to set a return status. Clearly the interpretation and implementation of these facilities depends very much on the underlying operating system.

The function `Command_Name` returns (as a string) the command that invoked the Ada program and the function `Argument_Count` returns the number of arguments associated with the command. The function `Argument` takes an integer and returns the corresponding individual command argument also as a string.

The exit status can be set by a call of `Set_Exit_Status` which takes an integer parameter.

An alternative scheme based on using the parameters and results of the Ada main subprogram as the command arguments and exit status was rejected for a number of reasons. The main reason was that the start and end of the main subprogram are not the start and end of the execution of the Ada program as a whole; elaboration of library packages occurs before and might want access to command arguments and similarly, library tasks can outlive the main subprogram and might want to set the exit status.

A.6 Requirements Summary

The study topic

S10.4-A(1) — Varying-length String Package

is met by the bounded and unbounded-length string packages, and the study topic

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

is met in part by the string handling packages.

The requirement

R11.1-A(1) — Standard Mathematics Packages

is met by the generic elementary functions and random number packages.

The somewhat general requirement

R4.6-B(1) — Additional Input/Output Functions

calls for additional capability. In particular it suggests that there should be a standard way to append data to an existing file and the ability to have heterogeneous files. These specific requirements (and others) have been met as we have seen. Moreover the requirement

R4.6-A(1) — Interactive TEXT_IO

is specifically addressed by the introduction of the subprograms `Get_Immediate`, `Look_Ahead` and `Flush`; see A.4.2.

B Interface to Other Languages

It is very important for Ada 95 programs to be able to interface effectively with systems written in other languages. For example, the success of Ada 95 depends in part on its ability to cleanly and portably support interfaces to such systems as X Windows, POSIX, and commercial windows-based personal computer environments. (The portability in question is the ability to take a given Ada program or binding that interfaces with an external system, and move it to an environment with the same external system but a different Ada implementation.) To achieve this goal we have supplied three pragmas for interfacing with non-Ada software, and child packages `Interfaces.C`, `Interfaces.COBOL`, and `Interfaces.Fortran` which declare types, subprograms and other entities useful for interfacing with the three languages. The root package `Interfaces` contains declarations for hardware-specific numeric types, described in 3.3.

B.1 Interfacing Pragmas

Experience with pragma `Interface` in Ada 83 has uncovered a number of issues that may interfere with developing portable Ada code that is to be linked with foreign language modules. We have therefore removed pragma `Interface` (though the implementation may choose still to support it for upward compatibility) and have added the three pragmas `Import` (effectively replacing `Interface`), `Export` and `Convention` which provide the following capabilities:

- Calling Ada subprograms from other languages. Ada 83 only supported calls in one direction, from Ada to external code modules.

- Communicating with external systems via access to subprogram types.

- Specifying external names (and link names) where appropriate. Most Ada 83 implementations supported such an ability and it is beneficial to users that it be standardized [Fowler 89].

- Communicating with external systems via objects and other entities. Ada 83 only supported interfacing via subprogram calls.

The following example illustrates how Ada 95 procedures can call and be called from a program written in the C language.

```
type XT_Callback is access
   procedure (Widget_Id : in out XT_Intrinsics.Widget;
              Closure   : in X_Lib.X_Address;
              Call_Data : in X_Lib.X_Address);

pragma Convention(C, XT_Callback);

procedure XT_Add_Callback
      (The_Widget    : in out XT_Intrinsics.Widget;
       Callback_Name : in String;
       Callback      : in XT_Callback;
       Client_Data   : in XT_Intrinsics.XT_Pointer);
```

```
pragma Import(C, XT_Add_Callback, External_Name => "XtAddCallBack");

procedure My_Callback(Widget_Id : in out XT_Intrinsics.Widget;
                      Closure   : in X_Lib.X_Address;
                      Call_Data : in X_Lib.X_Address) is separate;

pragma Convention(C, My_Callback);

My_Widget : XT_Intrinsics.Widget;
...
XT_Add_Callback(My_Widget,
                "Mousedown" & ASCII.Nul,
                My_Callback'Access,
                XT_Intrinsics.Null_Data);
```

The pragma `Convention` applies to the type `XT_Callback`, and indicates that values of this type designate subprograms callable from programs written in C. The machine code generated for calls through the access values of the type `XT_Callback` will follow the conventions of the C compiler.

The pragma `Import` indicates that the procedure `XT_Add_Callback` is written with the calling conventions of a C compiler. The third parameter of the pragma specifies the external name (in this case the C name) of the subprogram.

The pragma `Convention` also applies to `My_Callback`. This informs the compiler that the procedure is written in Ada but is intended to be called from a C program, which may affect how it will reference its parameters.

`My_Callback'Access` will yield a value compatible with `XT_Callback`, because the same calling convention is specified for both. Note that it is unnecessary to apply the pragma `Export` to `My_Callback` since, although called from the C program, it is called indirectly through the access to subprogram value and the Ada identifier itself is not required externally.

The pragmas `Import` and `Export` may omit the external name if it is the same as the Ada identifier. A fourth parameter may be used to specify the link name if necessary.

The pragmas `Import` and `Export` may also be applied to objects. In particular a deferred constant can be completed by a pragma `Import`; this specifies that the object is defined externally to the Ada program. Similarly a pragma `Export` can be used to indicate that an object is used externally.

A programmer would typically use pragma Export in situations where the main subprogram is written in the external language. This raises some semantic issues, because correct execution of the exported Ada subprogram might depend on having certain Ada library units elaborated before the subprogram is invoked. For example, the subprogram might reference library package data objects that are initialized by the package body; or the subprogram might execute a construct (such as an allocator) that requires the Ada run-time system to have been elaborated. To handle such situations, Ada 95 advises the implementation [RM95 B.1(39)] to supply subprograms with link names `"adainit"` and `"adafinal"`. The `adainit` subprogram contains elaboration code for the Ada library units, and `adafinal` contains any needed finalization code (such as finalization of the environment task). Thus a main subprogram written in the external language should call `adainit` before the first call to an Ada subprogram, and `adafinal` after the last.

B.2 C Interface Package

The C interface package, `Interfaces.C`, supports importing C functions into Ada, and exporting Ada subprograms to C. Since many bindings and other external systems are written in C, one of the more important objectives of Ada 95 is to ease the job of having Ada code work with such software.

Part of the issue in arranging an interface to a foreign language, of particular importance with C, is to allow an Ada subprogram to be called from code written in the foreign language. This is handled in Ada 95 through a combination of pragma `Convention` and access to subprogram types, as illustrated above.

Further child packages `Interfaces.C.Strings` and `Interfaces.C.Pointers` provide specialized functionality for dealing with C strings and pointers.

B.2.1 Scalar Types

C's predefined integer, floating point, and character types are modelled directly in `Interfaces.C`. The Ada implementation is responsible for defining the Ada types such that they have the same representation as the corresponding C types in the supported C implementation.

Since C parameters are passed copy-in, interfacing to a C function taking a scalar parameter is straightforward. The program declares an Ada subprogram with an `in` parameter of the corresponding type.

A C function may have a `t*` parameter, where `t` is a scalar type, and where the caller is supposed to pass a reference to a scalar. If such a function is imported, then the corresponding Ada subprogram would declare either an **`access`** `T` parameter, or an **`in out`** `T` parameter.

B.2.2 Strings

C's string representation and manipulation come in several varieties, and we have tried to define the interface package so as to support the most typical applications. The `Interfaces.C` package provides an implementation-defined character type, `char`, designed to model the C run-time character type. This may or may not be the same as Ada's type `Character`; thus the package provides mappings between the types `char` and `Character`. Unlike COBOL, the mappings between the C and Ada character types do not need to be dynamically modifiable; hence they are captured by functions. In the common case where the character set is the same in C and Ada, the implementation should define the conversion functions through unchecked conversions expanded inline, with thus no run-time overhead.

One important application of the C interface package is for the programmer to compose a C string and pass it to a C function. We provide several ways to accomplish this goal. One approach is for the programmer to declare an object that will hold the C array, and then pass this array to the C function. This is realized via the type `char_array`:

```
type char_array is array (size_t range <>) of char;
```

The programmer can declare an Ada `String` and convert it to a `char_array` (or simply declare a `char_array` directly), and pass the `char_array` as actual parameter to the C function that is expecting a `char *`. The implication of pragma `Import` on the subprogram is that the `char_array` will be passed by reference, with no "descriptor" for the bounds; the compiler needs to implement this in such a way that what is passed is a pointer to the first element.

The package `Interfaces.C` , which provides the above conversions, is `Pure`; this extends its applicability in distributed applications that need to interface with C code.

An alternative approach for passing strings to C functions is for the programmer to obtain a C `char` pointer from an Ada `String` (or from a `char_array`) by invoking an allocation function. The child package `Interfaces.C.Strings` provides a private type `chars_ptr` that corresponds to C's `char *`, and two allocation functions. To avoid storage leakage, we also provide a `Free` procedure that releases the storage that was claimed by one of these allocate functions. If one of these allocate functions is invoked from an Ada program, then it is the responsibility of the Ada program (rather than the called C function) to reclaim that storage.

It is typical for a C function that deals with strings to adopt the convention that the string is delimited by a `nul` character. The C interface package supports this convention. A constant `nul` of type `char` is declared, and the function `Value(`*`chars_ptr`*`)` in `Interfaces.C.Strings` returns a `char_array` up to and including the first `nul` in the array that the `chars_ptr` points to.

