GNAT Reference Manual

GNAT, The GNU Ada 95 Compiler
Version 3.11p
$Revision: 1.58 $
August 28, 1998

Ada Core Technologies, Inc.

About This Guide

This manual contains useful information in writing programs using the GNAT compiler. It includes information on implementation dependent characteristics of GNAT, including all the information required by Annex M of the standard.

Ada 95 is designed to be highly portable,and guarantees that, for most programs, Ada 95 compilers behave in exactly the same manner on different machines. However, since Ada 95 is designed to be used in a wide variety of applications, it also contains a number of system dependent features to be used in interfacing to the external world.

Note: Any program that makes use of implementation-dependent features may be non-portable. You should follow good programming practice and isolate and clearly document any sections of your program that make use of these features in a non-portable manner.

What This Reference Manual Contains

This reference manual contains the following chapters:

- Chapter 1 [Implementation Defined Pragmas], page 3 lists GNAT implementation-dependent pragmas, which can be used to extend and enhance the functionality of the compiler.
- Chapter 2 [Implementation Defined Attributes], page 29 lists GNAT implementation-dependent attributes which can be used to extend and enhance the functionality of the compiler.
- Chapter 3 [Implementation Advice], page 37 provides information on generally desirable behavior which are not requirements that all compilers must follow since it cannot be provided on all systems, or which may be undesirable on some systems.
- Chapter 4 [Implementation Defined Characteristics], page 61 provides a guide to minimizing implementation dependent features.
- Chapter 5 [Standard Library Routines], page 81 provides a listing of packages and a brief description of the functionality that is provided by Ada's extensive set of standard library routines as implemented by GNAT.
- Chapter 6 [The Implementation of Standard I/O], page 89 details how the GNAT implementation of the input-output facilities.
- Chapter 7 [Interfacing to Other Languages], page 103 describes how programs written in Ada using GNAT can be interfaced to other programming languages.
- Chapter 9 [Specialized Needs Annexes], page 111 describes the GNAT implementation of all of the special needs annexes.
- Chapter 10 [Compatibility Guide], page 113 includes sections on compatibility of GNAT with other Ada 83 and Ada 95 compilation systems, to assist in porting code from other environments.

This reference manual assumes that you are familiar with Ada 95 language, as described in the International Standard ANSI/ISO/IEC-8652:1995, Jan 1995.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Functions`, `utility program names`, `standard names`, and `classes`.
- '`Option flags`'
- '`File Names`', '`button names`', and '`field names`'.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

  ```
  and then shown this way.
  ```

Commands that are entered by the user are preceded in this manual by the characters "`$ `" (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the $ replaced by whatever prompt character you are using.

Related Information

See the following documents for further information on GNAT

- *GNAT User's Guide*, which provides information on how to use the GNAT compiler system.
- *Ada 95 Reference Manual*, which contains all reference material for the Ada 95 programming language.
- *Ada 95 Annotated Reference Manual*, which is an annotated version of the standard reference manual cited above. The annotations describe detailed aspects of the design decision, and in particular contain useful sections on Ada 83 compatibility.
- *DEC Ada, Technical Overview and Comparison on DIGITAL Platforms*, which contains specific information on compatibility between GNAT and DEC Ada 83 systems.
- *DEC Ada, Language Reference Manual, part number AA-PYZAB-TK* which describes in detail the pragmas and attributes provided by the DEC Ada 83 compiler system.
-

1 Implementation Defined Pragmas

Ada 95 defines a set of pragmas that can be used to supply additional information to the compiler. These language defined pragmas are implemented in GNAT and work as described in the Ada 95 Reference Manual.

In addition, Ada 95 allows implementations to define additional pragmas whose meaning is defined by the implementation. GNAT provides a number of these implementation-dependent pragmas which can be used to extend and enhance the functionality of the compiler. This section of the GNAT Reference Manual describes these additional pragmas.

Note that any program using these pragmas may not be portable to other compilers (although GNAT implements this set of pragmas on all platforms). Therefore if portability to other compilers is an important consideration, the use of these pragmas should be minimized.

`pragma Abort_Defer`

Syntax:

```
pragma Abort_Defer;
```

This pragma must appear at the start of the statement sequence of a handled sequence of statements (right after the `begin`). It has the effect of deferring aborts for the sequence of statements (but not for the declarations or handlers, if any, associated with this statement sequence).

`pragma Ada_83`

Syntax:

```
pragma Ada_83;
```

A configuration pragma that establishes Ada 83 mode for the unit to which it applies, regardless of the mode set by the command line switches. In Ada 83 mode, GNAT attempts to be as compatible with the syntax and semantics of Ada 83, as defined in the original Ada 83 Reference Manual as possible. In particular, the new Ada 95 keywords are not recognized, optional package bodies are allowed, and generics may name types with unknown discriminants without using the (`<>`) notation. In addition, some but not all of the additional restrictions of Ada 83 are enforced.

Ada 83 mode is intended for two purposes. Firstly, it allows existing legacy Ada 83 code to be compiled and adapted to GNAT with less effort. Secondly, it aids in keeping code backwards compatible with Ada 83. However, there is no guarantee that code that is processed correctly by GNAT in Ada 83 mode will in fact compile and execute with an Ada 83 compiler, since GNAT does not enforce all the additional checks required by Ada 83.

`pragma Ada_95`

Syntax:

```
pragma Ada_95;
```

A configuration pragma that establishes Ada 95 mode for the unit to which it applies, regardless of the mode set by the command line switches. This mode is set automatically for the `Ada` and `System` packages and their children, so you need not specify it in these contexts. This pragma is useful when writing a reusable component that itself uses Ada 95 features, but which is intended to be usable from either Ada 83 or Ada 95 programs.

`pragma Annotate`

Syntax:

```
pragma Annotate (IDENTIFIER {, ARG});

ARG ::= NAME | EXPRESSION
```

This pragma is used to annotate programs. *identifier* identifies the type of annotation. GNAT verifies this is an identifier, but does not otherwise analyze it. The *arg* argyment can be either a string literal or an expression. String literals are assumed to be of type `Standard.String`. Names of entities are simply analyzed as entity names. All other expressions are analyzed as expressions, and must be unambiguous.

The analyzed pragma is retained in the tree, but not otherwise processed by any part of the GNAT compiler. This pragma is intended for use by external tools, including ASIS.

`pragma Assert`

Syntax:

```
pragma Assert (
  boolean_EXPRESSION
  [, static_string_EXPRESSION])
```

The effect of this pragma depends on whether the corresponding command line switch is set to activate assertions. If assertions are inactive, the pragma has no effect. If assertions are enabled, then the semantics of the pragma is exactly equivalent to:

```
if not Boolean_EXPRESSION then
   System.Assertions.Raise_Assert_Failure (string_EXPRESSION);
end if;
```

The effect of the call is to raise `System.Assertions.Assert_Failure`. The string argument, if given, is the message associated with the exception occurrence. If no second argument is given, the default message is '*file*:*nnn*', where *file* is the name of the source file containing the assert, and *nnn* is the line number of the assert. A pragma is not a statement, so if a statement sequence contains nothing but a pragma assert, then a null statement is required in addition, as in:

```
...
if J > 3 then
   pragma (Assert (K > 3, "Bad value for K"));
   null;
end if;
```

If the boolean expression has side effects, these side effects will turn on and off with the setting of the assertions mode, resulting in assertions that have an effect on the program. You should generally avoid side effects in the expression of this pragma.

`pragma Ast_Entry`

Syntax:

```
pragma AST_Entry (entry_IDENTIFIER);
```

This pragma is implemented only in the OpenVMS implementation of GNAT. The argument is the simple name of a single entry; at most one `AST_Entry` pragma is allowed for any given entry. This pragma must be used in conjunction with the `AST_Entry` attribute, and is only allowed after the entry declaration and in the same task type specification or single task as the entry to which it applies. This pragma specifies that the given entry may be used to handle an OpenVMS asynchronous system trap (`AST`) resulting from an OpenVMS system service call. The pragma does not affect normal use of the entry. For further details on this pragma, see the DEC Ada Language Reference Manual, section 9.12a.

`pragma C_Pass_By_Copy`

Syntax:

```
pragma C_Pass_By_Copy
  ([Max_Size =>] static_integer_EXPRESSION);
```

Normally the default mechanism for passing C convention records to C convention subprograms is to pass them by reference, as suggested by RM B.3(69). Use the configuration pragma `C_Pass_By_Copy` to change this default, by requiring that record formal parameters be passed by copy if all of the following conditions are met:

- The size of the record type does not exceed *static_integer_expression*.
- The record type has `Convention C`.
- The formal parameter has this record type, and the subprogram has a foreign (non-Ada) convention.

If these conditions are met the argument is passed by copy, i.e. in a manner consistent with what C expects if the corresponding formal in the C prototype is a struct (rather than a pointer to a struct).

You can also pass records by copy by specifying the convention `C_Pass_By_Copy` for the record type, or by using the extended `Import` and `Export` pragmas, which allow specification of passing mechanisms on a parameter by parameter basis.

`pragma Common_Object`

Syntax:

```
pragma Common_Object
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL,
  [, [Size     =>] EXTERNAL_SYMBOL]

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma enables the shared use of variables stored in overlaid linker areas corresponding to the use of `COMMON` in Fortran. The single object *local_name* is assigned to the area designated by the *External* argument. You may define a record to correspond to a series of fields. The *size* argument is syntax checked in GNAT, but otherwise ignored.

`pragma Complex_Representation`

Syntax:

```
pragma Complex_Representation ([Entity =>] LOCAL_NAME);
```

The *Entity* argument must be the name of a record type which has two fields of the same floating-point type. The effect of this pragma is to force gcc to use the special internal complex representation form for this record, which may be more efficient. Note that this may result in the code for this type not conforming to standard ABI (application binary interface) requirements for the handling of record types. For example, in some environments, there is a requirement for passing records by pointer, and the use of this pragma may result in passing this type in floating-point registers.

`pragma Component_Alignment`

Syntax:

```
pragma Component_Alignment (
     [Form =>] ALIGNMENT_CHOICE
  [, [Name =>] type_LOCAL_NAME]);

ALIGNMENT_CHOICE ::=
  Component_Size
| Component_Size_4
| Storage_Unit
| Default
```

Specifies the alignment of components in array or record types. The meaning of the *Form* argument is as follows:

`Component_Size`

Aligns scalar components and subcomponents of the array or record type on boundaries appropriate to their inherent size (naturally aligned). For example, 1-byte components are aligned on byte boundaries, 2-byte integer components are aligned on 2-byte boundaries, 4-byte integer components are aligned on 4-byte boundaries and so on. These alignment rules correspond to the normal rules for C compilers on all machines except the VAX.

`Component_Size_4`

Naturally aligns components with a size of four or fewer bytes. Components that are larger than 4 bytes are placed on the next 4-byte boundary.

`Storage_Unit`
: Specifies that array or record components are byte aligned, i.e. aligned on boundaries determined by the value of the constant `System.Storage_Unit`.

`Default`
: Specifies that array or record components are aligned on default boundaries, appropriate to the nderlying hardware or operating system or both. For OpenVMS VAX systems, the `Default` choice is the same as the `Storage_Unit` choice (byte alignment). For all other systems, the `Default` choice is the same as `Component_Size` (natural alignment).

If the `Name` parameter is present, *type_local_name* must refer to a local record or array type, and the specified alignment choice applies to the specified type. The use of `Component_Alignment` together with a pragma `Pack` causes the `Component_Alignment` pragma to be ignored. The use of `Component_Alignment` together with a record representation clause is only effective for fields not specified by the representation clause.

If the `Name` parameter is absent, the pragma can be used as either a configuration pragma, in which case it applies to one or more units in accordance with the normal rules for configuration pragmas, or it can be used within a declarative part, in which case it applies to types that are declared within this declarative part, or within any nested scope within this declarative part. In either case it specifies the alignment to be applied to any record or array type which has otherwise standard representation.

If the alignment for a record or array type is not specified (using pragma `Pack`, pragma `Component_Alignment`, or a record rep clause), the GNAT uses the default alignment as described previously.

`pragma CPP_Class`

Syntax:

```
pragma CPP_Class ([Entity =>] LOCAL_NAME);
```

The argument denotes an entity in the current declarative region that is declared as a tagged or untagged record type. It indicates that the type corresponds to an externally declared C++ class type, and is to be laid out the same way that C++ would lay out the type.

If (and only if) the type is tagged, at least one component in the record must be of type `Interfaces.CPP.Vtable_Ptr`, corresponding to the C++ Vtable (or Vtables in the case of multiple inheritance) used for dispatching.

Types for which `CPP_Class` is specified do not have assignment or equality operators defined (such operations can be imported or declared as subprograms as required). Initialization is allowed only by constructor functions (see pragma `CPP_Constructor`).

Pragma `CPP_Class` is usually generated automatically using the C++ binding generator tool; See Section 7.2 [Interfacing to C++], page 104 for more details.

`pragma CPP_Constructor`

Syntax:

```
pragma CPP_Constructor ([Entity =>] LOCAL_NAME);
```

This pragma identifies an imported function (imported in the usual way with pragma Import) as corresponding to a C++ constructor. The argument is a name that must have been previously mentioned in a pragma Import with *Convention CPP*, and must be of one of the following forms:

- `function` *Fname* `return` *T*`'Class`
- `function` *Fname* `(...)` `return` *T*`'Class`

where *T* is a tagged type to which the pragma `CPP_Class` applies.

The first form is the default constructor, used when an object of type *T* is created on the Ada side with no explicit constructor. Other constructors (including the copy constructor, which is simply a special case of the second form in which the one and only argument is of type *T*), can only appear in two contexts:

- On the right side of an initialization of an object of type *T*.
- In an extension aggregate for an object of a type derived from *T*.

Although the constructor is described as a function that returns a value on the Ada side, it is typically a procedure with an extra implicit argument (the object being initialized) at the implementation level. GNAT issues the appropriate call, whatever it is, to get the object properly initialized.

In the case of derived objects, you may use one of two possible forms for declaring and creating an object:

- `New_Object : Derived_T`
- `New_Object : Derived_T := (`*constructor-function-call with ...*`)`

In the first case the default constructor is called and extension fields if any are initialized according to the default initialization expressions in the Ada declaration. In the second case, the given constructor is called and the extension aggregate indicates the explicit values of the extension fields.

If no constructors are imported, it is impossible to create any objects on the Ada side. If no default constructor is imported, only the initialization forms using an explicit call to a constructor are permitted.

Pragma `CPP_Constructor` is usually constructed automatically using the C++ binding generator tool; See Section 7.2 [Interfacing to C++], page 104 for more details.

`pragma CPP_Destructor ([Entity =>] LOCAL_NAME);`

This pragma identifies an imported procedure (imported in the usual way with pragma Import) as corresponding to a C++ destructor. *LOCAL_NAME* must be previously mentioned in a pragma `Import` with `Convention CPP`, and be of the following form:

```
procedure Fname (obj : in out T'Class);
```

where *T* is a tagged type to which pragma `CPP_Class` applies. This procedure will be called automatically on scope exit if any objects of *T* are created on the Ada side.

Pragma `CPP_Destructor` is usually constructed automatically using the C++ binding generator tool; See Section 7.2 [Interfacing to C++], page 104 for more details.

`pragma CPP_Virtual`

Syntax:

```
pragma CPP_Virtual
     [Entity     =>] ENTITY,
  [, [Vtable_Ptr =>] vtable_ENTITY,]
  [, [Position   =>] static_integer_EXPRESSION])
```

This pragma serves the same function as pragma `Import` in that case of a virtual function imported from C++. The *Entity* argument must be a primitive subprogram of a tagged type to which pragma `CPP_Class` applies. The *Vtable_Ptr* argument specifies the Vtable_Ptr component which contains the entry for this virtual function. The *Position* argument is the sequential number counting virtual functions for this Vtable starting at 1.

The `Vtable_Ptr` and `Position` arguments may be omitted if there is one Vtable_Ptr present (single inheritance case) and all virtual functions are imported. In that case the compiler can deduce both these values.

No `External_Name` or `Link_Name` arguments are required for a virtual function, since it is always accessed indirectly via the appropriate Vtable entry.

Pragma `CPP_Virtual` is usually constructed automatically using the C++ binding generator tool; See Section 7.2 [Interfacing to C++], page 104 for more details.

`pragma CPP_Vtable`

Syntax:

```
pragma CPP_Vtable (
  [Entity      =>] ENTITY,
  [Vtable_Ptr  =>] vtable_ENTITY,
  [Entry_Count =>] static_integer_EXPRESSION);
```

Given a record to which the pragma `CPP_Class` applies, this pragma can be specified for each component of type `CPP.Interfaces.Vtable_Ptr`. *Entity* is the tagged type, *Vtable_Ptr* is the record field of type `Vtable_Ptr`, and *Entry_Count* is the number of virtual functions on the C++ side. Not all of these functions need to be imported on the Ada side.

You may omit the `CPP_Vtable` pragma if there is only one `Vtable_Ptr` component in the record and all virtual functions are imported on the Ada side (the default value for the entry count in this case is simply the total number of virtual functions).

Pragma `CPP_Vtable` is usually constructed automatically using the C++ binding generator tool; See Section 7.2 [Interfacing to C++], page 104 for more details.

`pragma Debug`

Syntax:

```
pragma Debug (PROCEDURE_CALL_STATEMENT);
```

If assertions are not enabled on the command line, this pragma has no effect. If asserts are enabled, the semantics of the pragma is exactly equivalent to the procedure call. Pragmas are permitted in sequences of declarations, so you can use pragma `Debug` to intersperse calls to debug procedures in the middle of declarations.

`pragma Eliminate`

Syntax:

```
pragma Eliminate (
    [Unit_Name       =>] IDENTIFIER |
                         SELECTED_COMPONENT
  [,[Entity          =>] IDENTIFIER |
                         SELECTED_COMPONENT |
                         STRING_LITERAL]
  [,[Parameter_Types =>] PARAMETER_TYPES]
  [,[Result_Type     =>] result_SUBTYPE_MARK]);


PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}
```

This pragma indicates that the given entity is unused in a program. The entity may be either a subprogram or a variable. If the entity to be eliminated is a library level subprogram, then only the first argument, specifying the corresponding unit name, is required. If the item is an entity of a library package, then the first argument specifies the unit name, and the second argument specifies the particular entity. If the second argument is in string form, it must correspond to the internal manner in which GNAT stores entity names (see compilation unit Namet in the compiler sources for details). The third and fourth parameters are optionally used to distinguish between overloaded subprograms, in the same manner as is used for pragma *Import_Procedure*.

The effect of the pragma is to allow the compiler to optionally eliminate the code or data associated with the named entity. If the declaration of the entity would have resulted in side effects, these side effects may or may not occur in the resulting program. Any reference to an eliminated entity may cause a compile time error, link time error, or incorrect results at runtime.

The intention of pragma Eliminate is to allow a program to be compiled in a system independent manner, with unused entities eliminated, without the requirement of modifying the source text. Normally the required set of Eliminate pragmas is constructed automatically using the gnatelim tool.

`pragma Export_Exception`

Syntax:

```
pragma Export_Exception (
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL,]
  [, [Form     =>] Ada | VMS]
```

```
  [, [Code     =>] static_integer_EXPRESSION]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma is implemented only in the OpenVMS implementation of GNAT. It causes the specified exception to be propagated outside of the Ada program, so that it can be handled by programs written in other OpenVMS languages. This pragma establishes an external name for an Ada exception and makes the name available to the OpenVMS Linker as a global symbol. For further details on this pragma, see the DEC Ada Language Reference Manual, section 13.9a3.2.

`pragma Export_Function` ...

Syntax:

```
pragma Export_Function (
     [Internal         =>] LOCAL_NAME,
  [, [External         =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types  =>] PARAMETER_TYPES]
  [, [Result_Type      =>] result_SUBTYPE_MARK]
  [, [Mechanism        =>] MECHANISM]
  [, [Result_Mechanism =>] MECHANISM_NAME]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

Use this pragma to make a function externally callable and optionally provide information on mechanisms to be used for passing parameter and result values. We recommend, for the purposes of improving portability, this pragma always be used in conjunction with a separate pragma `Export`, which must precede the pragma `Export_Function`. GNAT does not require a separate pragma `Export`, but if none is present, it assumes `Convention C`. Pragma `Export_Function` (and `Export`, if present) must appear in the same declarative region as the function to which they apply.

internal_name must uniquely designate the function to which the pragma applies. If more than one function name exists of this name in the declarative part you must use the `Parameter_Types` and `Result_Type` parameters is mandatory to achieve the required unique designation. *subtype_ mark*s in these parameters must exactly match the subtypes in the corresponding function specification, using positional notation to match parameters with subtype marks. Passing by descriptor is supported only on the OpenVMS ports of GNAT.

pragma `Export_Object` ...

Syntax:

```
pragma Export_Object
      [Internal =>] LOCAL_NAME,
   [, [External =>] EXTERNAL_SYMBOL]
   [, [Size     =>] EXTERNAL_SYMBOL]

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma designates an object as exported, and apart from the extended rules for exernal symbols, is identical in effect to the use of the normal `Export` pragma applied to an object. You may use a separate Export pragma (and you probably should from the point of view of portability), but it is not required. *Size* is syntax checked, but otherwise ignored by GNAT.

pragma `Export_Procedure` ...

Syntax:

```
pragma Export_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})
```

```
MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

This pragma is identical to `Export_Function` except that it applies to a procedure rather than a function and the parameters `Result_Type` and `Result_Mechanism` are not permitted.

`pragma Export_Valued_Procedure`

Syntax:

```
pragma Export_Valued_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

This pragma is identical to `Export_Procedure` except that the first parameter of *local_name*, which must be present, must be of mode `OUT`, and externally the subprogram is treated as a function with this parameter as the result of the function. GNAT provides for this capability to allow the use of `OUT` and `IN OUT` parameters in interfacing to external functions (which are not permitted in Ada functions).

`pragma Extend_System`

Syntax:

```
pragma Extend_System ([Name =>] IDENTIFIER);
```

This pragma is used to provide backwards compatibility with other implementations that extend the facilities of package `System`. In GNAT, `System` contains only the definitions that are present in the Ada 95 RM. However, other implementations, notably the DEC Ada 83 implementation, provide many extensions to package `System`.

For each such implementation accomodated by this pragma, GNAT provides a package `Aux_`*xxx*, e.g. `Aux_DEC` for the DEC Ada 83 implementation, which provides the required additional definitions. You can use this package in two ways. You can `with` it in the normal way and access entities either by selection or using a `use` clause. In this case no special processing is required.

However, if existing code contains references such as `System.`*xxx* where *xxx* is an entity in the extended definitions provided in package `System`, you may use this pragma to extend visibility in `System` in a non-standard way that provides greater compatibility with the existing code. Pragma `Extend_System` is a configuration pragma whose single argument is the name of the package containing the extended definition (e.g. `Aux_DEC` for the DEC Ada case). A unit compiled under control of this pragma will be processed using special visibility processing that looks in package `System.Aux_`*xxx* where `Aux_`*xxx* is the pragma argument for any entity referenced in package `System`, but not found in package `System`.

`pragma Float_Representation`

Syntax:

```
pragma Float_Representation (FLOAT_REP);

FLOAT_REP ::= VAX_Float | IEEE_Float
```

This pragma is implemented only in the OpenVMS implementation of GNAT. It allows control over the internal representation chosen for the predefined floating point types declared in the packages `Standard` and `System`. For further details on this pragma, see the DEC Ada Language Reference Manual, section 3.5.7a. Note that to use this pragma, the standard runtime libraries must be recompiled. See the description of the `GNAT LIBRARY` command in the OpenVMS version of the GNAT Users Guide for details on the use of this command.

`pragma Ident`

Syntax:

```
pragma Ident (static_string_EXPRESSION);
```

This pragma provides a string identification in the generated object file, if the system supports the concept of this kind of identification string. The maximum permitted length of the string literal is 31 characters. This pragma is allowed only in the outermost declarative part or declarative items of a compilation unit. On OpenVMS systems, the effect of the pragma is identical to the effect of the DEC Ada 83 pragma of the same name.

`pragma Import_Exception`

Syntax:

```
pragma Import_Exception (
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL,]
  [, [Form     =>] Ada | VMS]
  [, [Code     =>] static_integer_EXPRESSION]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma is implemented only in the OpenVMS implementation of GNAT. It allows OpenVMS conditions (for example, from OpenVMS system services or other OpenVMS languages) to be propagated to Ada programs as Ada exceptions. The pragma specifies that the exception associated with an exception declaration in an Ada program be defined externally (in non-Ada code). For further details on this pragma, see the DEC Ada Language Reference Manual, section 13.9a.3.1.

`Import_Function` ...

Syntax:

```
pragma Import_Function (
     [Internal                 =>] LOCAL_NAME,
  [, [External                 =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types          =>] PARAMETER_TYPES]
  [, [Result_Type              =>] SUBTYPE_MARK]
  [, [Mechanism                =>] MECHANISM]
  [, [Result_Mechanism         =>] MECHANISM_NAME]
  [, [First_Optional_Parameter =>] IDENTIFIER]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
```

```
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

This pragma is used in conjunction with a pragma `Import` to specify additional information for an imported function. The pragma `Import` (or equivalent pragma `Interface`) must precede the `Import_Function` pragma and both must appear in the same declarative part as the function specification.

The *Internal_Name* argument must uniquely designate the function to which the pragma applies. If more than one function name exists of this name in the declarative part you must use the `Parameter_Types` and *Result_Type* parameters to achieve the required unique designation. Subtype marks in these parameters must exactly match the subtypes in the corresponding function specification, using positional notation to match parameters with subtype marks.

You may optionally use the *Mechanism* and *Result_Mechanism* parameters to specify passing mechanisms for the parameters and result. If you specify a single mechanism name, it applies to all parameters. Otherwise you may specify a mechanism on a parameter by parameter basis using either positional or named notation. If the mechanism is not specified, the default mechanism is used.

Passing by descriptor is supported only on the to OpenVMS ports of GNAT

`First_Optional_Parameter` applies only to OpenVMS ports of GNAT. It specifies that the designated parameter and all following parameters are optional, meaning that they are not passed at the generated code level (this is distinct from the notion of optional parameters in Ada where the parameters are passed anyway with the designated optional parameters). All optional parameters must be of mode `IN` and have default parameter values that are either known at compile time expressions, or uses of the `'Null_Parameter` attribute.

`pragma Import_Object`

Syntax:

```
pragma Import_Object
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL],
  [, [Size     =>] EXTERNAL_SYMBOL])

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma designates an object as imported, and apart from the extended rules for exernal symbols, is identical in effect to the use of the normal `Import` pragma applied to an object. Unlike the subprogram case, you need not use a separate `Import` pragma, although you may do so (and probably should do so from a portability point of view). *size* is syntax checked, but otherwise ignored by GNAT.

`pragma Import_Procedure`

Syntax:

```
pragma Import_Procedure (
     [Internal                 =>] LOCAL_NAME,
  [, [External                 =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types          =>] PARAMETER_TYPES]
  [, [Mechanism                =>] MECHANISM]
  [, [First_Optional_Parameter =>] IDENTIFIER]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

This pragma is identical to `Import_Function` except that it applies to a procedure rather than a function and the parameters `Result_Type` and `Result_Mechanism` are not permitted.

`pragma Import_Valued_Procedure` ...

Syntax:

```
pragma Import_Valued_Procedure (
     [Internal                 =>] LOCAL_NAME,
  [, [External                 =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types          =>] PARAMETER_TYPES]
  [, [Mechanism                =>] MECHANISM]
  [, [First_Optional_Parameter =>] IDENTIFIER]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| SUBTYPE_MARK {, SUBTYPE_MARK}
```

```
MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
| Descriptor [([Class =>] CLASS_NAME)]

CLASS_NAME ::= ubs | ubsb | uba | s | sb | a | nca
```

This pragma is identical to `Import_Procedure` except that the first parameter of *local_name*, which must be present, must be of mode `OUT`, and externally the subprogram is treated as a function with this parameter as the result of the function. The purpose of this capability is to allow the use of `OUT` and `IN OUT` parameters in interfacing to external functions (which are not permitted in Ada functions). You may optionally use the `Mechanism` parameters to specify passing mechanisms for the parameters. If you specify a single mechanism name, it applies to all parameters. Otherwise you may specify a mechanism on a parameter by parameter basis using either positional or named notation. If the mechanism is not specified, the default mechanism is used.

pragma `Inline_Generic`

Syntax:

```
pragma Inline_Generic (generic_package_NAME)
```

This is implemented for compatibility with DEC Ada 83 and is recognized, but otherwise ignored, by GNAT. All generic instantiations are inlined by default when using GNAT.

pragma `Interface_Name`

Syntax:

```
pragma Interface_Name (
     [Entity        =>] LOCAL_NAME
  [, [External_Name =>] static_string_EXPRESSION]
  [, [Link_Name     =>] static_string_EXPRESSION]);
```

This pragma provides an alternative way of specifying the interface name for an interfaced subprogram, and is provided for compatibility with Ada 83 compilers that use the pragma for this purpose. You must provide at least one of *External_Name* or *Link_Name*.

pragma `Linker_Alias`

Syntax:

```
pragma Linker_Alias (
  [Entity =>] LOCAL_NAME
  [Alias  =>] static_string_EXPRESSION);
```

This pragma establishes a linker alias for the given named entity. For further details on the exact effect, consult the GCC manual.

`pragma Linker_Section`

Syntax:

```
pragma Linker_Section (
  [Entity  =>] LOCAL_NAME
  [Section =>] static_string_EXPRESSION);
```

This pragma specifies the name of the linker section for the given entity. For further details on the exact effect, consult the GCC manual.

`pragma Normalize_Scalars`

Syntax:

```
pragma Normalize_Scalars;
```

This is a language defined pragma which is fully implemented in GNAT. The effect is to cause all scalar objects that are not otherwise initialized to be initialized. The initial values are implementation dependent and are as follows:

`Standard.Character`
: Objects whose root type is Standard.Character are initialized to Character'Last. This will be out of range of the subtype only if the subtype range excludes this value.

`Standard.Wide_Character`
: Objects whose root type is Standard.Wide_Character are initialized to Wide_Character'Last. This will be out of range of the subtype only if the subtype range excludes this value.

`Integer types`
: Objects of an integer type are initialized to base_type'First, where base_type is the base type of the object type. This will be out of range of the subtype only if the subtype range excludes this value. For example, if you declare the subtype:

```
subtype Ityp is integer range 1 .. 10;
```

then objects of type x will be initialized to Integer'First, a negative number that is certainly outside the range of subtype `Ityp`.

`Real types`
: Objects of all real types (fixed and floating) are initialized to base_type'First, where base_Type is the base type of the object type. This will be out of range of the subtype only if the subtype range excludes this value.

`Modular types`
: Objects of a modular type are initialized to typ'Last. This will be out of range of the subtype only if the subtype excludes this value.

`Enumeration types`
: Objects of an enumeration type are initialized to all one-bits, i.e. to the value 2 ** typ'Size - 1. This will be out of range of the

enumeration subtype in all cases except where the subtype contains exactly 2**8, 2**16, or 2**32.

`pragma Long_Float`

Syntax:

```
pragma Long_Float (FLOAT_FORMAT);

FLOAT_FORMAT ::= D_Float | G_Float
```

This pragma is implemented only in the OpenVMS implementation of GNAT. It allows control over the internal representation chosen for the predefined type `Long_Float` and for floating point type representations with `digits` specified in the range 7 .. 15. For further details on this pragma, see the DEC Ada Language Reference Manual, section 3.5.7b. Note that to use this pragma, the standard runtime libraries must be recompiled. See the description of the `GNAT LIBRARY` command in the OpenVMS version of the GNAT Users Guide for details on the use of this command.

`pragma Machine_Attribute` ...

Syntax:

```
pragma Machine_Attribute (
  [Attribute_Name =>] string_EXPRESSION,
  [Entity         =>] LOCAL_NAME);
```

Machine dependent attributes can be specified for types and/or declarations. Currently only subprogram entities are supported. This pragma is semantically equivalent to `__attribute__((` *string˙expression*`))` in GNU C, where `string_expression>` is recognized by the GNU C macros `VALID_MACHINE_TYPE_ATTRIBUTE` and `VALID_MACHINE_DECL_ATTRIBUTE` which are defined in the configuration header file '`tm.h`' for each machine. See the GCC manual for further information.