Some C functions that deal with strings do not assume `nul` termination; instead, the programmer passes an explicit length along with the pointer to the first element. This style is also supported by `Interfaces.C`, since objects of type `char_array` need not be terminated by `nul`.

B.2.3 Pointers and Arrays

The generic package `Interfaces.C.Pointers` allows the Ada programmer to perform C-style operations on pointers. It includes an access type `Pointer`, `Value` functions that dereference a `Pointer` and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer. As in C, it treats an object `Ptr` of type `Pointer` as a pointer to the first element of an array, so that for example, adding `1` to `Ptr` yields a pointer to the second element of the array.

This generic package allows two styles of usage: one in which the array is terminated by a special terminator element; and another in which the programmer needs to keep track of the length.

This package may be used to interface with a C function that takes a `"*"` parameter. The `Pointer` type emerging from an instantiation corresponds to the `"*"` parameter to the C function.

B.2.4 Structs

If the C function expects a `"struct *"`, the Ada programmer should declare a corresponding simple record type and apply pragma `Convention` to this type. The Ada compiler will pass a reference to the record as the argument to the C function. Of course, it is not realistic to expect that any Ada record could be passed as a C struct; [RM95, B.1] allows restrictions so that only a "C-eligible" record type `T` need be supported for pragma `Convention(C, T)`. For example, records with discriminants or dynamically-sized components need not be supported. Nevertheless, the set of types for which pragma `Convention` needs to be supported is sufficiently broad to cover the kinds of interfaces that arise in practice.

In the (rare) situation where the C function takes a struct by value (for example a struct with a small number of small components), the programmer can declare a C function that takes a `struct *` and which then passes the value of its argument to the actual C function that is needed.

B.2.5 Example

The following example shows a typical use of the C interface facilities.

```
-- Calling the C Library Function strcpy

with Interfaces.C;
procedure Test is
   package C renames Interfaces.C;
   use type C.char_array;
   -- Call <string.h>strcpy:
   -- C definition of strcpy:
   --  char *strcpy(char *s1, const char *s2);
   --    This function copies the string pointed to by s2
   --    (including the terminating null character) into the
```

```
   --    array pointed to by s1.  If copying takes place
   --    between objects that overlap, the behavior is undefined.
   --    The strcpy function returns the value of s1.
   -- Note: since the C function's return value is of no interest,
   -- the Ada interface is a procedure

   procedure Strcpy(Target : out C.char_array;
                    Source : in  C.char_array);
   pragma Import(C, Strcpy, "strcpy");
   Chars1:  C.char_array(1 .. 20);
   Chars2:  C.char_array(1 .. 20);
begin
   Chars2(1 .. 6) := "qwert" & C.Nul;
   Strcpy(Chars1, Chars2);
   -- Now Chars1(1 .. 6) = "qwert" & C.Nul
end Test;
```

B.3 COBOL Interface Package

The package `Interfaces.COBOL` allows an Ada program to pass data as parameters to COBOL programs, allows an Ada program to make use of "external" data created by COBOL programs and stored in files or databases, and allows an Ada program to convert an Ada decimal type value to or from a COBOL representation.

In order to support the calling of and passing parameters to an existing COBOL program, the interface package supplies types that can be used in an Ada program as parameters to subprograms whose bodies will be in COBOL. These types map to COBOL's alphanumeric and numeric data categories.

Several types are provided for support of alphanumeric data. Since COBOL's run-time character set is not necessarily the same as Ada's, `Interfaces.COBOL` declares an implementation-defined character type `COBOL_Character` and mappings between `Character` and `COBOL_Character`. These mappings are visible variables (rather than, say, functions or constant arrays), since in the situation where `COBOL_Character` is EBCDIC, the flexibility of dynamically modifying the mappings is needed. Corresponding to COBOL's alphanumeric data is the array type `Alphanumeric`.

Numeric data may have either a "display" or "computational" representation in COBOL. On the Ada side, the data is of a decimal fixed point type. Passing an Ada decimal data item to a COBOL program requires conversion from the Ada decimal type to some type that reflects the representation expected on the COBOL side.

- Computational Representation

  Floating point representation is modelled by Ada floating point types, `Floating` and `Long_Floating`. Conversion between these types and Ada decimal types is obtained directly, since the type name serves as a conversion function.

  Binary representation is modelled by an Ada integer type, `Binary`, and possibly other types such as `Long_Binary`. Conversion between, say, `Binary` and a decimal type is through functions from an instantiation of the generic package `Decimal_Conversions`. An integer conversion using say `Binary` as the target and an object of a decimal type as the source does not work, since there would be no way to take into account the scale implicitly associated with the decimal type.

Packed decimal representation is modelled by the Ada array type `Packed_Decimal`. Conversion between packed decimal and a decimal type is through functions from an instantiation of the generic package `Decimal_Conversions`.

- Display Representation

  Display representation for numeric data is modelled by the array type `Numeric`. Conversion between display representation and a decimal type is through functions from an instantiation of the generic package `Decimal_Conversions`. A parameter to the conversion function indicates the desired interpretation of the data (e.g., signed leading separate, etc.)

The pragma `Convention(COBOL, T)` may be applied to a record type `T` to direct the compiler to choose a COBOL-compatible representation for objects of the type.

The package `Interfaces.COBOL` allows the Ada programmer to deal with data from files or databases created by a COBOL program. For data that is alphanumeric, or in display or packed decimal format, the approach is the same as for passing parameters: instantiate `Decimal_Conversions` to obtain the needed conversion functions. For binary data, the external representation is treated as a `Byte` array, and an instantiation of `Decimal_Conversions` produces a package that declares the needed conversion functions. A parameter to the conversion function indicates the desired interpretation of the data (e.g., high- versus low-order byte first).

We had considered defining the binary conversion functions in terms of a `Storage_Array` rather than a `Byte_Array` for the "raw data". However, `Storage_Array` reflects the properties of the machine that is running the Ada program, whereas the external file may have been produced in a different environment. Thus it is simpler to use a model in terms of COBOL-character-sized units.

The following examples show typical uses of the COBOL interface.

```
with Interfaces.COBOL;
procedure Test_Call is
   -- Calling a foreign COBOL program
   -- Assume that a COBOL program PROG has the following declaration
   --  in its LINKAGE section:
   --  01 Parameter-Area
   --     05 NAME   PIC X(20).
   --     05 SSN    PIC X(9)
   --     05 SALARY PIC 99999V99 USAGE COMP.
   -- The effect of PROG is to update SALARY based on some algorithm

   package COBOL renames Interfaces.COBOL;
   type Salary_Type is delta 0.01 digits 7;

   type COBOL_Record is
      record
         Name   : COBOL.Numeric(1 .. 20);
         SSN    : COBOL.Numeric(1 .. 9);
         Salary : COBOL.Binary;  -- Assume Binary = 32 bits
      end record;
   pragma Convention(COBOL, COBOL_Record);

   procedure Prog(Item : in out COBOL_Record);
   pragma Import(COBOL, Prog, "PROG");

   package Salary_Conversions is
      new COBOL.Decimal_Conversions(Salary_Type);

   Some_Salary : Salary_Type := 12_345.67;
```

```
      Some_Record : COBOL_Record :=
         (Name   => "Johnson, John       ",
          SSN    => "111223333",
          Salary => Salary_Conversions.To_Binary(Some_Salary));

   begin
      Prog(Some_Record);
      ...
   end Test_Call;


   with Interfaces.COBOL;
   with COBOL_Sequential_IO; -- Assumed to be supplied by implementation
   procedure Test_External_Formats is
      -- Using data created by a COBOL program
      -- Assume that a COBOL program has created a sequential file with
      --  the following record structure, and that we need to
      --  process the records in an Ada program
      --  01 EMPLOYEE-RECORD
      --     05 NAME    PIC X(20).
      --     05 SSN     PIC X(9)
      --     05 SALARY  PIC 99999V99 USAGE COMP.
      --     05 ADJUST  PIC S999V999 SIGN LEADING SEPARATE
      -- The COMP data is binary (32 bits), high-order byte first
      package COBOL renames Interfaces.COBOL;

      type Salary_Type is delta 0.01 digits 7 range 0.0 .. 99_999.99;
      type Adjustments_Type is delta 0.001 digits 6;

      type COBOL_Employee_Record_Type is  -- External representation
         record
            Name    : COBOL.Alphanumeric(1 .. 20);
            SSN     : COBOL.Alphanumeric(1 .. 9);
            Salary  : COBOL.Byte_Array(1 .. 4);
            Adjust  : COBOL.Numeric(1 .. 7);  -- Sign and 6 digits
         end record;
      pragma Convention(COBOL, COBOL_Employee_Record_Type);

      package COBOL_Employee_IO is
         new COBOL_Sequential_IO(COBOL_Employee_Record_Type);
      use COBOL_Employee_IO;

      COBOL_File : File_Type;

      type Ada_Employee_Record_Type is  -- Internal representation
         record
            Name    : String(1 .. 20);
            SSN     : String(1 .. 9);
            Salary  : Salary_Type;
            Adjust  : Adjustments_Type;
         end record;

      COBOL_Record : COBOL_Employee_Record_Type;

      Ada_Record   : Ada_Employee_Record_Type;

      package Salary_Conversions is
         new COBOL.Decimal_Conversions(Salary_Type);
      use Salary_Conversions;
```

```
      package Adjustments_Conversions is
         new COBOL.Decimal_Conversions(Adjustments_Type);
      use Adjustments_Conversions;
   begin
      Open(COBOL_File, Name => "Some_File");
      loop
         Read(COBOL_File, COBOL_Record);
         Ada_Record.Name := To_Ada(COBOL_Record.Name);
         Ada_Record.SSN  := To_Ada(COBOL_Record.SSN);
         begin
            Ada_Record.Salary :=
               To_Decimal(COBOL_Record.Salary, High_Order_First);
         exception
            when Conversion_Error =>
               ... -- Report "Invalid Salary Data"
         end;
         begin
            Ada_Record.Adjust :=
               To_Decimal(COBOL_Record.Adjust, Leading_Separate);
         exception
            when Conversion_Error =>
               ... -- Report "Invalid Adjustment Data"
         end;
         ... -- Process Ada_Record
      end loop;
   exception
      when End_Error => ...
   end Test_External_Formats;
```

B.4 Fortran Interface Package

Much mathematical software exists and continues to be written in Fortran and so there is a strong need for Ada programs to be able to interface to Fortran routines. Ada programs should be able to call Fortran subprograms, or Fortran library routines, passing parameters mapped the way Fortran would map them. Similarly, with increasing frequency, there will also be reasons for Fortran programs to call Ada subprograms as if they were written in Fortran (that is, with parameters passed in the normal way for Fortran). The Numerics Annex recommends that the facilities for interfacing to Fortran described in the annex on Interface to Other Languages be implemented if Fortran is widely supported in the target environment. Some high-performance mathematical software is also written in C, so a similar recommendation is made with regard to the facilities for interfacing to C. We discuss only the Fortran interfacing facilities here.

Interfacing to Fortran is provided by the child package `Interfaces.Fortran` and the convention identifier `Fortran` in the interfacing pragmas.

The package `Interfaces.Fortran` defines types having the same names as the Fortran intrinsic types (except where they would conflict with the names of Ada types predefined in `Standard`, in which case they are given different names) and whose representations match the default representations of those types in the target Fortran implementation. Multiple Fortran interface packages may be provided if several different implementations of Fortran are to be accommodated in the target environment; each would have an identifier denoting the corresponding implementation of Fortran. The same identifier would be used to denote that implementation in the interfacing pragmas.

Additional types may be added to a Fortran interface package as appropriate. For example, the package for an implementation of Fortran 77 might add declarations for `Integer_Star_2`, `Integer_Star_4`, `Logical_Star_1`, `Logical_Star_4`, and so on, while one for an implementation of Fortran 90 might add declarations for `Integer_Kind_0`, `Integer_Kind_1`, `Real_Kind_0`, `Real_Kind_1`, and so forth.

Use of the types defined in a Fortran interface package suffices when the application only requires scalar objects to be passed between Ada and Fortran subprograms. The `Convention` pragma can be used to indicate that a multidimensional array is to be mapped in Fortran's column-major order, or that a record object declared in a library subprogram or package is to be mapped the way Fortran would map a common block (the `Import` or `Export` pragma would also be specified for the object), or that a record type is to be mapped the way Fortran 90 would map a corresponding type (called a "derived type" in Fortran 90). Compatibility with Fortran 90's pointer types is provided by applying the `Convention` pragma to appropriate access types.

B.3 Requirements Summary

The requirement

R4.1-B(2) — Pragma Interface

is met by the introduction of the pragmas `Convention`, `Import` and `Export` for the better control of interfaces to programs in other languages.

The study topic

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

is met in part by the generic package `Interfaces.COBOL.Decimal_Conversions`.

The study topic

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

is satisfied in part by the facilities provided in `Interfaces.COBOL.Decimal_Conversions`.

The study topic

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

is satisfied in part by the types and conversions in the package `Interfaces.COBOL`.

The study topic

S11.2-A(1) — Array Representation

is met by the pragma `Convention` with a Fortran convention identifier, and more generally by the package `Interfaces.Fortran`.

C Systems Programming

The Systems Programming Annex specifies additional capabilities for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems.

The purpose of the Annex is to provide facilities for applications that are required to interface and interact with the outside world (i.e. outside the domain of an Ada program). Examples may be other languages, an operating system, the underlying hardware, devices and I/O channels. Since these kinds of interfaces lie outside the Ada semantic model, it is necessary to resort to low-level, environment specific programming paradigms. Such sections of the application are often implementation dependent and portability concerns are less critical. However, rigid isolation of these components helps in improving the portability of the rest of the application.

The application domains of such systems include: real-time embedded computers, I/O drivers, operating systems and run-time systems, multilingual/multicomponent systems, performance-sensitive hardware dependent applications, resource managers, user-defined schedulers, and so on. Accordingly, this annex covers the following facilities needed by such applications:

- Access to the underlying hardware (machine instructions, hardware devices, etc.);
- Access to the underlying operating or runtime system;
- Low-level, direct interrupt handling;
- Unchecked access to parts of the run-time model of the implementation;
- Specifying the representation and allocation of program data structures;
- Access to shared variables in a multitasking program;
- Access to the identity of tasks and the allocation of data on a per-task basis.

Note that implementation of this annex is a prerequisite for the implementation of the Real-Time Systems annex.

C.1 Access to Machine Operations

In systems programming and embedded applications, we need to write software that interfaces directly to hardware devices. This might be impossible if the Ada language implementation did not permit access to the full instruction set of the underlying machine. A need to access specific machine instructions arises sometimes from other considerations as well. Examples include instructions that perform compound operations atomically on shared memory, such as test-and-set and compare-and-swap, and instructions that provide high-level operations, such as translate-and-test and vector arithmetic.

It can be argued that Ada 83 already provides adequate access to machine operations, via the package `Machine_Code`. However, in practice, the support for this feature was optional, and some implementations support it only in a form that is inadequate for the needs of systems programming and real-time applications.

The mechanisms specified in this Annex for access to machine code are already allowed in Ada 83. The main difference is that now it is intended that the entire instruction set of a given machine should be accessible to an Ada program either via the `Machine_Code` package or via intrinsic subprograms (or indeed both). In addition, implementation-defined attributes ought to allow machine code to refer to the addresses or offsets of entities declared within the Ada program.

This Annex leaves most of the interface to machine code implementation defined. It is not appropriate for a language standard to specify exactly how access to machine operations must be provided, since machine instructions are inherently dependent on the machine.

We considered providing access to machine instructions only through interface to assembly language. This would not entirely satisfy the requirements, however, since it does not permit insertion of machine instructions in-line. Because the compiler cannot always perform register allocation across external subprogram calls, such calls generally require the saving and restoring of all registers. Thus, the overhead of assembly language subprogram calls is too high where the effect of a single instruction (e.g. test-and-set or direct I/O) is desired. For this, an in-line form of access is required. This requirement is satisfied by machine-code inserts or intrinsic subprograms.

To be useful, a mechanism for access to machine code must permit the flow of data and control between machine operations and the rest of the Ada program. There is not much value in being able to generate a machine code instruction if there is no way to apply it to operands in the Ada program. For example, an implementation that only permits the insertion of machine code as numeric literal data would not satisfy this requirement, since there would be no way for machine code operations to read or write the values of variables of the Ada program, or to invoke Ada procedures. However, this can be entirely satisfied by a primitive form of machine-code insertion, which allows an instruction sequence to be specified as a sequence of data values, so long as symbolic references to Ada entities are allowed in such a data sequence.

For convenience, it is desirable that certain instructions that are used frequently in systems programming, such as test-and-set and primitive I/O operations, be accessible as intrinsic subprograms, see [RM95 6.3.1]. However, it is not clear that it is practical for an implementation to provide access to all machine instructions in this form. Thus, it might be desirable to provide machine code inserts for generality, and intrinsic operations for convenient access to the more frequently needed operations.

The Pragma Export

The implementation advice concerning the pragma `Export` [RM95 B.1] addresses unintended interactions between compiler/linker optimizations and machine code inserts in Ada 83. A machine code insert might store an address, and later use it as a pointer or subprogram entry point — as with an interrupt handler. In general, the compiler cannot detect how the variable or subprogram address is used. When machine code is used in this way, it is the programmer's responsibility to inform the compiler about these usages, and it is the language's responsibility to specify a way for the programmer to convey this information. Without this information, the compiler or linker might perform optimizations so that the data object or subprogram code are deleted, or loads and stores referencing the object are suppressed.

In Ada 95, machine code subprograms are like external subprograms written in another language, in that they may be opaque to optimization. That is, in general, the compiler cannot determine which data objects a machine code subprogram might read or update, or to where it might transfer control. The `Export` pragma tells the compiler not to perform optimizations on an exported object. By requiring the user to specify as exported anything that might be modified by an external call, the compiler is provided with information that allows better optimization in the general case.

`Export` can also be used to ensure that the specified entity is allocated at an addressable location. For example, this might mean that a constant must actually be stored in memory, rather than only inserted in-line where used.