`pragma Main_Storage`

Syntax:

```
pragma Main_Storage
  (MAIN_STORAGE_OPTION [, MAIN_STORAGE_OPTION]);

MAIN_STORAGE_OPTION ::=
  [WORKING_STORAGE =>] static_SIMPLE_EXPRESSION
| [TOP_GUARD       =>] static_SIMPLE_EXPRESSION
```

This pragma is provided for compatibility with OpenVMS Vax Systems. It has no effect in GNAT, other than being syntax checked. Note that the pragma also has no effect in DEC Ada 83 for OpenVMS Alpha Systems.

`pragma No_Return`

Syntax:

```
pragma No_Return (procedure_LOCAL_NAME);
```

procedure_local_NAME must refer to one or more procedure declarations in the current declarative part. A procedure to which this pragma is applied may not contain any explicit `return` statements, and also may not contain any implicit return statements from falling off the end of a statement sequence. One use of this pragma is to identify procedures whose only purpose is to raise an exception.

Another use of this pragma is to suppress incorrect warnings about missing returns in functions, where the last statement of a function statement sequence is a call to such a procedure.

`pragma Passive`

Syntax:

```
pragma Passive ([Semaphore | No]);
```

Syntax checked, but otherwise ignored by GNAT. This is recognized for compatibility with DEC Ada 83 implementations, where it is used within a task definition to request that a task be made passive. If the argument `Semaphore` is present, or no argument is omitted, then DEC Ada 83 treats the pragma as an assertion that the containing task is passive and that optimization of context switch with this task is permitted and desired. If the argument `No` is present, the task must not be optimized. GNAT does not attempt to optimize any tasks in this manner (since protected objects are available in place of passive tasks).

`pragma Psect_Object`

Syntax:

```
pragma Psect_Object
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL]
  [, [Size     =>] EXTERNAL_SYMBOL]

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma is identical in effect to pragma `Common_Object`.

`Pure_Function`

Syntax:

```
pragma Pure_Function ([Entity =>] function_LOCAL_NAME);
```

This pragma appears in the same declarative part as a function declaration (or a set of function declarations if more than one overloaded declaration exists, in which case the pragma applies to all entities). If specifies that the function `Entity` is to be considered pure for the purposes of code generation. This means that the compiler can assume that there are no side effects, and in particular that two calls with identical arguments produce the same result. It also means that the function can be used in an address clause.

Note that, quite deliberately, there are no static checks to try to ensure that this promise is met, so *Pure_Function* can be used with functions that are conceptually pure, even if they do modify global variables. For example, a square root function that is instrumented to count the number of times it is called is still conceptually pure, and can still be optimized, even though it modifies a global variable (the count). Memo functions are another example (where a table of previous calls is kept and consulted to avoid recomputation).

Note: All functions in a `Pure` package are automatically pure, and there is no need to use pragma `Pure_Function` in this case.

Note: If pragma `Pure_Function` is applied to a renamed function, it applies to the underlying renamed function. This can be used to disambiguate cases of overloading where some but not all functions in a set of overloaded functions are to be designated as pure.

pragma `Share_Generic`

Syntax:

```
pragma Share_Generic (NAME {, NAME});
```

This pragma is recognized for compatibility with other Ada compilers but is ignored by GNAT. GNAT does not provide the capability for sharing of generic code. All generic instantiations result in making an inlined copy of the template with appropriate substitutions.

pragma `Source_File_Name`

Syntax:

```
pragma Source_File_Name (
  [Unit_Name   =>] unit_NAME,
  [FNAME_DESIG =>] static_string_EXPRESSION);

FNAME_DESIG => Body_File_Name | Spec_File_Name
```

Use this to override the normal naming convention. It is a configuration pragma, and so has the usual applicability of configuration pragmas (i.e. it applies to either an entire partition, or to all units in a compilation, or to a single unit, depending on how it is used. *unit_name* is mapped to *file_name_literal*. The identifier for the second argument is required, and indicates whether this is the file name for the spec or for the body.

pragma `Source_Reference`

Syntax:

```
pragma Source_Reference (INTEGER_LITERAL, STRING_LITERAL);
```

This pragma typically appears as the first line of a source file. *integer_literal* is the logical line number of the line following the pragma line (for use in error messages and debugging information). *string_literal* is a static string constant that specifies the file name to be used in error messages and debugging information. This is most notably used for the output of `gnatchop` with the '`-r`' switch, to make sure that the original unchopped source file is the one referred to.

The second argument must be a string literal, it cannot be a static string expression other than a string literal. This is because its value is needed for error messages issued by all phases of the compiler.

pragma `Stream_Convert`

Syntax:

```
pragma Stream_Convert (
  [Entity =>] type_LOCAL_NAME,
  [Read   =>] function_NAME,
  [Write  =>] function NAME);
```

This pragma provides an efficient way of providing stream functions for types defined in packages. Not only is it simpler to use than declaring the necessary functions with attribute representation clauses, but more significantly, it allows the declaration to made in such a way that the stream packages are not loaded unless they are needed. The use of the Stream_Convert pragma adds no overhead at all, unless the stream attributes are actually used on the designated type.

The first argument specifies the type for which stream functions are provided. The second parameter provides a function used to read values of this type. It must name a function whose argument type may be any subtype, and whose returned type must be the type given as the first argument to the pragma.

The meaning of the *Read* parameter is that if a stream attribute directly or indirectly specifies reading of the type given as the first parameter, then a value of the type given as the argument to the Read function is read from the stream, and then the Read function is used to convert this to the required target type.

Similarly the *Write* parameter specifies how to treat write attributes that directly or indirectly apply to the type given as the first parameter. It must have an input parameter of the type specified by the first parameter, and the return type must be the same as the input type of the Read function. The effect is to first call the Write function to convert to the given stream type, and then write the result type to the stream.

The Read and Write functions must not be overloaded subprograms. If necessary renamings can be supplied to meet this requirement. The usage of this attribute is best illustrated by a simple example, taken from the GNAT implementation of package Ada.Strings.Unbounded:

```
function To_Unbounded (S : String) return Unbounded_String
  renames To_Unbounded_String;

pragma Stream_Convert
  (Unbounded_String, To_Unbounded, To_String);
```

The specifications of the referenced functions, as given in the Ada 95 Reference Manual are:

```
function To_Unbounded_String (Source : String)
  return Unbounded_String;

function To_String (Source : Unbounded_String)
  return String;
```

The effect is that if the value of an unbounded string is written to a stream, then the representation of the item in the stream is in the same format used for `Standard.String`, and this same representation is expected when a value of this type is read from the stream.

pragma Subtitle

Syntax:

```
pragma Subtitle ([Subtitle =>] STRING_LITERAL);
```

This pragma is recognized for compatibility with other Ada compilers but is ignored by GNAT.

pragma Suppress_All

Syntax:

```
pragma Suppress_All;
```

This pragma can only appear immediately following a compilation unit. The effect is to apply `Suppress (All_Checks)` to the unit which it follows. This pragma is implemented for compatibility with DEC Ada 83 usage. The use of pragma `Suppress (All_Checks)` as a normal configuration pragma is the preferred usage in GNAT.

pragma Task_Info

Syntax

```
pragma Task_Info (EXPRESSION);
```

This pragma appears within a task definition (like pragma `Priority`) and applies to the task in which it appears. The argument must be of type `System.Task_Info.Task_Info_Type`. The `Task_Info` pragma provides system dependent control over aspect of tasking implementation, for example, the ability to map tasks to specific processors. For details on the facilities available for the version of GNAT that you are using, see the documentation in the specification of package System.Task_Info in the runtime library.

pragma Task_Storage

Syntax:

```
pragma Task_Storage
  [Task_Type =>] LOCAL_NAME,
  [Top_Guard =>] static_integer_EXPRESSION);
```

This pragma specifies the length of the guard area for tasks. The guard area is an additional storage area allocated to a task. A value of zero means that either no guard area is created or a minimal guard area is created, depending on the target. This pragma can appear anywhere a `Storage_Size` attribute definition clause is allowed for a task type.

pragma Time_Slice

Syntax:

```
pragma Time_Slice (static_duration_EXPRESSION);
```

For implementations of GNAT on operating systems where it is possible to supply a time slice value, this pragma may be used for this purpose. It is ignored if it is used in a system that does not allow this control, or if it appears in other than the main program unit. Note that the effect of this pragma is identical to the effect of the DEC Ada 83 pragma of the same name when operating under OpenVMS systems.

`pragma Title`

Syntax:

```
pragma Title (TITLING_OPTION [, TITLING OPTION]);

TITLING_OPTION ::=
  [Title    =>] STRING_LITERAL,
| [Subtitle =>] STRING_LITERAL
```

Syntax checked but otherwise ignored by GNAT. This is a listing control pragma used in DEC Ada 83 implementations to provide a title and/or subtitle for the program listing. The program listing generated by GNAT does not have titles or subtitles.

Unlike other pragmas, the full flexibility of named notation is allowed for this pragma, i.e. the parameters may be given in any order if named notation is used, and named and positional notation can be mixed following the normal rules for procedure calls in Ada.

`pragma Unchecked_Union`

Syntax:

```
pragma Unchecked_Union (first_subtype_LOCAL_NAME)
```

This pragma is used to declare that the specified type should be represented in a manner equivalent to a C union type, and is intended only for use in interfacing with C code that uses union types. In Ada terms, the named type must obey the following rules:

- It is a non-tagged non-limited record type.
- It has a single discrete discriminant with a default value.
- The component list consists of a single variant part.
- Each variant has a component list with a single component.
- No nested variants are allowed.
- No component has an explicit default value.
- No component has a non-static constraint.

In addition, given a type that meets the above requirements, the following restrictions apply to its use throughout the program:

- The discriminant name can be mentioned only in an aggregate.
- No subtypes may be created of this type.
- The type may not be constrained by giving a discriminant value.
- The type cannot be passed as the actual for a generic formal with a discriminant.

Equality and inequality operations on `unchecked_unions` are not available, since there is no discriminant to compare and the compiler does not even know how many bits to compare. It is implementation dependent whether this is detected at compile time as an illegality or whether it is undetected and considered to be an erroneous construct. In GNAT, a direct comparison is illegal, but GNAT does not attempt to catch the composite case (where two composites are compared that contain an unchecked union component), so such comparisons are simply considered erroneous.

The layout of the resulting type corresponds exactly to a C union, where each branch of the union corresponds to a single variant in the Ada record. The semantics of the Ada program is not changed in any way by the pragma, i.e. provided the above restrictions are followed, and no erroneous incorrect references to fields or erroneous comparisons occur, the semantics is exactly as described by the Ada reference manual. Pragma `Suppress (Discriminant_Check)` applies implicitly to the type and the default convention is C

`pragma Unimplemented_Unit`

Syntax:

```
pragma Unimplemented_Unit;
```

If this pragma occurs in a unit that is processed by the compiler, GNAT aborts with the message '*xxx* `not implemented`', where *xxx* is the name of the current compilation unit. This pragma is intended to allow the compiler to handle unimplemented library units in a clean manner.

The abort only happens if code is being generated. Thus you can use specs of unimplemented packages in syntax or semantic checking mode.

`pragma Unreserve_All_Interrupts`

Syntax:

```
pragma Unreserve_All_Interrupts;
```

Normally certain interrupts are reserved to the implementation. Any attempt to attach an interrupt causes Program_Error to be raised, as described in RM C.3.2(22). A typical example is the `SIGINT` interrupt used in many systems for an `Ctrl-C` interrupt. Normally this interrupt is reserved to the implementation, so that `Ctrl-C` can be used to interrupt execution.

If the pragma Unreserve_All_Interrupts appears anywhere in any unit in a program, then all such interrupts are unreserved. This allows the program to handle these interrupts, but disables their standard functions. For example, if this pragma is used, then pressing `Ctrl-C` will not automatically interrupt execution. However, a program can then handle the `SIGINT` interrupt as it chooses.

For a full list of the interrupts handled in a specific implementation, see the source code for the specification of Ada.Interrupts.Names in file s-intnam.ads. This is a target dependent file that contains the list of interrupts recognized for a given target. The documentation in this file also specifies what interrupts are affected by the use of the Unreserve_All_Interrupts pragma.

`pragma Unsuppress`

Syntax:

```
pragma Unsuppress (IDENTIFIER [, [On =>] NAME]);
```

This pragma undoes the effect of a previous pragma `Suppress`. If there is no corresponding pragma `Suppress` in effect, it has no effect. The range of the effect is the same as for pragma `Suppress`. The meaning of the arguments is identical to that used in pragma `Suppress`.

One important application is to ensure that checks are on in cases where code depends on the checks for its correct functioning, so that the code will compile correctly even if the compiler switches are set to suppress checks.

`pragma Use_VADS_Size`

Syntax:

```
pragma Use_VADS_Size;
```

This is a configuration pragma. In a unit to which it applies, any use of the 'Size attribute is automatically interpreted as a use of the 'VADS_Size attribute. Note that this may result in incorrect semantic processing of valid Ada 95 programs. This is intended to aid in the handling of legacy code which depends on the interpretation of Size as implemented in the VADS compiler. See description of the VADS_Size attribute for further details.

`pragma Volatile`

Syntax:

```
pragma Volatile (local_NAME)
```

This pragma is defined by the Ada 95 Reference Manual, and the GNAT implementation is fully conformant with this definition. The reason it is mentioned in this section is that a pragma of the same name was supplied in some Ada 83 compilers, including DEC Ada 83. The Ada 95 implementation of pragma Volatile is upwards compatible with the implementation in Dec Ada 83.

`pragma Warnings`

Syntax:

```
pragma Warnings (On | Off [, LOCAL_NAME]);
```

Normally warnings are enabled, with the output being controlled by the command line switch. Warnings (`Off`) turns off generation of warnings until a Warnings (`On`) is encountered or the end of the current unit. If generation of warnings is turned off using this pragma, then no warning messages are output, regardless of the setting of the command line switches.

The form with a single argument is a configuration pragma.

If the *local_name* parameter is present, warnings are suppressed for the specified entity. This suppression is effective from the point where it occurs till the end of the extended scope of the variable (similar to the scope of `Suppress`).

`pragma Weak_External`

Syntax:

```
pragma Weak_External ([Entity =>] LOCAL_NAME);
```

This pragma specifies that the given entity should be marked as a weak external (one that does not have to be resolved) for the linker. For further details, consult the GCC manual.

2 Implementation Defined Attributes

Ada 95 defines (throughout the Ada 95 reference manual, summarized in annex K), a set of attributes that provide useful additional functionality in all areas of the language. These language defined attributes are implemented in GNAT and work as described in the Ada 95 Reference Manual.

In addition, Ada 95 allows implementations to define additional attributes whose meaning is defined by the implementation. GNAT provides a number of these implementation-dependent attributes which can be used to extend and enhance the functionality of the compiler. This section of the GNAT reference manual describes these additional attributes.

Note that any program using these attributes may not be portable to other compilers (although GNAT implements this set of attributes on all platforms). Therefore if portability to other compilers is an important consideration, you should minimize the use of these attributes.

`Abort_Signal`

`Standard'Abort_Signal` (`Standard` is the only allowed prefix) provides the entity for the special exception used to signal task abort or asynchronous transfer of control. Normally this attribute should only be used in the tasking runtime (it is highly peculiar, and completely outside the normal semantics of Ada, for a user program to intercept the abort exception).

`Address_Size`

`Standard'Address_Size` (`Standard` is the only allowed prefix) is a static constant giving the number of bits in an `Address`. It is used primarily for constructing the definition of `Memory_Size` in package `Standard`, but may be freely used in user programs.

`AST_Entry`

This attribute is implemented only in OpenVMS versions of GNAT. Applied to the name of an entry, it yields a value of the predefined type AST_Handler (dclared in the predefined package System, as extended by the use of pragma Extend_System (Aux_DEC)). This value enables the given entry to be called when an AST occurs. For further details, refer to the DEC Ada Language Reference Manual, section 9.12a.

`Bit`

obj`'Bit`, where *obj* is any object, yields the bit offset within the storage unit (byte) that contains the first bit of storage allocated for the object. The value of this attribute is of the type `Universal_Integer`, and is always a non-negative number not exceeding the value of `System.Storage_Unit`.

For an object that is a variable or a constant allocated in a register, the value is zero. (The use of this attribute does not force the allocation of a variable to memory).

For an object that is a formal parameter, this attribute applies to either the matching actual parameter or to a copy of the matching actual parameter.

For an access object the value is zero. Note that *obj*`.all'Bit` is subject to an `Access_Check` for the designated object. Similarly for a record component

X.*C*'Bit is subject to a discriminant check and *X*(*I*).Bit and *X*(*I1*..*I2*)'Bit are subject to index checks.

This attribute is designed to be compatible with the DEC Ada 83 definition and implementation of the Bit attribute.

Bit_Position

R.C'Bit, where *R* is a record object and C is one of the fields of the record type, yields the bit offset within the record contains the first bit of storage allocated for the object. The value of this attribute is of the type Universal_Integer. The value depends only on the field *C* and is independent of the alignment of the containing record *R*.

Code_Address

The 'Address attribute may be applied to subprograms in Ada 95, but the intended effect from the Ada 95 reference manual seems to be to provide an address value which can be used to call the subprogram by means of an address clause as in the following example:

```
procedure K is ...

procedure L;
for L'Address use K'Address;
pragma Import (Ada, L);
```

A call to L is then expected to result in a call to K. In Ada 83, where there were no access-to-subprogram values, this was a common work around for getting the effect of an indirect call. GNAT implements the above use of Address and the technique illustrated by the example code works correctly.

However, for some purposes, it is useful to have the address of the start of the generated code for the subprogram. On some architectures, this is not necessarily the same as the Address value described above. For example, the Address value may reference a subprogram descriptor rather than the subprogram itself.

The 'Code_Address attribute, which can only be applied to subprogram entities, always returns the address of the start of the generated code of the specified subprogram, which may or may not be the same value as is returned by the corresponding 'Address attribute.

Default_Bit_Order

Standard'Default_Bit_Order (Standard is the only permissible prefix), provides the value System.Default_Bit_Order as a Pos value (0 for High_Order_First, 1 for Low_Order_First). This is used to construct the definition of Default_Bit_Order in package System.

Elaborated

The prefix of the 'Elaborated attribute must be a unit name. The value is a Boolean which indicates whether or not the given unit has been elaborated. This attribute is primarily intended for internal use by the generated code for dynamic elaboration checking, but it can also be used in user programs. The value will always be True once elaboration of all units has been completed.

`Elab_Body`

This attribute can only be applied to a program unit name. It returns the entity for the corresponding elaboration procedure for elaborating the body of the referenced unit. This is used in the main generated elaboration procedure by the binder and is not normally used in any other context. However, there may be specialized situations in which it is useful to be able to call this elaboration procedure from Ada code, e.g. if it is necessary to do selective re-elaboration to fix some error.

`Elab_Spec`

This attribute can only be applied to a program unit name. It returns the entity for the corresponding elaboration procedure for elaborating the specification of the referenced unit. This is used in the main generated elaboration procedure by the binder and is not normally used in any other context. However, there may be specialized situations in which it is useful to be able to call this elaboration procedure from Ada code, e.g. if it is necessary to do selective re-elaboration to fix some error.

`Enum_Rep`

For every enumeration subtype *S*, *S*`'Enum_Rep` denotes a function with the following specification:

```
function S'Enum_Rep (Arg : S'Base) return Universal_Integer;
```

It is also allowable to apply Enum_Rep directly to an object of an enumeration type or to a non-overloaded enumeration literal. In this case *S*`'Enum_Rep` is equivalent to *typ*`'Enum_Rep(`*S*`)` where *typ* is the type of the enumeration literal or object.

The function returns the representation value for the given enumeration value. This will be equal to value of the `Pos` attribute in the absence of an enumeration representation clause. This is a static attribute (i.e. the result is static if the argument is static).

`Fixed_Value`

For every fixed-point type *S*, *S*`'Fixed_Value` denotes a function with the following specification:

```
function S'Fixed_Value (Arg : Universal_Integer) return S;
```

The value returned is the fixed-point value *V* such that

```
V = Arg * S'Small
```

The effect is thus equivalent to first converting the argument to the integer type used to represent *S*, and then doing an unchecked conversion to the fixed-point type. This attribute is primarily intended for use in implementation of the input-output functions for fixed-point values.

`Has_Discriminants`

The prefix of the `Has_Disctriminants` attribute is a type. The result is a Boolean value which is True if the type has discriminants, and False otherwise. The intended use of this attribute is in conjunction with generic definitions. If the attribute is applied to a generic private type, it indicates whether or not the corresponding actual type has discriminants.

`Img` The `Img` attribute differs from `Image` in that it may be applied to objects as well as types, in which case it gives the `Image` for the subtype of the object. This is convenient for debugging:

```
Put_Line ("X = " & X'Img);
```

has the same meaning as the more verbose:

```
Put_Line ("X = " & type'Image (X));
```

where *type* is the subtype of the object X.

`Integer_Value`
For every integer type *S*, *S*`'Integer_Value` denotes a function with the following specification:

```
function S'Integer_Value (Arg : Universal_Fixed) return S;
```

The value returned is the integer value *V*, such that

```
Arg = V * type'Small
```

The effect is thus equivalent to first doing an unchecked convert from the fixed-point type to its corresponding implementation type, and then converting the result to the target integer type. This attribute is primarily intended for use in implementation of the standard input-output functions for fixed-point values.

`Machine_Size`
This attribute is identical to the `Object_Size` attribute. It is provided for compatibility with the DEC Ada 83 attribute of this name.

`Max_Interrupt_Priority`
`Standard'Max_Interrupt_Priority` (`Standard` is the only permissible prefix), provides the value `System.Max_Interrupt_Priority` and is intended primarily for constructing this definition in package `System`.

`Max_Priority`
`Standard'Max_Priority` (`Standard` is the only permissible prefix) provides the value `System.Max_Priority` and is intended primarily for constructing this definition in package `System`.

`Maximum_Alignment`
`Standard'Maximum_Alignment` (`Standard` is the only permissible prefix) provides the maximum useful alignment value for the target. This is a static value that can be used to specify the alignment for an object, guaranteeing that it is properly aligned in all cases. This is useful when an external object is imported and its alignment requirements are unknown.

`Mechanism_Code`
function`'Mechanism_Code` yields an integer code for the mechanism used for the result of function, and *subprogram*`'Mechanism_Code (`*n*`)` yields the mechanism used for formal parameter number *n* (a static integer value with 1 meaningthe first parameter) of *subprogram*. The code returned is:

1 by copy (value)

2 by reference

3	by descriptor (default descriptor class)
4	by descriptor (UBS: unaligned bit string)
5	by descriptor (UBSB: aligned bit string with arbitrary bounds)
6	by descriptor (UBA: unaligned bit array)
7	by descriptor (S: string, also scalar access type parameter)
8	by descriptor (SB: string with arbitrary bounds)
9	by descriptor (A: contiguous array)
10	by descriptor (NCA: non-contiguous array)

Values from 3-10 are only relevant to Digital OpenVMS implementations.

`Null_Parameter`

A reference *T*'`Null_Parameter` denotes an imaginary object of type or subtype *T* allocated at machine address zero. The attribute is allowed only as the default expression of a formal parameter, or as an actual expression of a subporgram call. In either case, the subprogram must be imported.

The identity of the object is represented by the address zero in the argument list, independent of the passing mechanism (explicit or default).

This capability is needed to specify that a zero address should be passed for a record or other composite object passed by reference. There is no way of indicating this without the `Null_Parameter` attribute.

`Object_Size`

The size of an object is not necessarily the same as the size of the type of an object. This is because by default object sizes are increased to be a multiple of the alignment of the object. For example, `Natural'Size` is 31, but by default objects of type `Natural` will have a size of 32 bits. Similarly, a record containing an integer and a character:

```
type Rec is record
   I : Integer;
   C : Character;
end record;
```

will have a size of 40 (that is `Rec'Size` will be 40. The alignment will be 4, because of the integer field, and so the default size of record objects for this type will be 64 (8 bytes).

The *type*'`Object_Size` attribute has been added to GNAT to allow the default object size of a type to be easily determined. For example, `Natural'Object_Size` is 32, and `Rec'Object_Size` (for the record type in the above example) will be 64. Note also that, unlike the situation with the `Size` attribute as defined in the Ada RM, the `Object_Size` attribute can be specified individually for different subtypes. For example:

```
type R is new Integer;
subtype R1 is R range 1 .. 10;
subtype R2 is R range 1 .. 10;
```

```
for R2'Object_Size use 8;
```

In this example, `R'Object_Size` and `R1'Object_Size` are both 32 since the default object size for a subtype is the same as the object size for the the parent subtype. This means that objects of type `R` or `R1` will by default be 32 bits (four bytes). But objects of type `R2` will be only 8 bits (one byte), since `R2'Object_Size` has been set to 8.

`Passed_By_Reference`
: *type*`'Passed_By_Reference` for any subtype *type* returns a value of type `Boolean` value that is `True` if the type is normally passed by reference and `False` if the type is normally passed by copy in calls. For scalar types, the result is always `False` and is static. For non-scalar types, the result is non-static.

`Range_Length`
: *type*`'Range_Length` for any discrete type *type* yields the number of values represented by the subtype (zero for a null range). The result is static for static subtypes. `Range_Length` applied to the index subtype of a one dimensional array always gives the same result as `Range` applied to the array itself.

`Storage_Unit`
: `Standard'Storage_Unit` (`Standard` is the only permissible prefix) provides the value `System.Storage_Unit` and is intended primarily for constructing this definition in package `System`.

`Tick`
: `Standard'Tick` (`Standard` is the only permissible prefix) provides the value of `System.Tick` and is intended primarily for constructing this definition in package `System`.

`Type_Class`
: *type*`'Type_Class` for any type or subtype *type* yields the value of the type class for the full type of *type*. If *type* is a generic formal type, the value is the value for the corresponding actual subtype. The value of this attribute is of type `System.Aux_DEC.Type_Class`, which has the following definition:

```
type Type_Class is
  (Type_Class_Enumeration,
   Type_Class_Integer,
   Type_Class_Fixed_Point,
   Type_Class_Floating_Point,
   Type_Class_Array,
   Type_Class_Record,
   Type_Class_Access,
   Type_Class_Task,
   Type_Class_Address);
```

Protected types yield the value `Type_Class_Task`, which thus applies to all concurrent types. This attribute is designed to be compatible with the DEC Ada 83 attribute of the same name.

`Universal_Literal_String`
: The prefix of `Universal_Literal_String` must be a named number. The static result is the string consisting of the characters of the number as defined

in the original source. This allows the user program to access the actual text of named numbers without intermediate conversions and without the need to enclose the strings in quotes (which would preclude their use as numbers). This is used internally for the construction of values of the floating-point attributes from the file 'ttypef.ads', but may also be used by user programs.

`Unrestricted_Access`
: The `Unrestricted_Access` attribute is similar to `Access` except that all accessibility and aliased view checks are omitted. This is a user-beware attribute. It is similar to `Address`, for which it is a desirable replacement where the value desired is an access type. In other words, its effect is identical to first applying the `Address` attribute and then doing an unchecked conversion to a desired access type. In GNAT, but not necessarily in other implementations, the use of static chains for inner level subprograms means that `Unrestricted_Access` applied to a subprogram yields a value that can be called as long as the subprogram is in scope (normal Ada 95 accessibility rules restrict this usage).

`VADS_Size`
: The `'VADS_Size` attribute is intended to make it easier to port legacy code which relies on the semantics of `'Size` as implemented by the VADS Ada 83 compiler. GNAT makes a best effort at duplicating the same semantic interpretation. In particular, `'VADS_Size` applied to a predefined or other primitive type with no Size clause yields the Object_Size (for example, `Natural'Size` is 32 rather than 31 on typical machines). In addition `'VADS_Size` applied to an object gives the result that would be obtained by applying the attribute to the corresponding type.

`Value_Size`
: *type*`'Value_Size` is the number of bits required to represent a value of the given subtype. It is the same as *type*`'Size`, but, unlike `Size`, may be set for non-first subtypes.

`Word_Size`
: `Standard'Word_Size` (`Standard` is the only permissible prefix) provides the value `System.Word_Size` and is intended primarily for constructing this definition in package `System`.

3 Implementation Advice

The main text of the Ada 95 Reference Manual describes the required behavior of all Ada 95 compilers, and the GNAT compiler conforms to these requirements.

In addition, there are sections throughout the Ada 95 reference manual headed by the phrase "implementation advice". These sections are not normative, i.e. they do not specify requirements that all compilers must follow. Rather they provide advice on generally desirable behavior. You may wonder why they are not requirements. The most typical answer is that they describe behavior that seems generally desirable, but cannot be provided on all systems, or which may be undesirable on some systems.

As far as practical, GNAT follows the implementation advice sections in the Ada 95 Reference Manual. This chapter contains a table giving the reference manual section number, paragraph number and several keywords for each advice. Each entry consists of the text of the advice followed by the GNAT interpretation of this advice. Most often, this simply says "followed", which means that GNAT follows the advice. However, in a number of cases, GNAT deliberately deviates from this advice, in which case the text describes what GNAT does and why.

1.1.3(20): Error Detection

> If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise `Program_Error` if feasible.

Not relevant. All specialized needs annex features are either supported, or diagnosed at compile time.

1.1.3(31): Child Units

> If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit.

Followed.

1.1.5(12): Bounded Errors

> If an implementation detects a bounded error or erroneous execution, it should raise `Program_Error`.

Followed in all cases in which the implementation detects a bounded error or erroneous execution. Not all such situations are detected at runtime.

2.8(16): Pragmas

> Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.

The following implementation defined pragmas are exceptions to this rule:

`Abort_Defer`
: Affects semantics

`Ada_83`
: Affects legality

`Assert`
: Affects semantics

`CPP_Class`
: Affects semantics

`CPP_Constructor`
: Affects semantics

`CPP_Destructor`
: Affects semantics

`CPP_Virtual`
: Affects semantics

`CPP_Vtable`
: Affects semantics

`Debug`
: Affects semantics

`Interface_Name`
: Affects semantics

`Machine_Attribute`
: Affects semantics

`Unimplemented_Unit`
: Affects legality

`Unchecked_Union`
: Affects semantics

In each of the above cases, it is essential to the purpose of the pragma that this advice not be followed. For details see the separate section on implementation defined pragmas.

2.8(17-19): Pragmas

> Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:

> A pragma used to complete a declaration, such as a pragma `Import`;

> A pragma used to configure the environment by adding, removing, or replacing `library_items`.

See response to paragraph 16 of this same section.

3.5.2(5): Alternative Character Sets

> If an implementation supports a mode with alternative interpretations for `Character` and `Wide_Character`, the set of graphic characters of `Character` should nevertheless remain a proper subset of the set of graphic characters of `Wide_Character`. Any character set "localizations" should be reflected in the results of the subprograms defined in the language-defined package `Characters.Handling` (see A.3) available in such a mode. In a mode with an alternative interpretation of `Character`, the implementation should also support a corresponding change in what is a legal `identifier_letter`.

Not all wide character modes follow this advice, in particular the JIS and IEC modes reflect standard usage in Japan, and in these encoding, the upper half of the Latin-1 set is not part of the wide-character subset, since the most significant bit is used for wide character encoding. However, this only applies to the external forms. Internally there is no such restriction.

3.5.4(28): Integer Types

> An implementation should support `Long_Integer` in addition to `Integer` if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package `Standard`. Instead, appropriate named integer subtypes should be provided in the library package `Interfaces` (see B.2).

`Long_Integer` is supported. Other standard integer types are supported so this advice is not fully followed. These types are supported for convenient interface to C, and so that all hardware types of the machine are easily available.

3.5.4(29): Integer Types

> An implementation for a two's complement machine should support modular types with a binary modulus up to `System.Max_Int*2+2`. An implementation should support a nonbinary modules up to `Integer'Last`.

Followed.

3.5.5(8): Enumeration Values

> For the evaluation of a call on *S*'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type (perhaps due to an un-initialized variable), then the implementation should raise `Program_Error`. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause.

Followed.

3.5.7(17): Float Types

> An implementation should support `Long_Float` in addition to `Float` if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package `Standard`. Instead, appropriate named floating point subtypes should be provided in the library package `Interfaces` (see B.2).

`Short_Float` and `Long_Long_Float` are also provided. The former provides improved compatibility with other implementations supporting this type. The latter corresponds to the highest precision floating-point type supported by the hardware. On most machines, this will be the same as `Long_Float`, but on some machines, it will correspond to the IEEE extended form. On the Silicon Graphics processors, which do not support IEEE extended form, `Long_Long_Float` is the same as `Long_Float`.

3.6.2(11): Multidimensional Arrays

> An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if a pragma `Convention` (`Fortran`, ...) applies to a multidimensional array type, then column-major order should be used instead (see B.5, "Interfacing with Fortran").

Followed.

9.6(30-31): Duration'Small

> Whenever possible in an implementation, the value of `Duration'Small` should be no greater than 100 microseconds.

Followed. (`Duration'Small` = 10**(-9)).

> The time base for `delay_relative_statements` should be monotonic; it need not be the same time base as used for `Calendar.Clock`.

Followed.

10.2.1(12): Consistent Representation

> In an implementation, a type declared in a pre-elaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version.

Followed, except in the case of tagged types. Tagged types involve implicit pointers to a local copy of a dispatch table, and these pointers have representations which thus depend on a particular elaboration of the package. It is not easy to see how it would be possible to follow this advice without severely impacting efficiency of execution.

11.4.1(19): Exception Information

> `Exception_Message` by default and `Exception_Information` should produce information useful for debugging. `Exception_Message` should be short, about one line. `Exception_Information` can be long. `Exception_Message` should not include the `Exception_Name`. `Exception_Information` should include both the `Exception_Name` and the `Exception_Message`.

Followed.

11.5(28): Suppression of Checks

> The implementation should minimize the code executed for checks that have been suppressed.

Followed.

13.1 (21-24): Representation Clauses

> The recommended level of support for all representation items is qualified as follows:

> An implementation need not support representation items containing non-static expressions, except that an implementation should support a representation item for a given entity if each non-static expression in the representation item is a name that statically denotes a constant declared before the entity.

Followed. GNAT does not support non-static expressions in representation clauses unless they are constants declared before the entity. For example:

```
X : typ;
for X'Address use To_address (16#2000#);
```

will be rejected, since the To_Address expression is non-static. Instead write:

```
X_Address : constant Address : =
```

```
To_Address    ((16#2000#);
X : typ;
for X'Address use X_Address;
```

> An implementation need not support a specification for the `Size` for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.

Followed. Size Clauses are not permitted on non-static components, as described above.

> An aliased component, or a component whose type is by-reference, should always be allocated at an addressable location.

Followed.

13.2(6-8): Packed Types

> If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.

> The recommended level of support pragma `Pack` is:

> For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any `record_representation_clause` that applies to the type; the mplementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose `Size` is greater than the word size may be allocated an integral number of words.

Followed. Tight packing of arrays is supported for all component sizes up to 32-bits, which is the word size on typical implementations of GNAT.

> An implementation should support Address clauses for imported subprograms.

Followed.

13.3(14-19): Address Clauses

> For an array *X*, *X*`'Address` should point at the first component of the array, and not at the array bounds.

Followed.

> The recommended level of support for the `Address` attribute is:

> *X*`'Address` should produce a useful result if *X* is an object that is aliased or of a by-reference type, or is an entity whose `Address` has been specified.

Followed. A valid address will be produced even if none of those conditions have been met. If necessary, the object is forced into memory to ensure the address is valid.

> An implementation should support `Address` clauses for imported subprograms.

Followed.

> Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries.

Followed.

> If the `Address` of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.

Followed.

13.3(29-35): Alignment Clauses

> The recommended level of support for the `Alignment` attribute for subtypes is:

> An implementation should support specified Alignments that are factors and multiples of the number of storage elements per word, subject to the following:

Followed.

> An implementation need not support specified `Alignment`s for combinations of `Size`s and `Alignment`s that cannot be easily loaded and stored by available machine instructions.

Followed.

> An implementation need not support specified `Alignment`s that are greater than the maximum `Alignment` the implementation ever returns by default.

Followed.

> The recommended level of support for the `Alignment` attribute for objects is:

> Same as above, for subtypes, but in addition:

Followed.

> For stand-alone library-level objects of statically constrained subtypes, the implementation should support all `Alignment`s supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

Followed.

13.3(42-43): Size Clauses

> The recommended level of support for the `Size` attribute of objects is:

> A `Size` clause should be supported for an object if the specified `Size` is at least as large as its subtype's `Size`, and corresponds to a size in storage elements that is a multiple of the object's `Alignment` (if the `Alignment` is nonzero).

Followed.

13.3(50-56): Size Clauses

> If the `Size` of a subtype is specified, and allows for efficient independent addressability (see 9.10) on the target architecture, then the `Size` of the following objects of the subtype should equal the `Size` of the subtype:

> Aliased objects (including components).

Followed.

> Size clause on a composite subtype should not affect the internal layout of components.

Followed.

> The recommended level of support for the Size attribute of subtypes is:

> The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation.

Followed.

> For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

Followed.

13.3(71-73): Component Size Clauses

> The recommended level of support for the Component_Size attribute is:

> An implementation need not support specified Component_Sizes that are less than the Size of the component subtype.

Followed.

> An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.

Followed.

13.4(9-10): Enumeration Representation Clauses

> The recommended level of support for enumeration representation clauses is:

> An implementation need not support enumeration representation clauses for boolean types, but should at minium support the internal codes in the range `System.Min_Int.System.Max_Int`.

Followed.

13.5.1(17-22): Record Representation Clauses

> The recommended level of support for `record_representation_clauses` is:

> An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.

Followed.

> A storage place should be supported if its size is equal to the `Size` of the component subtype, and it starts and ends on a boundary that obeys the `Alignment` of the component subtype.

Followed.

> If the default bit ordering applies to the declaration of a given type, then for a component whose subtype's `Size` is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

Followed.

> An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place.

Followed.

> An implementation need not support a `component_clause` for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified.

Followed. The above advice on record representation clauses is followed, and all mentioned features are implemented.

13.5.2(5): Storage Place Attributes

> If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontinuously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes.

Followed. There are no such components in GNAT.

13.5.3(7-8): Bit Ordering

> The recommended level of support for the non-default bit ordering is:

> If `Word_Size` = `Storage_Unit`, then the implementation should support the non-default bit ordering in addition to the default bit ordering.

Followed. Word size does not equal storage size in this implementation. Thus non-default bit ordering is not supported.

13.7(37): Address as Private

> `Address` should be of a private type.

Followed.

13.7.1(16): Address Operations

> Operations in `System` and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to "wrap around." Operations that do not make sense should raise `Program_Error`.

Followed. Address arithmetic is modular arithmetic that wraps around. No operation raises `Program_Error`, since all operations make sense.

13.9(14-17): Unchecked Conversion

> The `Size` of an array object should not include its bounds; hence, the bounds should not be part of the converted data.

Followed.

> The implementation should not generate unnecessary run-time checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment.

Followed. There are no restrictions on unchecked conversion. A warning is generated if the soure and target types do not have the same size since the semantics in this case may be target dependent.

> The recommended level of support for unchecked conversions is:

> Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph.

Followed.

13.11(23-25): Implicit Heap Usage

> An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator.

Followed, the only other points at which heap storage is dynamically allocated are as follows:

- At initial elaboration time, to allocate dynamically sized global objects.
- To allocate space for a task when a task is created.
- To extend the secondary stack dynamically when needed. The secondary stack is used for returning variable length results.

> A default (implementation-provided) storage pool for an access-to- constant type should not have overhead to support de-allocation of individual objects.

Followed.

> A storage pool for an anonymous access type should be created at the point of an allocator for the type, and be reclaimed when the designated object becomes inaccessible.

Followed.

13.11.2(17): Unchecked De-allocation

> For a standard storage pool, `Free` should actually reclaim the storage.

Followed.

13.13.2(17): Stream Oriented Attributes

> If a stream element is the same size as a storage element, then the normal in-memory representation should be used by `Read` and `Write` for scalar objects. Otherwise, `Read` and `Write` should use the smallest number of stream elements needed to represent all values in the base range of the scalar type.

Followed.

A.1(52): Implementation Advice

> If an implementation provides additional named predefined integer types, then the names should end with '`Integer`' as in '`Long_Integer`'. If an implementation provides additional named predefined floating point types, then the names should end with '`Float`' as in '`Long_Float`'.

Followed.

A.3.2(49): `Ada.Characters.Handling`

> If an implementation provides a localized definition of `Character` or `Wide_Character`, then the effects of the subprograms in `Characters.Handling` should reflect the localizations. See also 3.5.2.

Followed. GNAT provides no such localized definitions.

A.4.4(106): Bounded-Length String Handling

> Bounded string objects should not be implemented by implicit pointers and dynamic allocation.

Followed. No implicit pointers or dynamic allocation are used.

A.5.2(46-47): Random Number Generation

> Any storage associated with an object of type `Generator` should be reclaimed on exit from the scope of the object.

Followed.

> If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of `Initiator` passed to `Reset` should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value.

Followed. The generator period is sufficiently long for the first condition here to hold true.

A.10.7(23): `Get_Immediate`

> The `Get_Immediate` procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be *available* if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of `Get_Immediate`.

Followed.

B.1(39-41): Pragma `Export`

> If an implementation supports pragma `Export` to a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are `adainit` and `adafinal`. `adainit` should contain the elaboration code for library units. `adafinal` should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called.

Followed.

> Automatic elaboration of pre-elaborated packages should be provided when pragma Export is supported.

Followed when the main program is in Ada. If the main program is in a foreign language, then `adainit` must be called to elaborate pre-elaborated packages.

> For each supported convention *L* other than `Intrinsic`, an implementation should support `Import` and `Export` pragmas for objects of *L*-compatible types and for subprograms, and pragma `Convention` for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Pragma `Convention` need not be supported for scalar types.

Followed.

B.2(12-13): Package `Interfaces`

> For each implementation-defined convention identifier, there should be a child package of package Interfaces with the corresponding name. This package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in `Interfaces`.

Followed. An additional package not defined in the Ada 95 Reference Manual is `Interfaces.CPP`, used for interfacing to C++.

> An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following clauses.

Followed. GNAT provides all the packages described in this section.

B.3(63-71): Interfacing with C

> An implementation should support the following interface correspondences between Ada and C.

Followed.

> An Ada procedure corresponds to a void-returning C function.

Followed.

> An Ada function corresponds to a non-void C function.

Followed.

> An Ada `in` scalar parameter is passed as a scalar argument to a C function.

Followed.

> An Ada `in` parameter of an access-to-object type with designated type *T* is passed as a *t*`*` argument to a C function, where *t* is the C type corresponding to the Ada type *T*.

Followed.

> An Ada access *T* parameter, or an Ada `out` or `in out` parameter of an elementary type *T*, is passed as a *t*`*` argument to a C function, where *t* is the C type corresponding to the Ada type *T*. In the case of an elementary `out` or `in out` parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

Followed.

> An Ada parameter of a record type *T*, of any mode, is passed as a *t*`*` argument to a C function, where *t* is the C structure corresponding to the Ada type *T*.

Followed. This convention may be overridden by the use of the C_Pass_By_Copy pragma, or Convention, or by explicitly specifying the mechanism for a given call using an extended import or export pragma.

> An Ada parameter of an array type with component type *T*, of any mode, is passed as a *t*`*` argument to a C function, where *t* is the C type corresponding to the Ada type *T*.

Followed.

> An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

Followed.

B.4(95-98): Interfacing with COBOL

> An Ada implementation should support the following interface correspondences between Ada and COBOL.

Followed.

> An Ada access *T* parameter is passed as a "BY REFERENCE" data item of the COBOL type corresponding to *T*.

Followed.

> An Ada in scalar parameter is passed as a "BY CONTENT" data item of the corresponding COBOL type.

Followed.

> Any other Ada parameter is passed as a "BY REFERENCE" data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics.

Followed.

B.5(22-26): Interfacing with Fortran

> An Ada implementation should support the following interface correspondences between Ada and Fortran:

Followed.

> An Ada procedure corresponds to a Fortran subroutine.

Followed.

> An Ada function corresponds to a Fortran function.

Followed.

> An Ada parameter of an elementary, array, or record type T is passed as a T argument to a Fortran procedure, where T is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics.

Followed.

> An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification.

Followed.

C.1(3-5): Access to Machine Operations

> The machine code or intrinsic support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any.

Followed.

> The interfacing pragmas (see Annex B) should support interface to assembler; the default assembler should be associated with the convention identifier `Assembler`.

Followed.

> If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported.

Followed.

C.1(10-16): Access to Machine Operations

> The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms.

Followed for both intrinsics and machine-code subprograms.

> It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs.

Followed. A full set of machine operation intrinsic subprograms is provided.

> Atomic read-modify-write operations – e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.

Followed on any target supporting such operations.

> Standard numeric functions – e.g., sin, log.

Followed on any target supporting such operations.

> String manipulation operations – e.g., translate and test.

Followed on any target supporting such operations.

> Vector operations – e.g., compare vector against thresholds.

Followed on any target supporting such operations.

> Direct operations on I/O ports.

Followed on any target supporting such operations.

C.3(28): Interrupt Support

> If the `Ceiling_Locking` policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for a finer-grain control of interrupt blocking.

Followed. The underlying system does not allow for finer-grain control of interrupt blocking.

C.3.1(20-21): Protected Procedure Handlers

> Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.

Followed on any target where the underlying operating system permits such direct calls.

> Whenever practical, violations of any implementation-defined restrictions should be detected before run time.

Followed. Compile time warnings are given when possible.

C.3.2(25): Package `Interrupts`

> If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to `Parameterless_Handler` should be specified in a child package of `Interrupts`, with the same operations as in the predefined package Interrupts.

Followed.

C.4(14): Pre-elaboration Requirements

> It is recommended that pre-elaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements.

Followed. Executable code is generated in some cases, e.g. loops to initialize large arrays.

C.5(8): Pragma `Discard_Names`

> If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

Followed.

C.7.2(30): The Package Task_Attributes

> Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the task's attributes, or by using the pre-allocated storage for the first *N* attribute objects, and the heap for the others. In the latter case, *N* should be documented.

Not followed. This implementation is not targeted to such a domain.

D.3(17): Locking Policies

> The implementation should use names that end with '`_Locking`' for locking policies defined by the implementation.

Followed. No such implementation-defined locking policies exist.

D.4(16): Entry Queuing Policies

> Names that end with '`_Queuing`' should be used for all implementation-defined queuing policies.

Followed. No such implementation-defined queueing policies exist.

D.6(9-10): Preemptive Abort

> Even though the `abort_statement` is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the `abort_statement` to block.

Followed.

> On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.

Followed.

D.7(21): Tasking Restrictions

> When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation.

Not followed. GNAT does not currently take advantage of any specified restrictions.

D.8(47-49): Monotonic Time

> When appropriate, implementations should provide configuration mechanisms to change the value of `Tick`.

Such configuration mechanisms are not appropriate to this implementation and are thus not supported.

> It is recommended that `Calendar.Clock` and `Real_Time.Clock` be implemented as transformations of the same time base.

Followed.

> It is recommended that the *best* time base which exists in the underlying system be available to the application through `Clock`. *Best* may mean highest accuracy or largest range.

Followed.

E.5(28-29): Partition Communication Subsystem

> Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns.

Followed by GLADE, a separately supplied PCS that can be used with GNAT. For information on GLADE, contact Ada Core Technologies.

> The `Write` operation on a stream of type `Params_Stream_Type` should raise `Storage_Error` if it runs out of space trying to write the `Item` into the stream.

Followed by GLADE, a separately supplied PCS that can be used with GNAT. For information on GLADE, contact Ada Core Technologies.

F(7): COBOL Support

> If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package `Interfaces.COBOL` (respectively, `Interfaces.C`) specified in Annex B and should support a `convention_identifier` of COBOL (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language.

Followed.

F.1(2): Decimal Radix Support

> Packed decimal should be used as the internal representation for objects of subtype S when S'Machine_Radix = 10.

Not followed. GNAT ignores S'Machine_Radix and always uses binary representations.

G: Numerics

> If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package `Interfaces.Fortran` (respectively, `Interfaces.C`) specified in Annex B and should support a `convention_identifier` of Fortran (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language.

Followed.

G.1.1(56-58): Complex Types

> Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to IEC 559:1989, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand.

Not followed.

> Similarly, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In implementations in which the `Signed_Zeros` attribute of the component type is `True` (and which therefore conform to IEC 559:1989 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand.

Not followed.

> Implementations in which `Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the `Argument` function should have the sign of the imaginary component of the parameter `X` when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the `Compose_From_Polar` function should be the same as (respectively, the opposite of) that of the `Argument` parameter when that parameter has a value of zero and the `Modulus` parameter has a nonnegative (respectively, negative) value.

Followed.

G.1.2(49): Complex Elementary Functions

> Implementations in which `Complex_Types.Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative.

Followed.

G.2.4(19): Accuracy Requirements

> The versions of the forward trigonometric functions without a `Cycle` parameter should not be implemented by calling the correspondingg version with a `Cycle` parameter of `2.0*Numerics.Pi`, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of `Log` without a `Base` parameter should not be implemented by calling the corresponding version with a `Base` parameter of `Numerics.e`.

Followed.

G.2.6(15): Complex Arithmetic Accuracy

> The version of the `Compose_From_Polar` function without a `Cycle` parameter should not be implemented by calling the corresponding version with a `Cycle` parameter of `2.0*Numerics.Pi`, since this will not provide the required accuracy in some portions of the domain.

Followed.

4 Implementation Defined Characteristics

In addition to the implementation dependent pragmas and attributes, and the implementation advice, there are a number of other features of Ada 95 that are potentially implementation dependent. These are mentioned throughout the Ada 95 Reference Manual, and are summarized in annex M.

A requirement for conforming Ada compilers is that they provide documentation describing how the implementation deals with each of these issues. In this chapter, you will find each point in annex M listed followed by a description in italic font of how GNAT handles the implementation dependence.