Interface to Assembly Language

An Ada implementation conforming to this Annex should also support interface to the traditional "systems programming language" for that target machine. This might be necessary to interface with existing code provided by the hardware vendor, such as an operating system, device drivers, or built-in-test software. We considered the possibility of requiring support for an assembler, but this has obvious problems. It is hard to state this requirement in a way that would not create enforcement problems. For example, what if there are several assemblers available for a given target, and new assemblers are developed from time to time? Which ones must an implementor support? Likewise, how hard does an implementor need to look before concluding there are no assemblers for a given target? However, we believe that stating the requirement simply as "should support interface to assembler" together with market forces will provide the appropriate direction for implementors in this area, even though compliance can not be fully defined.

Documentation Requirements

The intent of the documentation requirements is to ensure that the implementation provides enough information for the user to write machine code subprograms that interact with the rest of the Ada program. To do so effectively, the machine code subprograms ought to be able to read constants and read and update variables (including protected objects), to call subprograms, and to transfer control to labels.

Validation

The specifications for machine code are not likely to be enforceable by standard validation tests, but it should be possible to check for the existence of the required documentation and interfaces by examination of vendor supplied documentation, and to carry out spot checks with particular machine instructions.

C.2 Required Representation Support

The recommended levels of support defined in [RM95 13] are made into firm requirements if this annex is implemented because systems programming applications need to control data representations, and need to be able to count on a certain minimum level of support.

C.3 Interrupt Support

The ability to write handlers for interrupts is essential in systems programming and in real-time embedded applications.

The model of interrupts and interrupt handling specified in Ada 95 is intended to capture the common elements of most hardware interrupt schemes, as well as the software interrupt models used by some application interfaces to operating systems, notably POSIX [1003.1 90]. The specification allows an implementation to handle an interrupt efficiently by arranging for the interrupt handler to be invoked directly by the hardware. This has been a major consideration in the design of the interrupt handling mechanisms.

The reason for distinguishing *treatments* from handlers is that executing a handler is only one of the possible treatments. In particular, executing a handler constitutes delivery of the interrupt. The default treatment for an interrupt might be to keep the interrupt pending, or to discard it without delivery. These treatments cannot be modelled as a default handler.

The notion of *blocking* an interrupt is an abstraction for various mechanisms that may be used to prevent delivery of an interrupt. These include operations that "mask" off a set of interrupts, raise the hardware priority of the processor, or "disable" the processor interrupt mechanism.

On many hardware architectures it is not practical to allow a direct-execution interrupt handler to become blocked. Trying to support blocking of interrupt handlers results in extra overhead, and can also lead to deadlock or stack overflow. Therefore, interrupt handlers are not allowed to block. To enable programmers to avoid unintentional blocking in handlers, the language specifies which operations are potentially blocking, see [RM95 9.5.1].

We introduced the concept of *reserved* interrupts to reflect the need of the Ada run-time system to install handlers for certain interrupts, including interrupts used to implement time delays or various constraint checks. The possibility of simply making these interrupts invisible to the application was considered. This is not possible without restricting the implementation of the `Interrupt_ID` type. For example, if this type is an integer type and certain values within this range are reserved (as is the case with POSIX signals, for example), there is no way to prevent the application from *attempting* to attach a handler to one of the reserved interrupts; however, any such attempt will raise `Program_Error`. Besides, other (implementation-defined) uses for an interrupt-id type are anticipated for which the full range of values might be needed; if the standard interrupt-id type did not include all values, the implementation would have to declare an almost identical type for such purposes.

We also need to reserve certain interrupts for task interrupt entries. There are many ways in which implementations can support interrupt entries. The higher-level mechanisms involve some degree of interaction with the Ada run-time system. It could be disastrous if the run-time system is relying on one of these high-level delivery mechanisms to be in place, and the user installs a low-level handler. For this reason, the concept of reserved interrupt is used here also, to prevent attachment of another handler to an interrupt while an interrupt entry is still attached to it.

On some processor architectures, the priority of an interrupt is determined by the device sending the interrupt, independent of the identity of the interrupt. For this reason, we need to allow an interrupt to be generated at different priorities at different times. This can be modelled by hypothesizing several hardware tasks, at different priorities, which may all call the interrupt handler.

A consequence of direct hardware invocation of interrupt handlers is that one cannot speak meaningfully of the "currently executing task" within an interrupt handler (see [RM95 C.7.1]). The alternative, of requiring the implementation to create the illusion of an Ada task as context for the execution of the handler would add execution time overhead to interrupt handling. Since interrupts may occur very frequently, and require fast response, any such unnecessary overhead is intolerable.

For these and other reasons, care has been taken not to specify that interrupt handlers behave exactly as if they are called by a hardware "task". The language must not preclude the writing of efficient interrupt handlers, just because the hardware does not provide a reasonable way to preserve the illusion of the handler being called by a task.

The Annex leaves as implementation-defined the semantics of interrupts when more than one interrupt subsystem exists on a multi-processor target. This kind of configuration may dictate that different interrupts are delivered only to particular processors, and will require that additional rules be placed on the way handlers are attached. In essence, such a system cannot be treated completely as a homogeneous multi-processor. The means for identifying interrupt sources, and the specification of the circumstances when interrupts are blocked are therefore left open by the Annex. It is expected that these additional rules will be defined as a logical extension of the existing ones.

From within the program the form of an interrupt handler is a protected procedure. Typically we write

```
protected Alarm is
   procedure Response;
   pragma Attach_Handler(Response, Alarm_Int);
end Alarm;

protected body Alarm is
   procedure Response is
   begin
      ...  -- the interrupt handling code
   end Response;
end Alarm;
```

where `Alarm_Int` identifies the physical interrupt as discussed in C.3.2.

Protected procedures have appropriate semantics for fast interrupt handlers; they are directly invoked by the hardware and share data with tasks and other interrupt handlers. Thus, once the interrupt handler begins to execute it cannot block; on the other hand while any shared data is being accessed by other threads of control, an interrupt must be blocked.

For upward compatibility, the Ada 83 interrupt entry mechanism is retained although classified as obsolescent. It has been extended slightly, as a result of the integration with the protected procedure interrupt-handling model. In addition, this Annex does not preclude implementations from defining other forms of interrupt handlers such as protected procedures with parameters. The recommendation is that such extensions will follow the model defined by this Annex.

Exceptions and Interrupt Handlers

Propagating an exception from an interrupt handler is specified to have no effect. (If interrupt handlers were truly viewed as being "called" by imaginary tasks, the propagation of an exception back to the "caller" of an interrupt handler certainly should not affect any user-defined task.)

The real question seems to be whether the implementation is required to hide the effect of an interrupt from the user, or whether it can be allowed to cause a system crash. If the implementation uses the underlying interrupt mechanism directly, i.e. by putting the address of a handler procedure into an interrupt vector location, the execution context of the handler will be just the stack frame that is generated by the hardware interrupt mechanism. If an exception is raised and not handled within the interrupt handler, the exception propagation mechanism will try to unroll the stack, beyond the handler. There needs to be some indication on the stack that it should stop at the interrupt frame, and not try to propagate beyond. Lacking this, either the exception might just be propagated back to the interrupted task (if the hardware interrupt frame structure looks enough like a normal call), or the difference in frame structures will cause a failure. The failure might be detected in the run-time system, might cause the run-time system to crash, or might result in transfer of control to an incorrect handler, thereby causing the application to run amok. The desired behavior is for the exception mechanism to recognize the handler frame as a special case, and to simply do an interrupt return. Unless the hardware happens to provide enough information in the handler frame to allow recognition, it seems an extra layer of software will be needed, i.e. a software wrapper for the user interrupt handler. (This wrapper might be provided by the compiler, or by the run-time system.)

Thus, the requirement that propagating an exception from a handler be "safe" is likely to impose some extra run-time overhead on interrupt handlers but is justified by the additional safety it provides. It is not expected that exceptions will be raised intentionally in interrupt handlers, but when an unexpected error (a bug) causes an exception to be raised, it is much better to contain the effect of this error than to allow the propagation to affect arbitrary code (including the RTS itself).

Implementation Requirements

It is not safe to write interrupt handlers without some means of reserving sufficient stack space for them to execute. Implementations will differ in whether such handlers borrow stack space from the task they interrupt, or whether they execute on a separate stack. In either case, with dynamic attachment of interrupt handlers, the application needs to inform the implementation of its maximum interrupt stack depth requirement. This could be done via a pragma or a link-time command.

Documentation Requirements

Where hardware permits an interrupt to be handled but not to be blocked (while in the handler), it might not be possible for an implementation to support the protected object locking semantics for such an interrupt. The documentation must describe any variations from the model.

For example, in many implementations, it may not be possible for an interrupted task to resume execution until the interrupt handler returns. The intention here is to allow the implementation to choose whether to run the handler on a separate stack or not. The basic issue is whether the hardware (or underlying operating system) interrupt mechanism switches stacks, or whether the handler begins execution on the stack of the interrupted task. Adding software to the handler to do stack switching (in both directions) can add significantly to the run-time overhead, and this may be unacceptable for high frequency interrupts.

Situations in which this would make a difference are rather unusual. Since the handler can interrupt, the task that it interrupts must have a lower active priority at that time. Therefore, the only situations where the interrupted task would need to resume execution before the handler returns are:

- If there is more than one processor, and the interrupted task could migrate to another available processor;

- If the handler or some higher priority task causes the priority of the interrupted task to be raised (via a call to `Set_Priority`); or

- If priorities are not supported.

The semantic model, when the interrupt handler uses the stack of the interrupted task, is that the handler has taken a non-preemptable processing resource (the upper part of the stack) which the interrupted task needs in order to resume execution. Note that this stack space was not in use by the interrupted task at the time it was preempted, since the stack did not yet extend that far, but it is needed by the interrupted task before it can resume execution.

C.3.1 Protected Procedure Handlers

A handler can be statically attached to an interrupt by the use of the pragma `Attach_Handler` as in the example above. Alternatively the connection can be made dynamic by using the pragma `Interrupt_Handler` together with the procedure `Attach_Handler`. The example might then become

```
protected Alarm is
   procedure Response;
   pragma Interrupt_Handler(Response);
end Alarm;

protected body Alarm is
```

```
      procedure Response is
      begin
      ...  -- the interrupt handling code
      end Response;
   end Alarm;
   ...
   Attach_Handler(Alarm.Response, Alarm_Int);
```

Note therefore that the name `Attach_Handler` is used for both the pragma and for the procedure.

The procedure form is needed to satisfy the requirement for dynamic attachment. The pragma form is provided to permit attachment earlier, during initialization of objects, or possibly at program load time. Another advantage of the pragma form is that it permits association of a handler attachment with a lexical scope, ensuring that it is detached on scope exit. Note that while the protected type is required to be a library level declaration, the protected object itself may be declared in a deeper level.

Under certain conditions, implementations can preelaborate protected objects, see [RM95 10.2.1] and [RM95 C.4]. For such implementations, the `Attach_Handler` pragma provides a way to establish interrupt handlers truly statically, at program load time.

The `Attach_Handler and Interrupt_Handler` pragmas specify a protected procedure as one that is or may be used as an interrupt handler and (in the latter case) be attached dynamically. This has three purposes:

- It informs the compiler that it might need to generate code for a protected procedure that can be invoked directly by the hardware interrupt mechanism, and to generate appropriate calls to procedures contained in such an object if they are called from software.

- It allows the implementation to allocate enough space for the corresponding protected object to store the interrupt-id to which the handler is attached. (This might be needed on some implementations in order to mask that interrupt when operations on the protected object are called from software.)

- It serves as important documentation about the protected object.

In general, the hardware mechanism might require different code generation than for procedures called from software. For example, a different return instruction might be used. Also, the hardware mechanism may implicitly block and unblock interrupts, whereas a software call may require this to be done explicitly. For a procedure that can be called from hardware or software, the compiler generally must choose between:

- Compiling the procedure in the form for hardware invocation and adding some sort of glue-code when it is called via software;

- Compiling the procedure in the form for software invocation, and call it indirectly, from an extra layer of interrupt handler, when a hardware interrupt occurs.

Because code generation is involved, the pragma is associated with the protected type declaration, rather than with a particular protected object.

The restrictions on protected procedures should be sufficient to eliminate the need for an implementation to place any further restrictions on the form or content of an interrupt handler. Ordinarily, there should be no need for implementation-defined restrictions on protected procedure interrupt handlers, such as those imposed by Ada 83 on tasks with fast interrupt handler entries. However, such restrictions are permitted, in case they turn out to be needed by implementations.

This Annex requires only that an implementation support parameterless procedures as handlers. In fact, some hardware interrupts do provide information about the cause of the interrupt and the state of the processor at the time of the interrupt. Such information is also provided by operating systems that support software interrupts. The specifics of such interrupt parameters are necessarily dependent on the execution environment, and so are not suitable for standardization. Where appropriate, implementation-defined child packages of `Ada.Interrupts` should provide services for such interrupt handlers, analogous to those defined for parameterless protected procedures in the package `Ada.Interrupts` itself.

Note that only procedures of library-level protected objects are allowed as dynamic handlers. This is because the execution context of such procedures persists for the full lifetime of the partition. If local procedures were allowed to be handlers, some extra prologue code would need to be added to the procedure, to set up the correct execution environment. To avoid problems with dangling references, the attachment would need to be broken on scope exit. This does not seem practical for the handlers that might be attached and detached via a procedure interface. On the other hand, it could be practical for handlers that are attached via a pragma. Some implementations may choose to allow local procedures to be used as handlers with the `Attach_Handler` pragma.

For some environments, it may be appropriate to also allow ordinary subprograms to serve as interrupt handlers; an implementation may support this, but the mechanism is not specified. Protected procedures are the preferred mechanism because of the better semantic fit in the general case. However, there are some situations where the fit might not be so good. In particular, if the handler does not access shared data in a manner that requires the interrupt to be blocked, or if the hardware does not support blocking of the interrupt, the protected object model may not be appropriate. Also, if the handler procedure needs to be written in another language, it may not be practical to use a protected procedure.

Issues Related to Ceiling Priorities

With priority-ceiling locking, it is important to specify the active priority of the task that "calls" the handler, since it determines the ability of the interrupt to preempt whatever is executing at the time. It is also relevant to the user, since the user must specify the ceiling priority of the handler object to be at least this high, or else the program will be erroneous (might crash).

Normally, a task has its active priority raised when it calls a protected operation with a higher ceiling than the task's own active priority. The intent is that execution of protected procedures as interrupt handlers be consistent with this model. The ability of the interrupt handler "call" from the hardware to preempt an executing task is determined by the hardware interrupt priority. In this respect, the effect is similar to a call from a task whose active priority is at the level of the hardware interrupt priority. Once the handler begins to execute, its active priority is set to the ceiling priority of the protected object. For example, if a protected procedure of an object whose ceiling priority is 5 is attached as a handler to an interrupt of priority 3, and the interrupt occurs when a task of priority 4 runs, the interrupt will remain pending until there is no task executing with active priority higher than or equal to 3. At that point, the interrupt will be serviced. Once the handler starts executing, it will raise its active priority to 5.

It is impractical to specify that the hardware must perform a run-time check before calling an interrupt handler, in order to verify that the ceiling priority of the protected object is not lower than that of the hardware interrupt. This means that checks must either be done at compile-time, at the time the handler is attached, or by the handler itself.

The potential for compile-time checking is limited, since dynamic attachment of handlers is allowed and the priority can itself be dynamic; all that can be done is to verify that the handler ceiling is specified via the `Interrupt_Priority` pragma thus

```
protected Alarm is
   pragma Interrupt_Priority(N);
   procedure Response;
   pragma Interrupt_Handler(Response);
end Alarm;
```

Doing the check when the handler is attached is also limited on some systems. For example, with some architectures, different occurrences of the same interrupt may be delivered at different hardware priorities. In this case, the maximum priority at which an interrupt might be delivered is determined by the peripheral hardware rather than the processor architecture. An implementation that chooses to provide attach-time ceiling checks for such an architecture could either assume the worst (i.e. that all interrupts can be delivered at the maximum priority) or make the maximum priority at which each interrupt can be delivered a configuration parameter.

A last-resort method of checking for ceiling violations is for the handler to start by comparing its own ceiling against the active priority of the task it interrupted. Presumably, if a ceiling violation were detected, the interrupt could be ignored or the entire program could be aborted. Providing the run-time check means inserting a layer of "wrapper" code around the user-provided handler, to perform the check. Executing this code will add to the execution time of the handler, for every execution. This could be significant if the interrupt occurs with high frequency.

Because of the difficulty of guaranteeing against ceiling violations by handlers on all architectures, and the potential loss of efficiency, an implementation is not required to detect situations where the hardware interrupt mechanism violates a protected object ceiling. Incorrect ceiling specification for an interrupt handler is "erroneous" programming, rather than a bounded error, since it might be impractical to prevent this from crashing the Ada RTS without actually performing the ceiling check. For example, the handler might interrupt a protected action while an entry queue is being modified. The epilogue code of the handler could then try to use the entry queue. It is hard to predict how bad this could be, or whether this is the worst thing that could happen. At best, the effect might be limited to loss of entry calls and corresponding indefinite blocking of the calling tasks.

Since the priorities of the interrupt sources are usually known a priori and are an important design parameter, it seems that they are not likely to vary a lot and create problems after the initial debugging of the system. Simple coding conventions can also help in preventing such cases.

Non-Suspending Locks

With interrupt handlers, it is important to implement protected object locking without suspension. Two basic techniques can be applied. One of these provides mutual exclusion between tasks executing on a single processor. The other provides mutual exclusion between tasks executing on different processors, with shared memory. The minimal requirement for locking in shared memory is to use some form of spin-wait on a word representing the protected object lock. Other, more elaborate schemes are allowed (such as priority-based algorithms, or algorithms that minimize bus contention).

Within a single processor, a non-suspending implementation of protected object locking can be provided by limiting preemption. The basic prerequisite is that *while a protected object is locked, other tasks that might lock that protected object are not allowed to preempt*. This imposes a constraint on the dispatching policy, which can be modelled abstractly in terms of *locking sets*. The locking set of a protected object is the set of tasks and protected operations that might execute protected operations on that object, directly or indirectly. More precisely, the locking set of a protected object `R` includes:

- Tasks that call protected operations of `R` directly.
- Protected procedures and entries whose bodies call a protected operation of `R`.
- The locking set of `Q`, if `Q` is a protected object with a procedure or entry whose body contains a call to an operation (including a requeue) in the locking set of `R`.