You can use this chapter as a guide to minimizing implementation dependent features in your programs if portability to other compilers and other operating systems is an important consideration. The numbers in each section below correspond to the paragraph number in the Ada 95 Reference Manual.

2. Whether or not each recommendation given in Implementation Advice is followed. See 1.1.2(37).

See Chapter 3 [Implementation Advice], page 37.

3. Capacity limitations of the implementation. See 1.1.3(3).

The complexity of programs that can be processed is limited only by the total amount of available virtual memory, and disk space for the generated object files.

4. Variations from the standard that are impractical to avoid given the implementation's execution environment. See 1.1.3(6).

There are no variations from the standard.

5. Which `code_statement`s cause external interactions. See 1.1.3(10).

Any `code_statement` can potentially cause external interactions.

6. The coded representation for the text of an Ada program. See 2.1(4).

See separate section on source representation.

7. The control functions allowed in comments. See 2.1(14).

See separate section on source representation.

8. The representation for an end of line. See 2.2(2).

See separate section on source representation.

9. Maximum supported line length and lexical element length. See 2.2(15).

The maximum line length is 255 characters an the maximum length of a lexical element is also 255 characters.

10. Implementation defined pragmas. See 2.8(14).

See Chapter 1 [Implementation Defined Pragmas], page 3.

11. Effect of pragma `Optimize`. See 2.8(27).

Pragma `Optimize`, if given with a `Time` or `Space` parameter, checks that the optimization flag is set, and aborts if it is not.

12. The sequence of characters of the value returned by *S*`'Image` when some of the graphic characters of *S*`'Wide_Image` are not defined in `Character`. See 3.5(37).

The sequence of characters is as defined by the wide character encoding method used for the source. See section on source representation for further details.

13. The predefined integer types declared in `Standard`. See 3.5.4(25).

`Short_Short_Integer`
: 8 bit signed

`Short_Integer`
: (Short) 16 bit signed

`Integer`
: 32 bit signed

`Long_Integer`
: 32 bit signed

`Long_Long_Integer`
: 64 bit signed

14. Any nonstandard integer types and the operators defined for them. See 3.5.4(26).

There are no nonstandard integer types.

15. Any nonstandard real types and the operators defined for them. See 3.5.6(8).

There are no nonstandard real types.

16. What combinations of requested decimal precision and range are supported for floating point types. See 3.5.7(7).

The precision and range is as defined by the IEEE standard.

17. The predefined floating point types declared in `Standard`. See 3.5.7(16).

`Short_Float`
: 32 bit IEEE short

`Float`
: (Short) 32 bit IEEE short

`Long_Float`
: 64 bit IEEE long

`Long_Long_Float`
: 64 bit IEEE long

18. The small of an ordinary fixed point type. See 3.5.9(8).

`Fine_Delta` is 2**(-63)

19. What combinations of small, range, and digits are supported for fixed point types. See 3.5.9(10).

Any combinations are permitted that do not result in a small less than `Fine_Delta` and do not result in a mantissa larger than 63 bits.

20. The result of `Tags.Expanded_Name` for types declared within an unnamed `block_statement`. See 3.9(10).

Block numbers of the form `B`*nnn*, where *nnn* is a decimal integer are allocated.

21. Implementation-defined attributes. See 4.1.4(12).

See Chapter 2 [Implementation Defined Attributes], page 29.

22. Any implementation-defined time types. See 9.6(6).

There are no implementation-defined time types.

23. The time base associated with relative delays.

See 9.6(20). The time base used is that provided by the C library function `gettimeofday`.

24. The time base of the type `Calendar.Time`. See 9.6(23).

The time base used is that provided by the C library function `gettimeofday`.

25. The timezone used for package `Calendar` operations. See 9.6(24).

The timezone used by package `Calendar` is the current system timezone setting for local time, as accessed by the C library function `localtime`.

26. Any limit on `delay_until_statements` of `select_statements`. See 9.6(29).

There are no such limits.

27. Whether or not two nonoverlapping parts of a composite object are independently addressable, in the case where packing, record layout, or `Component_Size` is specified for the object. See 9.10(1).

Separate components are independently addressable if they do not share overlapping storage units.

28. The representation for a compilation. See 10.1(2).

A compilation is represented by a sequence of files presented to the compiler in a single invocation of the '`gcc`' command.

29. Any restrictions on compilations that contain multiple compilation_units. See 10.1(4).

No single file can contain more than one compilation unit, but any sequence of files can be presented to the compiler as a single compilation.

30. The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3).

See separate section on compilation model.

31. The manner of explicitly assigning library units to a partition. See 10.2(2).

See separate section on binding and linking programs.

32. The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2).

See separate section on compilation unit.

33. The manner of designating the main subprogram of a partition. See 10.2(7).

The main program is designated by providing the name of the corresponding ali file as the input parameter to the binder.

34. The order of elaboration of `library_items`. See 10.2(18).

The first constraint on ordering is that it meets the requirements of chapter 10 of the Ada 95 Reference Manual. This still leaves some implementation dependent choices, which are resolved by first elaborating bodies as early as possible (i.e. in preference to specs where there is a choice), and second by evaluating the immediate with clauses of a unit to determine the probably best choice, and third by elaborating in alphabetical order of unit names where a choice still remains.

35. Parameter passing and function return for the main subprogram. See 10.2(21).

The main program has no parameters. It may be a procedure, or a function returning an integer type. In the latter case, the returned integer value is the return code of the program.

36. The mechanisms for building and running partitions. See 10.2(24).

GNAT itself supports programs with only a single partition. The GNATDIST tool provided with the GLADE package (which also includes an implementation of the PCS) provides a completely flexible method for building and running programs consisting of multiple partitions. See the separate GLADE manual for details.

37. The details of program execution, including program termination. See 10.2(25).

See separate section on compilation model.

38. The semantics of any nonactive partitions supported by the implementation. See 10.2(28).

Passive partitions are supported on targets where shared memory is provided by the operating system. See the GLADE reference manual for further details.

39. The information returned by `Exception_Message`. See 11.4.1(10).

Exception message returns the null string unless a specific message has been passed by the program.

40. The result of `Exceptions.Exception_Name` for types declared within an unnamed `block_statement`. See 11.4.1(12).

Blocks have implementation defined names of the form `B`*nnn* where *nnn* is an integer.

41. The information returned by `Exception_Information`. See 11.4.1(13).

`Exception_Information` contains the expanded name of the exception in upper case, and no other information.

42. Implementation-defined check names. See 11.5(27).

No implementation-defined check names are supported.

43. The interpretation of each aspect of representation. See 13.1(20).

See separate section on data representations.

44. Any restrictions placed upon representation items. See 13.1(20).

See separate section on data representations.

45. The meaning of `Size` for indefinite subtypes. See 13.3(48).

Size for an indefinite subtype is the maximum possible size, except that for the case of a subprogram parameter, the size of the parameter object is the actual size.

46. The default external representation for a type tag. See 13.3(75).

The default external representation for a type tag is the fully expanded name of the type in upper case letters.

47. What determines whether a compilation unit is the same in two different partitions. See 13.3(76).

A compilation unit is the same in two different partitions if and only if it derives from the same source file.

48. Implementation-defined components. See 13.5.1(15).

The only implementation defined component is the tag for a tagged type, which contains a pointer to the dispatching table.

49. If `Word_Size` = `Storage_Unit`, the default bit ordering. See 13.5.3(5).

`Word_Size` (32) is not the same as `Storage_Unit` (8) for this implementation, so no non-default bit ordering is supported. The default bit ordering corresponds to the natural endianness of the target architecture.

50. The contents of the visible part of package `System` and its language-defined children. See 13.7(2).

See the definition of these packages in files '`system.ads`' and '`s-stoele.ads`'.

51. The contents of the visible part of package `System.Machine_Code`, and the meaning of `code_statements`. See 13.8(7).

See the definition and documentation in file '`s-maccod.ads`'.

52. The effect of unchecked conversion. See 13.9(11).

Unchecked conversion between types of the same size and results in an uninterpreted transmission of the bits from one type to the other. If the types are of unequal sizes, then in the case of discrete types, a shorter source is first zero or sign extended as necessary, and a shorter target is simply truncated on the left. For all non-discrete types, the source is first copied if necessary to ensure that the alignment requirements of the target are met, then a pointer is constructed to the source value, and the result is obtained by dereferencing this pointer after converting it to be a pointer to the target type.

53. The manner of choosing a storage pool for an access type when Storage_Pool is not specified for the type. See 13.11(17).

There are 3 different standard pools used by the compiler when Storage_Pool is not specified depending whether the type is local to a subprogram or defined at the library level and whether Storage_Sizeis specified or not. See documentation in the runtime library units System.Pool_Global, System.Pool_Size and System.Pool_Local in files 's-poosiz.ads', 's-pooglo.ads' and 's-pooloc.ads' for full details on the default pools used.

54. Whether or not the implementation provides user-accessible names for the standard pool type(s). See 13.11(17).

See documentation in the sources of the run time mentioned in paragraph **53** . All these pools are accessible by means of with'ing these units.

55. The meaning of Storage_Size. See 13.11(18).

Storage_Size is measured in storage units, and refers to the total space available for an access type collection, or to the primary stack space for a task.

56. Implementation-defined aspects of storage pools. See 13.11(22).

See documentation in the sources of the run time mentioned in paragraph **53** for details on GNAT-defined aspects of storage pools.

57. The set of restrictions allowed in a pragma Restrictions. See 13.12(7).

All RM defined Restriction identifiers are implemented. The following additional restriction identifiers are provided:

No_Implementation_Attributes
: This restriction checks at compile time that no GNAT-defined attributes are present. With this restriction, the only attributes that can be used are those defined in the Ada 95 Reference Manual.

No_Implementation_Pragmas
: This restriction checks at compile time that no GNAT-defined pragmas are present. With this restriction, the only pragmas that can be used are those defined in the Ada 95 Reference Manual.

No_Elaboration_Code
: This restriction ensures at compile time that no elaboration code is generated. Note that this is not the same condition as is enforced by pragma Preelaborate. There are cases in which pragma Preelaborate still permits code to be generated (e.g. code to initialize a large array to all zeroes), and there are cases of units

which do not meet the requirements for pragma Preelaborate, but for which no elaboration code is generated. Generally, it is the case that preelaborable units will meet the restrictions, with the exception of large aggregates initialized by others.

58. The consequences of violating limitations on `Restrictions` pragmas. See 13.12(9).

Restrictions that can be checked at compile time result in illegalities if violated. Currently there are no other consequences of violating restrictions.

59. The representation used by the `Read` and `Write` attributes of elementary types in terms of stream elements. See 13.13.2(9).

The representation is the in-memory representation of the base type of the type, using the number of bits corresponding to the *type*`'Size` value, and the natural ordering of the machine.

60. The names and characteristics of the numeric subtypes declared in the visible part of package `Standard`. See A.1(3).

See items describing the integer and floating-point types supported.

61. The accuracy actually achieved by the elementary functions. See A.5.1(1).

The elementary functions correspond to the functions available in the C library. Only fast math mode is implemented.

62. The sign of a zero result from some of the operators or functions in `Numerics.Generic_Elementary_Functions`, when `Float_Type'Signed_Zeros` is `True`. See A.5.1(46).

The sign of zeroes follows the requirements of the IEEE 754 standard on floating-point.

63. The value of `Numerics.Float_Random.Max_Image_Width`. See A.5.2(27).

Maximum image width is 649, see library file '`a-numran.ads`'.

64. The value of `Numerics.Discrete_Random.Max_Image_Width`. See A.5.2(27).

Maximum image width is 80, see library file '`a-nudira.ads`'.

65. The algorithms for random number generation. See A.5.2(32).

The algorithm is documented in the source files 'a-numran.ads' and 'a-numran.adb'.

66. The string representation of a random number generator's state. See A.5.2(38).

See the documentation contained in the file 'a-numran.adb'.

67. The minimum time interval between calls to the time-dependent Reset procedure that are guaranteed to initiate different random number sequences. See A.5.2(45).

The minimum period between reset calls to guarantee distinct series of random numbers is one microsecond.

68. The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported. See A.5.3(72).

See the source file 'ttypef.ads' for the values of all numeric attributes.

69. Any implementation-defined characteristics of the input-output packages. See A.7(14).

There are no special implementation defined characteristics for these packages.

70. The value of Buffer_Size in Storage_IO. See A.9(10).

All type representations are contiguous, and the Buffer_Size is the value of *type*'Size rounded up to the next storage unit boundary.

71. External files for standard input, standard output, and standard error See A.10(5).

These files are mapped onto the files provided by the C streams libraries. See source file 'i-cstrea.ads' for further details.

72. The accuracy of the value produced by Put. See A.10.9(36).

If more digits are requested in the output than are represented by the precision of the value, zeroes are output in the corresponding least significant digit positions.

73. The meaning of Argument_Count, Argument, and Command_Name. See A.15(1).

These are mapped onto the `argv` and `argc` parameters of the main program in the natural manner.

74. Implementation-defined convention names. See B.1(11).

The following convention names are supported

`Ada` Ada

`Asm` Assembly language

`Assembler`
Assembly language

`C` C

`C_Pass_By_Copy`
Treated like C, except for record types

`COBOL` COBOL

`CPP` C++

`Default` Treated the same as C

`External` Treated the same as C

`Fortran` Fortran

`Intrinsic`
Intrinsic

`Stdcall` Stdcall (used for NT implementations only)

In addition, all otherwise unrecognized convention names are also treated as being synonymous with convention C. In all implementations except for VMS, use of such other names results in a warning. In VMS implementations, these names are accepted silently.

75. The meaning of link names. See B.1(36).

Link names are the actual names used by the linker.

76. The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See B.1(36).

The default linker name is that which would be assigned by the relevant external language, interpreting the Ada name as being in all lower case letters.

77. The effect of pragma `Linker_Options`. See B.1(37).

The string passed to `Linker_Options` is presented uninterpreted as an argument to the link command.

78. The contents of the visible part of package `Interfaces` and its language-defined descendants. See B.2(1).

See files with prefix '`i-`' in the distributed library.

79. Implementation-defined children of package `Interfaces`. The contents of the visible part of package `Interfaces`. See B.2(11).

See files with prefix '`i-`' in the distributed library.

80. The types `Floating`, `Long_Floating`, `Binary`, `Long_Binary`, `Decimal_ Element`, and `COBOL_Character`; and the initialization of the variables `Ada_To_COBOL` and `COBOL_To_Ada`, in `Interfaces.COBOL`. See B.4(50).

`Floating` Float

`Long_Floating`
(Floating) Long_Float

`Binary` Integer

`Long_Binary`
Long_Long_Integer

`Decimal_Element`
Character

`COBOL_Character`
Character

For initialization, see the file '`i-cobol.ads`' in the distributed library.

81. Support for access to machine instructions. See C.1(1).

See documentation in file '`s-maccod.ads`' in the distributed library.

82. Implementation-defined aspects of access to machine operations. See C.1(9).

See documentation in file '`s-maccod.ads`' in the distributed library.

83. Implementation-defined aspects of interrupts. See C.3(2).

Interrupts are mapped to signals or conditions as appropriate. See definition of unit `Ada.Interrupt_Names` in source file '`a-intnam.ads`' for details on the interrupts supported on a particular target.

84. Implementation-defined aspects of pre-elaboration. See C.4(13).

GNAT does not permit a partition to be restarted without reloading, except under control of the debugger.

85. The semantics of pragma `Discard_Names`. See C.5(7).

Pragma `Discard_Names` is currently ignored.

86. The result of the `Task_Identification.Image` attribute. See C.7.1(7).

The result of this attribute is an 8-digit hexadecimal string representing the virtual address of the task control block.

87. The value of `Current_Task` when in a protected entry or interrupt handler. See C.7.1(17).

Protected entries or interrupt handlers can be executed by any convenient thread, so the value of `Current_Task` is undefined.

88. The effect of calling `Current_Task` from an entry body or interrupt handler. See C.7.1(19).

The effect of calling `Current_Task` from an entry body or interrupt handler is to return the identification of the task currently executing the code.

89. Implementation-defined aspects of `Task_Attributes`. See C.7.2(19).

There are no implementation-defined aspects of `Task_Attributes`.

90. Values of all `Metrics`. See D(2).

Information on metrics is not yet available.

91. The declarations of `Any_Priority` and `Priority`. See D.1(11).

See declarations in file '`system.ads`'.

92. Implementation-defined execution resources. See D.1(15).

There are no implementation-defined execution resources.

93. Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See D.2.1(3).

On a multi-processor, a task that is waiting for access to a protected object does not keep its processor busy.

94. The affect of implementation defined execution resources on task dispatching. See D.2.1(9).

Tasks map to threads in the threads package used by GNAT. Where possible and appropriate, these threads correspond to native threads of the underlying operating system.

95. Implementation-defined `policy_identifiers` allowed in a pragma `Task_Dispatching_Policy`. See D.2.2(3).

There are no implementation-defined policy-identifiers allowed in this pragma.

96. Implementation-defined aspects of priority inversion. See D.2.2(16).

Execution of a task cannot be preempted by the implementation processing of delay expirations for lower priority tasks.

97. Implementation defined task dispatching. See D.2.2(18).

The policy is the same as that of the underlying threads implementation.

98. Implementation-defined `policy_identifiers` allowed in a pragma `Locking_Policy`. See D.3(4).

There are no implementation defined policy identifiers allowed in this pragma.

99. Default ceiling priorities. See D.3(10).

The ceiling priority of protected objects of the type `System.Interrupt_Priority'Last` as described in the Ada 95 Reference Manual D.3(10),

100. The ceiling of any protected object used internally by the implementation. See D.3(16).

The ceiling priority of internal protected objects is `System.Priority'Last`.

101. Implementation-defined queuing policies. See D.4(1).

There are no implementation-defined queueing policies.

102. On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. See D.6(3).

The semantics for abort on a multi-processor is the same as on a single processor, there are no further delays.

103. Any operations that implicitly require heap storage allocation. See D.7(8).

The only operation that implicitly requires heap storage allocation is task creation.

104. Implementation-defined aspects of pragma `Restrictions`. See D.7(20).