While a protected object is held locked by a task or interrupt handler, the implementation must prevent the other tasks and interrupt handlers in the locking set from executing on the same processor. This can be enforced conservatively, by preventing a larger set of tasks and interrupt handlers from executing. At one extreme, it may be enforced by blocking all interrupts, and disabling the dispatching of any other task. The priority-ceiling locking scheme described in [RM95 D.3] approximates the locking set a little less conservatively, by locking out all tasks and interrupt handlers with lower or equal priority to the one that is holding the protected object lock.

The above technique (for a single processor) can be combined with the spin-wait approach on a multiprocessor. The spinning task raises its priority to the ceiling or mask interrupts before it tries to grab the lock, so that it will not be preempted after grabbing the lock (while still being in the "wrong" priority).

Metrics

The interrupt handler overhead metric is provided so that a programmer can determine whether a given implementation can be used for a particular application. There is clearly imprecision in the definition of such a metric. The measurements involved require the use of a hardware test instrument, such as a logic analyzer; the inclusion of instructions to trigger such a device might alter the execution times slightly. The validity of the test depends on the choice of reference code sequence and handler procedure. It also relies on the fact that a compiler will not attempt in-line optimization of normal procedure calls to a protected procedure that is attached as an interrupt handler. However, there is no requirement for the measurement to be absolutely precise. The user can always obtain more precise information by carrying out specific testing. The purpose of the metric here is to allow the user to determine whether the implementation is close enough to the requirements of the application to be worth considering. For this purpose, the accuracy of a metric could be off by a factor of two and still be useful.

C.3.2 The Package Interrupts

The operations defined in the package `Ada.Interrupts` are intended to be a minimum set needed to associate handlers with interrupts. The type `Interrupt_ID` is implementation defined to allow the most natural choice of the type for the underlying computer architecture. It is not required to be private, so that if the architecture permits it to be an integer, a non-portable application may take advantage of this information to construct arrays and loops indexed by `Interrupt_ID`. It is, however, required to be a discrete type so that values of the type `Interrupt_ID` can be used as discriminants of task and protected types. This capability is considered necessary to allow a single protected or task type to be used as handler for several interrupts thus

```
Device_Priority: constant array (1 .. 5) of Interrupt_Priority :=
                                                          ( ... );
protected type Device_Interface(Int_ID: Interrupt_ID) is
   procedure Handler;
   pragma Attach_Handler(Handler, Int_ID);
   ...
   pragma Interrupt_Priority(Device_Priority(Int_ID));
end Device_Interface;
...
Device_1_Driver: Device_Interface(1);
...
Device_5_Driver: Device_Interface(5);
...
```

Some interrupts may originate from more than one device, so an interrupt handler may need to perform additional tests to decide which device the interrupt came from. For example, there might be several timers that all generate the same interrupt (and one of these timers might be used by the Ada run-time system to implement delays). In such a case, the implementation may define multiple logical interrupt-id's for each such physical interrupt.

The atomic operation `Exchange_Handler` is provided to attach a handler to an interrupt and return the previous handler of that interrupt. In principle, this functionality might also be obtained by the user through a protected procedure that locks out interrupts and then calls `Current_Handler` and `Attach_Handler`. However, support for priority-ceiling locking of protected objects is not required. Moreover, an exchange-handler operation is already provided in atomic form by some operating systems (e.g. POSIX). In these cases, attempting to achieve the same effect via the use of a protected procedure would be inefficient, if feasible at all.

The value returned by `Current_Handler` and `Exchange_Handler` in the case that the default interrupt treatment is in force is left implementation-defined. It is guaranteed however that using this value for `Attach_Handler` and `Exchange_Handler` will restore the default treatment. The possibility of simply requiring the value to be null in this case was considered but was believed to be an over-specification and to introduce an additional overhead for checking this special value on each operation.

Operations for blocking and unblocking interrupts are intentionally not provided. One reason is that the priority model provides a way to lock out interrupts, using either the ceiling-priority of a protected object or the `Set_Priority` operation (see [RM95 D.5]). Providing any other mechanism here would raise problems of interactions and conflicts with the priority model. Another reason is that the capabilities for blocking interrupts differ enough from one machine to another that any more specific control over interrupts would not be applicable to all machines.

In Ada 83, interrupt entries are attached to interrupts using values of the type `System.Address`. In Ada 95, protected procedures are attached as handlers using values of `Interrupt_ID`. Changing the rules for interrupt entries was not considered feasible as it would introduce upward-incompatibilities, and would require support of the `Interrupts` package by all implementations. To resolve the problem of two different ways to map interrupts to Ada types, the `Reference` function is provided. This function is intended to provide a portable way to convert a value of the type `Interrupt_ID` to a value of type `System.Address` that can be used in a representation clause to attach an interrupt entry to an interrupt source.

The Interrupts.Names Package

The names of interrupts are segregated into the child package `Interrupts.Names`, because these names will be implementation-defined. In this way, a use clause for package `Interrupts` will not hide any user-defined names.

C.3.3 Task Entries as Handlers

Attaching task entries to interrupts is specified as an obsolescent feature (see [RM95 J.7.1]). This is because support of this feature in Ada 83 was never required and important semantic details were not given. Requiring every implementation to support attaching both protected procedures and task entries to interrupts was considered to be an unnecessarily heavy burden. Also, with entries the implementation must choose between supporting the full semantics of rendezvous for interrupts (with more implementation overhead than protected procedures) versus imposing restrictions on the form of handler tasks (which will be implementation-dependent and subtle). The possibility of imposing standard restrictions, such as those on protected types, was considered. It was rejected on the grounds that it would not be upward compatible with existing implementations of interrupt entries (which are diverse in this respect). Therefore, if only one form of handler is to be supported, it should be protected procedures.

As compared to Ada 83, the specifications for interrupt entries are changed slightly. First, the implementation is explicitly permitted to impose restrictions on the form of the interrupt handler task, and on calls to the interrupt entry from software tasks. This affirms the existing practice of language implementations which support high-performance direct-execution interrupt entries. Second, the dynamic attachment of different handlers to the same interrupt, sequentially, is explicitly allowed. That is, when an interrupt handler task terminates and is finalized, the attachment of the interrupt entry to the interrupt is broken. The interrupt again becomes eligible for attachment. This is consistent with the dynamic attachment model for protected procedures as interrupt handlers, and is also consistent with a liberal reading of the Ada 83 standard. Finally, in anticipation of Ada 95 applications that use protected procedures as handlers together with existing Ada 83 code that uses interrupt entries, interrupts that are attached to entries, are specified as reserved, and so effectively removed from the set of interrupts available for attachment to protected procedures. This separation can therefore eliminate accidental conflicts in the use of values of the `Interrupt_ID` type.

C.4 Preelaboration Requirements

The primary systems programming and real-time systems requirement addressed by preelaboration is the fast starting (and possibly restarting) of programs. Preelaboration also provides a way to possibly reduce the run-time memory requirement of programs, by removing some of the elaboration code. This section is in the Systems Programming Annex (rather than the Real-Time Annex) because the functionality is not limited to real-time applications. It is also required to support distribution.

Rejected Approaches

There is a spectrum of techniques that can be used to reduce or eliminate elaboration code. One possible technique is to run the entire program up to a certain point, then take a snap-shot of the memory image, which is copied out and transformed into a file that can be reloaded. Program start-up would then consist of loading this check-point file and resuming execution. This "core-dump" approach is suitable for some applications and it does not require special support from the language. However, it is not really what has been known as preelaboration, nor does it address all the associated requirements.

The core-dump approach to accelerating program start-up suffers from several defects. It is error-prone and awkward to use or maintain. It requires the entire active writable memory of the application to be dumped. This can take a large amount of storage, and a proportionately long load time. It is also troublesome to apply this technique to constant tables that are to be mapped to read-only memory; if the compiler generates elaboration code to initialize such tables, writable memory must be provided in the development target during the elaboration, and replaced by read-only memory after the core-dump has been produced; the core-dump must then be edited to separate out the portions that need to be programmed into read-only memory from those that are loaded in the normal way. This technique presumes the existence of an external reload device, which might not be available on smaller real-time embedded systems. Finally, effective use of this method requires very precise control over elaboration order to ensure that the desired packages, and only those packages, are elaborated prior to the core-dump. Since this order often includes implementation packages, it is not clear that the user can fully control this order.

The Chosen Approach

Preelaboration is controlled by the two pragmas `Pure` and `Preelaborate` as mentioned in 10.3. Many predefined packages are `Pure` such as

```
package Ada.Characters is
   pragma Pure(Characters);
end Ada.Characters;
```

The introduction of pure packages together with shared passive and remote call interface packages (see [RM95 E.2]) for distribution, created the need to talk about packages whose elaboration happens "before any other elaboration". To accommodate this, the concept of a *preelaborable* construct is introduced in [RM95 10.2.1]. (Preelaborability is naturally the property of an entity which allows it to be preelaborated.) Pure packages are always preelaborated, as well as packages to which the pragma `Preelaborate` specifically applies such as

```
package Ada.Characters.Handling is
   pragma Preelaborate(Handling);
   ...
```

The difference between pure packages and any other preelaborated package is that the latter may have "state". In the core, being preelaborated does not necessarily mean "no code is generated for preelaboration", it only means that these library units are preelaborated before any other unit.

The Systems Programming Annex defines additional implementation and documentation requirements to ensure that the elaboration of preelaborated packages does not execute any code at all.

Issues Related to Preelaboration

Given this approach, some trade-offs had to be made between the generality of constructs to which this requirement applies, the degree of reduction in run-time elaboration code, the complexity of the compiler, and the degree to which the concerns of the run-time system can be separated from those of the compiler.

Bodies of subprograms that are declared in preelaborated packages are guaranteed to be elaborated before they can be called. Therefore, implementations are required to suppress the elaboration checks for such subprograms. This eliminates a source of a distributed overhead that has been an issue in Ada 83.

Tasks, as well as allocators for other than access-to-constant types, are not included among the things specified as preelaborable, because the initialization of run-time system data structures for tasks and the dynamic allocation of storage for general access types would ordinarily require code to be executed at run time. While it might be technically possible to preelaborate tasks and general allocators under sufficiently restrictive conditions, this is considered too difficult to be required of every implementation and would make the compiler very dependent on details of the run-time system. The latter is generally considered to be undesirable by real-time systems developers, who often express the need to customize the Ada run-time environment. It is also considered undesirable by compiler vendors, since it aggravates their configuration management and maintenance problem. (Partial preelaboration of tasks might be more practical for the simple tasking model, described in [RM95 D.7].)

Entryless protected objects are preelaborable and are essential for shared passive packages. They are therefore allowed in preelaborated packages. The initialization of run time data structures might require run-time system calls in some implementations. In particular, where protected object locking is implemented using primitives of an operating system, it might be necessary to perform a system call to create and initialize a lock for the protected object. On such systems, the system call for lock initialization could be postponed until the first operation that is performed on the protected object, but this means some overhead on every protected object operation (perhaps a load, compare, and conditional jump, or an indirect call from a dispatching table). It seems that this kind of distributed overhead on operations that are intended to be very efficient is too high a price to pay for requiring preelaboration of protected objects. These implementations can conform to the requirements in this Annex by doing all initializations "behind-the-scene" before the program actually starts. On most other systems, it is expected that protected objects will be allocated and initialized statically and thus be elaborated when the program starts. Thus, the difference between these two cases is not semantic, and can be left to metrics and documentation requirements.

C.5 Shared Variable Control

Objects in shared memory may be used to communicate data between Ada tasks, between an Ada program and concurrent non-Ada software processes, or between an Ada program and hardware devices.

Ada 83 provided a limited facility for supporting variables shared between otherwise unsynchronized tasks. The pragma `Shared` indicated that a particular elementary object is being concurrently manipulated by two or more tasks, and that all loads and stores should be indivisible. The pragma `Shared` was quite weak. The semantics were only defined in terms of tasks, and not very clearly. This made it inadequate for communication with non-Ada software or hardware devices. Moreover, it could be applied only to a limited set of objects. For example, it could not be applied to a component of an array. One of the most common requirements for shared data access is for buffers, which are typically implemented as arrays. For these reasons, the pragma `Shared` was removed from the language and replaced by the pragmas `Atomic` and `Volatile`.

This Annex thus generalizes the capability to allow data sharing between non-Ada programs and hardware devices, and the sharing of composite objects. In fact, two levels of sharability are introduced:

atomic This indicates that all loads and stores of the object should be indivisible, and that no local copies of the object may be retained. It need only be supported for types where the underlying hardware memory access allows indivisible load and store operations. This imposes requirements on both the size and the alignment of the object.

volatile This indicates that the object can be updated asynchronously. However, there is no need for indivisible load and store.

So we can write

```
type Data is new Long_Float;
pragma Atomic(Data);       -- applying to a type
...
I: Integer;
pragma Volatile(I);        -- applying to a single object
```

Atomic types and objects are implicitly volatile as well. This is because it would make little sense to have an operation applied to an atomic object while allowing the object itself not to be flushed to memory immediately afterwards.

Since the atomicity of an object might affect its layout, it is illegal to explicitly specify other aspects of the object layout in a conflicting manner.

These pragmas may be applied to a constant, but only if the constant is imported. In Ada, the constant designation does not necessarily mean that the object's value cannot change, but rather that it is read-only. Therefore, it seems useful to allow an object to be read-only, while its value changes from the "outside". The rules about volatile objects ensure that the Ada code will read a fresh value each time.

The run-time semantics of atomic/volatile objects are defined in terms of external effects since this is the only way one can talk formally about objects being flushed to or refreshed from memory (as is required to support such objects).

When using such pragmas, one would not want to have the effect of the pragma more general than is minimally needed (this avoids the unnecessary overhead of atomic or load/store operations). That is why separate forms of the pragmas exist for arrays. Writing

```
Buffer: array (1 .. Max) of Integer;
pragma Atomic_Components(Buffer);
```

indicates that the atomicity applies individually to each component of the array but not to the array `Buffer` as a whole.

These pragmas must provide all the necessary information for the compiler to generate the appropriate code each time atomic or volatile objects are accessed. This is why the indication is usually on the type declaration rather than on the object declaration itself. For a stand-alone object there is no problem for the designation to be per-object, but for an array object whose components are atomic or volatile complications would arise. Specifying array components as atomic or volatile is likely to have implications on the layout of the array objects (e.g. components have to be on word boundaries). In addition, if the type of a formal parameter does not have volatile components and the actual parameter does, one would have to pass the parameter by copy, which is generally undesirable. Anonymous array types (as in the example above) do not present this problem since they cannot be passed as parameters directly; explicit conversion is always required, and it is not unreasonable to presume that a copy is involved in an explicit conversion.

The rules for parameter passing distinguish among several cases according to the type; some must be passed by copy, some must be passed by reference and in some cases either is permitted. The last possibility presents a problem for atomic and volatile parameters. To solve this, the rules in this section make them by reference if the parameter (or a component) is atomic or volatile. Moreover, if the actual is atomic or volatile and the formal is not then the parameter is always passed by copy; this may require a local copy of the actual to be made at the site of the call.

The following example shows the use of these pragmas on the components of a record for doing memory-mapped I/O

```
type IO_Rec_Type is
   record
      Start_Address: System.Address;
      pragma Volatile(Start_Address);
      Length: Integer;
      pragma Volatile(Length);
      Operation: Operation_Type;
      pragma Atomic(Operation);
      Reset: Boolean;
      pragma Atomic(Reset);
   end record;

-- A store into the Operation field triggers an I/O operation.
-- Reading the Reset field terminates the current operation.

IO_Rec: IO_Rec_Type;

for IO_Rec'Address use ... ;
```

By using the pragmas to indicate the sharability of data, the semantics of reading and writing components can be controlled. By declaring `Operation` and `Reset` to be atomic, the user ensures that reads and writes of these fields are not removed by optimization, and are performed indivisibly. By declaring `Start_Address` and `Length` to be volatile, the user forces any store to happen immediately, without the use of local copies.

Other Alternatives

Another concept considered was a subtype modifier which would appear in a subtype indication, an object declaration, or parameter specification. However, for simplicity, pragmas were chosen instead.

Other possibilities included an "independent" indication. This would mark the object as being used to communicate between synchronized tasks. Furthermore, the object should be allocated in storage so that loads and stores to it may be performed independently of neighboring objects. Since in Ada 83, all objects were implicitly assumed as independent, such a change would have created upward-compatibility problems. For this reason, and for the sake of simplicity, this feature was rejected.

C.6 Task Identification and Attributes

In order to permit the user to define task scheduling algorithms and to write server tasks that accept requests in an order different from entry service order, it is necessary to introduce a type which identifies a general task (not just of a particular task type) plus some basic operations. This `Task_ID` type is used also by other language-defined packages to operate on task objects such as `Dynamic_Priorities` and `Asynchronous_Task_Control`. In addition, a common need is to be able to associate user-defined properties with all tasks on a per-task basis; this is done through task attributes.

C.6.1 The Package Task_Identification

The `Task_ID` type allows the user to refer to task objects using a copyable type. This is often necessary when one wants to build tables of tasks with associated information. Using access-to-task types is not always suitable since there is no way to define an access-to-any-task type. Task types differ mostly in the entry declarations. The common use of task-id's does not require this entry information, since no rendezvous is performed using objects of `Task_ID`. Instead, the more generic information about the object as being a task is all that is needed. Several constructs are provided to create objects of the `Task_ID` type. These are the `Current_Task` function to query the task-id of the currently executing task; the `Caller` attribute for the task-id of entry callers; and the `Identity` attribute for the task-id of any task object. It is believed that together these mechanisms provide the necessary functionality for obtaining task-id values in various circumstances. In addition, the package provides various rudimentary operations on the `Task_ID` type.

Thus using the example from 9.6, the user might write

```
Joes_ID: Task_ID := Joe'Identity;
...
Set_Priority(Urgent, Joes_ID);  -- see D.5
...
Abort_Task(Joes_ID);            -- same as abort Joe;
```

Another use might be for remembering a caller in one rendezvous and then recognizing the caller again in a later rendezvous, thus

```
task body Some_Service is
   Last_Caller: Task_ID;
begin
   ...
   accept Hello do
      Last_Caller := Hello'Caller;
```

```
            ...
        end Hello;
        ...
        accept Goodbye do
            if Last_Caller /= Goodbye'Caller then
                raise Go_Away;  -- propagate exception to caller
            end if;
            ...
        exception
            when Go_Away => null;
        end Goodbye;
        ...
    end Some_Service;
```

Since objects of Task_ID no longer have the corresponding type and scope information, the possibility for dangling references exist (since Task_ID objects are nonlimited, the value of such an object might contain a task-id of a task that no longer exists). This is particularly so since a server is likely to be at a library level, while the managed tasks might be at a deeper level with a shorter life-time. Operating on such values (here and in other language-defined packages) is defined to be erroneous. Originally, the possibility of requiring scope checking on these values was considered. Such a requirement would impose certain execution overhead on operations and space overhead on the objects of such a type. Since this capability is mainly designed for low-level programming, such an overhead was considered unacceptable. (Note, however, that nothing prevents an implementation from implementing a Task_ID as a record object containing a generation number and thereby providing a higher degree of integrity.)

The Task_ID type is defined as private to promote portability and to allow for flexibility of implementation (such as with a high degree of integrity). The possibility of having a visible (implementation-defined) type was considered. The main reason for this was to allow values of the type to be used as indices in user-defined arrays or as discriminants. To make this usable, the type would have to be discrete. However a discrete type would not allow for schemes that use generation numbers (some sort of a record structure would then be required as mentioned above). A visible type would also reduce portability. So, in the end, a private type approach was chosen. As always, implementations can provide a child package to add hashing functions on Task_ID values, if indexing seems to be an important capability.

Other Alternatives

In an earlier version of Ada 9X, the Task_ID type was defined as the root type of the task class. Since that class was limited, a language-defined access type was also defined to accommodate the need for a copyable type. This definition relied on the existence of untagged class types which were later removed from the language. The approach provided a nice encapsulation of the natural properties of such a type. The general rules of derivations and scope checking could then be fitted directly into the needs of the Task_ID type. Since the underlying model no longer exists, the simpler and more direct approach of a private type with specialized semantics and operation, was adopted.

Another possibility that was considered was to define a special, language-defined, access-to-task type which, unlike other access types, would be required to hold enough information to ensure safe access to dereferenced task objects. This type could then be used as the task-id. Values of such a type would necessarily be larger. This was rejected on the grounds that supporting this special access type in the compiler would be more burdensome to implementations.

Obtaining the Task Identity

The `Current_Task` function is needed in order to obtain the identity of the currently executing task when the name of this task is not known from the context alone; for example when the identity of the environment task is needed or in general service routines that are used by many different tasks.

There are two situations in which it is not meaningful to speak of the "currently executing task". One is within an interrupt handler, which may be invoked directly by the hardware. The other is within an entry body, which may be executed by one task on behalf of a call from another task (the `Caller` attribute may be used in the latter case, instead). For efficiency, the implementation is not required to preserve the illusion of there being an interrupt handler task, or of each execution of an entry body being done by the task that makes the call. Instead, calling `Current_Task` in these situations is defined to be a bounded error.

The values that may be returned if the error is not detected are based on the assumption that the implementation ordinarily keeps track of the currently executing task, but might not take the time to update this information when an interrupt handler is invoked or a task executes an entry body on behalf of a call from another task. In this model, the value returned by `Current_Task` would identify the last value of "current task" recorded by the implementation. In the case of an interrupt handler, this might be the task that the interrupt handler preempted, or an implementation task that executes when the processor would otherwise be idle.

If `Current_Task` could return a value that identifies an implementation task, it might be unsafe to allow a user to abort it or change its priority. However, the likelihood of this occurring is too small to justify checking, especially in an implementation that is sufficiently concerned with efficiency not to have caught the error in the first place. The documentation requirements provide a way for the user to find out whether this possibility exists.

Conversion from an object of any task type to a `Task_ID` is provided by the `Identity` attribute. This conversion is always safe. Support for conversion in the opposite direction is intentionally omitted. Such a conversion is rarely useful since `Task_ID` is normally used when the specific type of the task object is not known, and would be extremely error-prone; (a value of one task type could then be used as another type with all the dangerous consequences of different entries and masters).

`Caller` and `Identity` are defined as attributes and not as functions. For `Caller`, an attribute allows for an easier compile-time detection of an incorrect placement of the construct. For `Identity`, a function would require a formal parameter of a universal task type which does not exist.

The `Caller` attribute is allowed to apply to enclosing accept bodies (not necessarily the innermost one) since it seems quite useful without introducing additional run-time overhead.

Documentation Requirements

In some implementations, the result of calling `Current_Task` from an interrupt handler might be meaningful. Non-portable applications may be able to make use of this information.

C.6.2 The Package Task_Attributes

The ability to have data for which there is a copy for each task in the system is useful for providing various general services. This was provided for example in RTL/2 [Barnes 76] as SVC data.

For Ada 95, several alternatives were considered for the association of user-defined information with each task in a program.

The approach which was finally selected is to have a language-defined generic package whose formal type is the type of an attribute object. This mechanism allows for multiple attributes to be associated with a task using the associated `Task_ID`. These attributes may be dynamically defined, but they cannot be "destroyed". A new attribute is created through instantiation of the generic package.

Thus if we wished to associate some integer token with every task in a program we could write

```
package Token is
   new Ada.Task_Attributes(Attribute => Integer, Initial_Value => 0);
```

and then give the task `Joe` its particular token by

```
Token.Set_Value(99, Joes_ID);
```

Note that the various operations refer to the current task by default, so that

```
Token.Set_Value(101);
```

sets the token of the task currently executing the statement.

After being defined, an object of that attribute exists for each current and newly created task and will be initialized with a user-provided value. Internally the hidden object will typically be derived from the type `Finalization.Controlled` so that finalization of the attribute objects can be performed. When a task terminates, all of its attribute objects are finalized. Note that the attribute objects themselves are allocated in the RTS space, and are not directly accessible by the user program. This avoids problems with dangling references. Since the object is not in user space, it cannot live longer than the task that has it as an attribute. Similarly, this object cannot be deallocated or finalized while the run-time system data structures still point to it.

Obviously, other unrelated user objects might still contain references to attribute objects after they have gone (as part of task termination). This can only happen when one dereferences the access value returned by the `Reference` function since the other operations return (or store) a copy of the attribute. Such dereference (after the corresponding task has terminated) is therefore defined as erroneous. Note that one does not have to wait until a master is left for this situation to arise; referencing an attribute of a terminated task is equally problematic. In general, the `Reference` function is intended to be used "locally" by the task owning the attribute. When the actual attribute object is large, it is sometimes useful to avoid the frequent copying of its value; instead a pointer to the object is obtained and the data is read or written in the user code. When the `Reference` function is used for tasks other then the calling task, the safe practice should be to ensure, by other means, that the corresponding task is not yet terminated.

The generic package `Task_Attributes` can be instantiated locally in a scope deeper than the library level. The effect of such an instantiation is to define a new attribute (for all tasks) for the lifetime of that scope. When the scope is left, the corresponding attribute no longer exists and cannot be referenced anymore. An implementation may therefore deallocate all such attribute objects when that scope is left.

For the implementation of this feature, several options exist. The simplest approach is of a single pointer in the task control block (TCB) to a table containing pointers to the actual attributes, which in turn are allocated from a heap. This table can be preallocated with some initial size. When new attributes are added and the table space is exhausted, a larger one can be allocated with the contents of the old one being copied to the new one. The index of this table can serve as the attribute-id. Each instantiated attribute will have its own data structure (one per partition, not per task), which will contain some type information (for finalization) and the initial value. The attribute-id in the TCB can then point to this attribute type information. Instead of having this level of indirection, the pointer in the TCB can point to a linked list of attributes which then can be dynamically extended or shrunk. Several optimizations on this general scheme are possible. One can preallocate the initial table in the TCB itself, or store in it a fixed number of single-word attributes (presumably, a common case). Since implementations are allowed to place restrictions on the maximum number of attributes and their sizes, static allocation of attribute space is possible, when the application demands more deterministic behavior at run time. Finally, attributes that have not been set yet, or have been reinitialized, do not have to occupy space at all. A flag indicating this state is sufficient for the Value routine to retrieve the initial value from the attribute type area instead of requiring per-task replication of the same value.

Other Approaches

A number of other approaches were considered. One was the idea of a per-task data area. Library-level packages could have been characterized by a pragma as Per_Task, meaning that a fresh copy of the data area of such a package would be available for each newly created task.

Several problems existed with this approach. These problems related to both the implementation and the usability aspects. A task could only access its own data, not the data of any other task. There were also serious problems concerning which packages actually constituted the per-task area.

Another approach considered was a singe attribute per task. Operations to set and retrieve the attribute value of any task were included in addition to finalization rules for when the task terminates. This approach was quite attractive. It was simple to understand and could be implemented with minimal overhead. Larger data structures could be defined by the user and anchored at the single pointer attribute. Operations on these attributes were defined as atomic to avoid race conditions. But the biggest disadvantage of this approach, which eventually led to its rejection, was that such a mechanism does not compose very well and is thus difficult to use in the general case.

C.7 Requirements Summary

The requirements

> *R6.3-A(1) — Interrupt Servicing*

> *R6.3-A(2) — Interrupt Binding*

are met by the pragmas Interrupt_Handler and Attach_Handler plus the package Ada.Interrupts.

The requirement

> *R7.1-A(1) — Control of Shared Memory*

is met by the pragmas Atomic and Volatile discussed in C.5.