There are no such implementation-defined aspects.

105. Implementation-defined aspects of package `Real_Time`. See D.8(17).

There are no implementation defined aspects of package `Real_Time`.

106. Implementation-defined aspects of `delay_statements`. See D.9(8).

Any difference greater than one microsecond will cause the task to be delayed (see D.9(7)).

107. The upper bound on the duration of interrupt blocking caused by the implementation. See D.12(5).

The upper bound is determined by the underlying operating system. In no cases is it more than 10 milliseconds.

108. The means for creating and executing distributed programs. See E(5).

The GLADE package provides a utility GNATDIST for creating and executing distributed programs. See the GLADE reference manual for further details.

109. Any events that can result in a partition becoming inaccessible. See E.1(7).

See the GLADE reference manual for full details on such events.

110. The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See E.1(11).

See the GLADE reference manual for full details on these aspects of multi-partition execution.

111. Events that cause the version of a compilation unit to change. See E.3(5).

Editing the source file of a compilation unit, or the source files of any units on which it is dependent in a significant way cause the version to change. No other actions cause the version number to change. All changes are significant except those which affect only layout, capitalization or comments.

112. Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See E.4(13).

See the GLADE reference manual for details on the effect of abort in a distributed application.

113. Implementation-defined aspects of the PCS. See E.5(25).

See the GLADE reference manual for a full description of all implementation defined aspects of the PCS.

114. Implementation-defined interfaces in the PCS. See E.5(26).

See the GLADE reference manual for a full description of all implementation defined interfaces.

115. The values of named numbers in the package `Decimal`. See F.2(7).

`Max_Scale`
+18

`Min_Scale`
-18

`Min_Delta`
1.0E-18

`Max_Delta`
1.0E+18

`Max_Decimal_Digits`
18

116. The value of `Max_Picture_Length` in the package `Text_IO.Editing`. See F.3.3(16).

64

117. The value of `Max_Picture_Length` in the package `Wide_Text_IO.Editing`. See F.3.4(5).

64

118. The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See G.1(1).

Standard library functions are used for the complex arithmetic operations. Only fast math mode is currently supported.

119. The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic_Complex_Types`, when `Real'Signed_Zeros` is True. See G.1.1(53).

The signs of zero values are as recommended by the relevant implementation advice.

120. The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic_Complex_Elementary_Functions`, when `Real'Signed_Zeros` is `True`. See G.1.2(45).

The signs of zero values are as recommended by the relevant implementation advice.

121. Whether the strict mode or the relaxed mode is the default. See G.2(2).

The relaxed mode is the default.

122. The result interval in certain cases of fixed-to-float conversion. See G.2.1(10).

For cases where the result interval is implementation dependent, the accuracy is that provided by performing all operations in 64-bit IEEE floating-point format.

123. The result of a floating point arithmetic operation in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.1(13).

Infinite and Nan values are produced as dictated by the IEEE floating-point standard.

124. The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See G.2.1(16).

Not relevant, division is IEEE exact.

125. The definition of close result set, which determines the accuracy of certain fixed point multiplications and divisions. See G.2.3(5).

Operations in the close result set are performed using IEEE long format floating-point arithmetic. The input operands are converted to floating-point, the operation is done in floating-point, and the result is converted to the target type.

126. Conditions on a `universal_real` operand of a fixed point multiplication or division for which the result shall be in the perfect result set. See G.2.3(22).

The result is only defined to be in the perfect result set if the result can be computed by a single scaling operation involving a scale factor representable in 64-bits.

127. The result of a fixed point arithmetic operation in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.3(27).

Not relevant, `Machine_Overflows` is `True` for fixed-point types.

128. The result of an elementary function reference in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.4(4).

IEEE infinite and Nan values are produced as appropriate.

129. The value of the angle threshold, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See G.2.4(10).

Information on this subject is not yet available.

130. The accuracy of certain elementary functions for parameters beyond the angle threshold. See G.2.4(10).

Information on this subject is not yet available.

131. The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the `Machine_Overflows` attribute of the corresponding real type is `False`. See G.2.6(5).

IEEE infinite and Nan values are produced as appropriate.

132. The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See G.2.6(8).

Information on those subjects is not yet available.

133. Information regarding bounded errors and erroneous execution. See H.2(1).

Information on this subject is not yet available.

134. Implementation-defined aspects of pragma `Inspection_Point`. See H.3.2(8).

Pragma `Inspection_Point` ensures that the variable is live and can be examined by the debugger at the inspection point.

135. Implementation-defined aspects of pragma `Restrictions`. See H.4(25).

There are no implementation-defined aspects of pragma `Restrictions`. The use of pragma `Restrictions [No_Exceptions]` has no effect on the generated code. Checks must suppressed by use of pragma `Suppress`.

136. Any restrictions on pragma `Restrictions`. See H.4(27).

There are no restrictions on pragma `Restrictions`.

5 Standard Library Routines

The Ada 95 Reference Manual contains in Annex A a full description of an extensive set of standard library routines that can be used in any Ada program, and which must be provided by all Ada compilers. They are analogous to the standard C library used by C programs.

GNAT implements all of the facilities described in annex A, and for most purposes the description in the Ada 95 reference manual, or appropriate Ada text book, will be sufficient for making use of these facilities.

In the case of the input-output facilities, See Chapter 6 [The Implementation of Standard I/O], page 89, gives details on exactly how GNAT interfaces to the file system. For the remaining packages, the Ada 95 reference manual should be sufficient. The following is a list of the packages included, together with a brief description of the functionality that is provided.

For completeness, references are included to other predefined library routines defined in other sections of the Ada 95 reference manual (these are cross-indexed from annex A).

`Ada (A.2)`
: This is a parent package for all the standard library packages. It is usually included implicitly in your program, and itself contains no useful data or routines.

`Ada.Calendar (9.6)`
: `Calendar` provides time of day access, and routines for manipulating times and durations.

`Ada.Characters (A.3.1)`
: This is a dummy parent package that contains no useful entities

`Ada.Characters.Handling (A.3.2)`
: This package provides some basic character handling capabilities, including classification functions for classes of characters (e.g. test for letters, or digits).

`Ada.Characters.Latin_1 (A.3.3)`
: This package includes a complete set of definitions of the characters that appear in type CHARACTER. It is useful for writing programs that will run in international environments. For example, if you want an upper case E with an acute accent in a string, it is often better to use the definition of `UC_E_Acute` in this package. Then your program will print in an understandable manner even if your environment does not support these extended characters.

`Ada.Command_Line (A.15)`
: This package provides access to the command line parameters and the name of the current program (analogous to the use of argc and argv in C), and also allows the exit status for the program to be set in a system-independent manner.

`Ada.Decimal (F.2)`
: This package provides constants describing the range of decimal numbers implemented, and also a decimal divide routine (analogous to the COBOL verb DIVIDE .. GIVING .. REMAINDER ..)

`Ada.Direct_IO (A.8.4)`
: This package provides input-output using a model of a set of records of fixed-length, containing an arbitrary definite Ada type, indexed by an integer record number.

`Ada.Dynamic_Priorities (D.5)`
: This package allows the priorities of a task to be adjusted dynamically as the task is running.

`Ada.Exceptions (11.4.1)`
: This package provides additional information on exceptions, and also contains facilities for treating exceptions as data objects, and raising exceptions with associated messages.

`Ada.Finalization (7.6)`
: This package contains the declarations and subprograms to support the use of controlled types, providing for automatic initialization and finalization (analogous to the constructors and destructors of C++)

`Ada.Interrupts (C.3.2)`
: This package provides facilities for interfacing to interrupts, which includes the set of signals or conditions that can be raised and recognized as interrupts.

`Ada.Interrupts.Names (C.3.2)`
: This package provides the set of interrupt names (actually signal or condition names) that can be handled by GNAT.

`Ada.IO_Exceptions (A.13)`
: This package defines the set of exceptions that can be raised by use of the standard IO packages.

`Ada.Numerics`
: This package contains some standard constants and exceptions used throughout the numerics packages. Note that the constants pi and e are defined here, and it is better to use these definitions than rolling your own.

`Ada.Numerics.Complex_Elementary_Functions`
: Provides the implementation of standard elementary functions (such as log and trigonometric functions) operating on complex numbers using the standard `Float` and the `Complex` and `Imaginary` types created by the package `Numerics.Complex_Types`.

`Ada.Numerics.Complex_Types`
: This is a predefined instantiation of `Numerics.Generic_Complex_Types` using `Standard.Float` to build the type `Complex` and `Imaginary`.

`Ada.Numerics.Discrete_Random`
: This package provides a random number generator suitable for generating random integer values from a specified range.

`Ada.Numerics.Float_Random`
: This package provides a random number generator suitable for generating uniformly distributed floating point values.

`Ada.Numerics.Generic_Complex_Elementary_Functions`

This is a generic version of the package that provides the implementation of standard elementary functions (such as log an trigonometric functions) for an arbitrary complex type.

The following predefined instantiations of this package exist

`Short_Float`
`Ada.Numerics.Short_Complex_Elementary_Functions`

`Float` `Ada.Numerics.Complex_Elementary_Functions`

`Long_Float`
`Ada.Numerics.Long_Complex_Elementary_Functions`

`Ada.Numerics.Generic_Complex_Types`

This is a generic package that allows the creation of complex types, with associated complex arithmetic operations.

The following predefined instantiations of this package exist

`Short_Float`
`Ada.Numerics.Short_Complex_Complex_Types`

`Float` `Ada.Numerics.Complex_Complex_Types`

`Long_Float`
`Ada.Numerics.Long_Complex_Complex_Types`

`Ada.Numerics.Generic_Elementary_Functions`

This is a generic package that provides the implementation of standard elementary functions (such as log an trigonometric functions) for an arbitrary float type.

The following predefined instantiations of this package exist

`Short_Float`
`Ada.Numerics.Short_Elementary_Functions`

`Float` `Ada.Numerics.Elementary_Functions`

`Long_Float`
`Ada.Numerics.Long_Elementary_Functions`

`Ada.Real_Time (D.8)`

This package provides facilities similar to those of `Calendar`, but operating with a finer clock suitable for real time control.

`Ada.Sequential_IO (A.8.1)`

This package provides input-output facilities for sequential files, which can contain a sequence of values of a single type, which can be any Ada type, including indefinite (unconstrained) types.

`Ada.Storage_IO (A.9)`

This package provides a facility for mapping arbitrary Ada types to and from a storage buffer. It is primarily intended for the creation of new IO packages.

`Ada.Streams (13.13.1)`
: This is a generic package that provides the basic support for the concept of streams as used by the stream attributes (`Input`, `Output`, `Read` and `Write`).

`Ada.Streams.Stream_IO (A.12.1)`
: This package is a specialization of the type `Streams` defined in package `Streams` together with a set of operations providing Stream_IO capability. The Stream_IO model permits both random and sequential access to a file which can contain an arbitrary set of values of one or more Ada types.

`Ada.Strings (A.4.1)`
: This package provides some basic constants used by the string handling packages.

`Ada.Strings.Bounded (A.4.4)`
: This package provides facilities for handling variable length strings. The bounded model requires a maximum length. It is thus somewhat more limited than the unbounded model, but avoids the use of dynamic allocation or finalization.

`Ada.Strings.Fixed (A.4.3)`
: This package provides facilities for handling fixed length strings.

`Ada.Strings.Maps (A.4.2)`
: This package provides facilities for handling character mappings and arbitrarily defined subsets of characters. For instance it is useful in defining specialized translation tables.

`Ada.Strings.Maps.Constants (A.4.6)`
: This package provides a standard set of predefined mappings and predefined character sets. For example, the standard upper to lower case conversion table is found in this package. Note that upper to lower case conversion is non-trivial if you want to take the entire set of characters, including extended characters like E with an acute accent, into account. You should use the mappings in this package (rather than adding 32 yourself) to do case mappings.

`Ada.Strings.Unbounded (A.4.5)`
: This package provides facilities for handling variable length strings. The unbounded model allows arbitrary length strings, but requires the use of dynamic allocation and finalization.

`Ada.Strings.Wide_Bounded (A.4.7)`
`Ada.Strings.Wide_Fixed (A.4.7)`
`Ada.Strings.Wide_Maps (A.4.7)`
`Ada.Strings.Wide_Maps.Constants (A.4.7)`
`Ada.Strings.Wide_Unbounded (A.4.7)`
: These package provide analogous capabilities to the corresponding packages without '`Wide_`' in the name, but operate with the types `Wide_String` and `Wide_Character` instead of `String` and `Character`.

`Ada.Synchronous_Task_Control (D.10)`
: This package provides some standard facilities for controlling task communication in a synchronous manner.

`Ada.Tags` This package contains definitions for manipulation of the tags of tagged values.

`Ada.Task_Attributes`
: This package provides the capability of associating arbitrary task-specific data with separate tasks.

`Ada.Text_IO`
: This package provides basic text input-output capabilities for character, string and numeric data. The subpackages of this package are listed next.

`Ada.Text_IO.Decimal_IO`
: Provides input-output facilities for decimal fixed-point types

`Ada.Text_IO.Enumeration_IO`
: Provides input-output facilities for enumeration types.

`Ada.Text_IO.Fixed_IO`
: Provides input-output facilities for ordinary fixed-point types.

`Ada.Text_IO.Float_IO`
: Provides input-output facilities for float types. The following predefined instantiations of this generic package are available:

  `Short_Float`
  : `Short_Float_Text_IO`

  `Float` `Float_Text_IO`

  `Long_Float`
  : `Long_Float_Text_IO`

`Ada.Text_IO.Integer_IO`
: Provides input-output facilities for integer types. The following predefined instantiations of this generic package are available:

  `Short_Short_Integer`
  : `Ada.Short_Short_Integer_Text_IO`

  `Short_Integer`
  : `Ada.Short_Integer_Text_IO`

  `Integer` `Ada.Integer_Text_IO`

  `Long_Integer`
  : `Ada.Long_Integer_Text_IO`

  `Long_Long_Integer`
  : `Ada.Long_Long_Integer_Text_IO`

`Ada.Text_IO.Modular_IO`
: Provides input-output facilities for modular (unsigned) types

`Ada.Text_IO.Complex_IO (G.1.3)`
: This package provides basic text input-output capabilities for complex data.

`Ada.Text_IO.Editing (F.3.3)`
: This package contains routines for edited output, analogous to the use of pictures in COBOL. The picture formats used by this package are a close copy of the facility in COBOL.

`Ada.Text_IO.Text_Streams (A.12.2)`
: This package provides a facility that allows Text_IO files to be treated as streams, so that the stream attributes can be used for writing arbitrary data, including binary data, to Text_IO files.

`Ada.Unchecked_Conversion (13.9)`
: This generic package allows arbitrary conversion from one type to another of the same size, providing for breaking the type safety in special circumstances.

`Ada.Unchecked_Deallocation (13.11.2)`
: This generic package allows explicit freeing of storage previously allocated by use of an allocator.

`Ada.Wide_Text_IO (A.11)`
: This package is similar to `Ada.Text_IO`, except that the external file supports wide character representations, and the internal types are `Wide_Character` and `Wide_String` instead of `Character` and `String`. It contains generic subpackages listed next.

`Ada.Wide_Text_IO.Decimal_IO`
: Provides input-output facilities for decimal fixed-point types

`Ada.Wide_Text_IO.Enumeration_IO`
: Provides input-output facilities for enumeration types.

`Ada.Wide_Text_IO.Fixed_IO`
: Provides input-output facilities for ordinary fixed-point types.

`Ada.Wide_Text_IO.Float_IO`
: Provides input-output facilities for float types. The following predefined instantiations of this generic package are available:

  `Short_Float`
  : `Short_Float_Wide_Text_IO`

  `Float`
  : `Float_Wide_Text_IO`

  `Long_Float`
  : `Long_Float_Wide_Text_IO`

`Ada.Wide_Text_IO.Integer_IO`
: Provides input-output facilities for integer types. The following predefined instantiations of this generic package are available:

  `Short_Short_Integer`
  : `Ada.Short_Short_Integer_Wide_Text_IO`

`Short_Integer`
: `Ada.Short_Integer_Wide_Text_IO`

`Integer`
: `Ada.Integer_Wide_Text_IO`

`Long_Integer`
: `Ada.Long_Integer_Wide_Text_IO`

`Long_Long_Integer`
: `Ada.Long_Long_Integer_Wide_Text_IO`

`Ada.Wide_Text_IO.Modular_IO`
: Provides input-output facilities for modular (unsigned) types

`Ada.Wide_Text_IO.Complex_IO (G.1.3)`
: This package is similar to `Ada.Text_IO.Complex_IO`, except that the external file supports wide character representations.

`Ada.Wide_Text_IO.Editing (F.3.4)`
: This package is similar to `Ada.Text_IO.Editing`, except that the types are `Wide_Character` and `Wide_String` instead of `Character` and `String`.

`Ada.Wide_Text_IO.Streams (A.12.3)`
: This package is similar to `Ada.Text_IO.Streams`, except that the types are `Wide_Character` and `Wide_String` instead of `Character` and `String`.

6 The Implementation of Standard I/O

GNAT implements all the required input-output facilities described in A.6 through A.14. These sections of the Ada 95 reference manual describe the required behavior of these packages from the Ada point of view, and if you are writing a portable Ada program that does not need to know the exact manner in which Ada maps to the outside world when it comes to reading or writing external files, then you do not need to read this chapter. As long as your files are all regular files (not pipes or devices), and as long as you write and read the files only from Ada, the description in the Ada 95 reference manual is sufficient.

However, if you want to do input-output to pipes or other devices, such as the keyboard or screen, or if the files you are dealing with are either generated by some other language, or to be read by some other language, then you need to know more about the details of how the GNAT implementation of these input-output facilities behaves.

In this chapter we give a detailed description of exactly how GNAT interfaces to the file system. As always, the sources of the system are available to you for answering questions at an even more detailed level, but for most purposes the information in this chapter will suffice.

Another reason that you may need to know more about how input-output is implemented arises when you have a program written in mixed languages where, for example, files are shared between the C and Ada sections of the same program. GNAT provides some additional facilities, in the form of additional child library packages, that facilitate this sharing, and these additional facilities are also described in this chapter.

6.1 Standard I/O Packages

The Standard I/O packages described in Annex A for

- Ada.Text_IO
- Ada.Text_IO.Complex_IO
- Ada.Text_IO.Text_Streams,
- Ada.Wide_Text_IO
- Ada.Wide_Text_IO.Complex_IO,
- Ada.Wide_Text_IO.Text_Streams
- Ada.Stream_IO
- Ada.Sequential_IO
- Ada.Direct_IO

are implemented using the C library streams facility; where

- All files are opened using `fopen`.
- All input/output operations use `fread`/`fwrite`.

There is no internal buffering of any kind at the Ada library level. The only buffering is that provided at the system level in the implementation of the C library routines that support streams. This facilitates shared use of these streams by mixed language programs.

6.2 FORM Strings

The format of a FORM string in GNAT is:

```
"keyword=value,keyword=value,...,keyword=value"
```

where letters may be in upper or lower case, and there are no spaces between values. The order of the entries is not important. Currently there are two keywords defined.

```
SHARED=[YES|NO]
WCEM=[n|h|u|s\e]
```

The use of these parameters is described later in this section.

6.3 Direct_IO

Direct_IO can only be instantiated for definite types. This is a restriction of the Ada language, which means that the records are fixed length (the length being determined by *type*`'Size`, rounded up to the next storage unit boundary if necessary).

The records of a Direct_IO file are simply written to the file in index sequence, with the first record starting at offset zero, and subsequent records following. There is no control information of any kind. For example, if 32-bit integers are being written, each record takes 4-bytes, so the record at index K starts at offset (K - 1)*4.

There is no limit on the size of Direct_IO files, they are expanded as necessary to accommodate whatever records are written to the file.

6.4 Sequential_IO

Sequential_IO may be instantiated with either a definite (constrained) or indefinite (unconstrained) type.

For the definite type case, the elements written to the file are simply the memory images of the data values with no control information of any kind. The resulting file should be read using the same type, no validity checking is performed on input.

For the indefinite type case, the elements written consist of two parts. First is the size of the data item, written as the memory image of a `Interfaces.C.size_t` value, followed by the memory image of the data value. The resulting file can only be read using the same (unconstrained) type. Normal assignment checks are performed on these read operations, and if these checks fail, `Data_Error` is raised. In particular, in the array case, the lengths must match, and in the variant record case, if the variable for a particular read operation is constrained, the discriminants must match.

Note that it is not possible to use Sequential_IO to write variable length array items, and then read the data back into different length arrays. For example, the following will raise `Data_Error`:

```
package IO is new Sequential_IO (String);
F : IO.File_Type;
S : String (1..4);
...
IO.Create (F)
```

```
IO.Write (F, "hello!")
IO.Reset (F, Mode=>In_File);
IO.Read (F, S);
Put_Line (S);
```

On some Ada implementations, this will print '`hell`', but the program is clearly incorrect, since there is only one element in the file, and that element is the string '`hello!`'.

In Ada 95, this kind of behavior can be legitimately achieved using Stream_IO, and this is the preferred mechanism. In particular, the above program fragment rewritten to use Stream_IO will work correctly.

6.5 Text_IO

Text_IO files consist of a stream of characters containing the following special control characters:

```
LF (line feed, 16#0A#) Line Mark
FF (form feed, 16#0C#) Page Mark
```

A canonical Text_IO file is defined as one in which the following conditions are met:

- The character `LF` is used only as a line mark, i.e. to mark the end of the line.
- The character `FF` is used only as a page mark, i.e. to mark the end of a page and consequently can appear only immediately following a `LF` (line mark) character.
- The file ends with either `LF` (line mark) or `LF-FF` (line mark, page mark). In the former case, the page mark is implicitly assumed to be present.

A file written using Text_IO will be in canonical form provided that no explicit `LF` or `FF` characters are written using `Put` or `Put_Line`. There will be no `FF` character at the end of the file unless an explicit `New_Page` operation was performed before closing the file.

A canonical Text_IO file that is a regular file, i.e. not a device or a pipe, can be read using any of the routines in Text_IO. The semantics in this case will be exactly as defined in the Ada 95 reference manual and all the routines in Text_IO are fully implemented.

A text file that does not meet the requirements for a canonical Text_IO file has one of the following:

- The file contains `FF` characters not immediately following a `LF` character.
- The file contains `LF` or `FF` characters written by `Put` or `Put_Line`, which are not logically considered to be line marks or page marks.
- The file ends in a character other than `LF` or `FF`, i.e. there is no explicit line mark or page mark at the end of the file.

Text_IO can be used to read such non-standard text files but subprograms to do with line or page numbers do not have defined meanings. In particular, a `FF` character that does not follow a `LF` character may or may not be treated as a page mark from the point of view of page and line numbering. Every `LF` character is considered to end a line, and there is an implied `LF` character at the end of the file.

6.5.1 Stream Pointer Positioning

`Ada.Text_IO` has a definition of current position for a file that is being read. No internal buffering occurs in Text_IO, and usually the physical position in the stream used to implement the file corresponds to this logical position defined by Text_IO. There are two exceptions:

- After a call to `End_Of_Page` that returns `True`, the stream is positioned past the `LF` (line mark) that precedes the page mark. Text_IO maintains an internal flag so that subsequent read operations properly handle the logical position which is unchanged by the `End_Of_Page` call.
- After a call to `End_Of_File` that returns `True`, if the Text_IO file was positioned before the line mark at the end of file before the call, then the logical position is unchanged, but the stream is physically positioned right at the end of file (past the line mark, and past a possible page mark following the line mark. Again Text_IO maintains internal flags so that subsequent read operations properly handle the logical position.

These discrepancies have no effect on the observable behavior of Text_IO, but if a single Ada stream is shared between a C program and Ada program, or shared (using '`shared=yes`' in the form string) between two Ada files, then the difference may be observable in some situations.

6.5.2 Reading and Writing Non-Regular Files

A non-regular file is a device (such as a keyboard), or a pipe. Text_IO can be used for reading and writing. Writing is not affected and the sequence of characters output is identical to the normal file case, but for reading, the behavior of Text_IO is modified to avoid undesirable look-ahead as follows:

An input file that is not a regular file is considered to have no page marks. Any `Ascii.FF` characters (the character normally used for a page mark) appearing in the file are considered to be data characters. In particular:

- `Get_Line` and `Skip_Line` do not test for a page mark following a line mark. If a page mark appears, it will be treated as a data character.
- This avoids the need to wait for an extra character to be typed or entered from the pipe to complete one of these operations.
- `End_Of_Page` always returns `False`
- `End_Of_File` will return `False` if there is a page mark at the end of the file.

Output to non-regular files is the same as for regular files. Page marks may be written to non-regular files using `New_Page`, but as noted above they will not be treated as page marks on input if the output is piped to another Ada program.

Another important discrepancy when reading non-regular files is that the end of file indication is not `"sticky"`. If an end of file is entered, e.g. by pressing the `EOT` key, then end of file is signalled once (i.e. the test `End_Of_File` will yield `True`, or a read will raise `End_Error`), but then reading can resume to read data past that end of file indication, until another end of file indication is entered.

6.5.3 Get_Immediate

Get_Immediate returns the next character (including control characters) from the input file. In particular, Get_Immediate will return LF or FF characters used as line marks or page marks. Such operations leave the file positioned past the control character, and it is thus not treated as having its normal function. This means that page, line and column counts after this kind of Get_Immediate call are set as though the mark did not occur. In the case where a Get_Immediate leaves the file positioned between the line mark and page mark (which is not normally possible), it is undefined whether the FF character will be treated as a page mark.

6.5.4 Treating Text_IO Files as Streams

The package `Text_IO.Streams` allows a Text_IO file to be treated as a stream. Data written to a Text_IO file in this stream mode is binary data. If this binary data contains bytes 16#0A# (`LF`) or 16#0C# (`FF`), the resulting file may have non-standard format. Similarly if read operations are used to read from a Text_IO file treated as a stream, then `LF` and `FF` characters may be skipped and the effect is similar to that described above for `Get_Immediate`.

6.6 Wide_Text_IO

`Wide_Text_IO` is similar in most respects to Text_IO, except that both input and output files may contain special sequences that represent wide character values. The encoding scheme for a given file may be specified using a FORM parameter:

```
WCEM=x
```

as part of the FORM string (WCEM = wide character encoding method), where *x* is one of the following characters

‘`h`’ Hex ESC encoding

‘`u`’ Upper half encoding

‘`s`’ Shift-JIS encoding

‘`e`’ EUC Encoding

‘`8`’ UTF-8 encoding

‘`b`’ Brackets encoding

The encoding methods match those that can be used in a source program, but there is no requirement that the encoding method used for the source program be the same as the encoding method used for files, and different files may use different encoding methods.

The default encoding method for the standard files, and for opened files for which no WCEM parameter is given in the FORM string matches the wide character encoding specified for the main program (the default being brackets encoding if no coding method was specified with -gnatW).

Hex Coding

In this encoding, a wide character is represented by a five character sequence:

```
ESC a b c d
```

where *a*, *b*, *c*, *d* are the four hexadecimal characters (using upper case letters) of the wide character code. For example, ESC A345 is used to represent the wide character with code 16#A345#. This scheme is compatible with use of the full `Wide_Character` set.

Upper Half Coding
: The wide character with encoding 16#abcd#, where the upper bit is on (i.e. a is in the range 8-F) is represented as two bytes 16#ab# and 16#cd#. The second byte may never be a format control character, but is not required to be in the upper half. This method can be also used for shift-JIS or EUC where the internal coding matches the external coding.

Shift JIS Coding
: A wide character is represented by a two character sequence 16#ab# and 16#cd#, with the restrictions described for upper half encoding as described above. The internal character code is the corresponding JIS character according to the standard algorithm for Shift-JIS conversion. Only characters defined in the JIS code set table can be used with this encoding method.

EUC Coding
: A wide character is represented by a two character sequence 16#ab# and 16#cd#, with both characters being in the upper half. The internal character code is the corresponding JIS character according to the EUC encoding algorithm. Only characters defined in the JIS code set table can be used with this encoding method.

UTF-8 Coding
: A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value, the representation is a one, two, or three byte sequence:

```
16#0000#-16#007f#: 2#0xxxxxxx#
16#0080#-16#07ff#: 2#110xxxxx# 2#10xxxxxx#
16#0800#-16#ffff#: 2#1110xxxx# 2#10xxxxxx# 2#10xxxxxx#
```

where the xxx bits correspond to the left-padded bits of the the 16-bit character value. Note that all lower half ASCII characters are represented as ASCII bytes and all upper half characters and other wide characters are represented as sequences of upper-half (The full UTF-8 scheme allows for encoding 31-bit characters as 6-byte sequences, but in this implementation, all UTF-8 sequences of four or more bytes length will raise a Constraint_Error, as will all illegal UTF-8 sequences.)

Brackets Coding
: In this encoding, a wide character is represented by the following eight character sequence:

```
[ " a b c d " ]
```

Where `a`, `b`, `c`, `d` are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, `["A345"]` is used to represent the wide character with code `16#A345#`. This scheme is compatible with use of the full Wide_Character set. On input, brackets coding can also be used for upper half characters, e.g. `["C1"]` for lower case a. However, on output, brackets notation is only used for wide characters with a code greater than `16#FF#`.

For the coding schemes other than Hex and Brackets encoding, not all wide character values can be represented. An attempt to output a character that cannot be represented using the encoding scheme for the file causes Constraint_Error to be raised. An invalid wide character sequence on input also causes Constraint_Error to be raised.

6.6.1 Stream Pointer Positioning

`Ada.Wide_Text_IO` is similar to `Ada.Text_IO` in its handling of stream pointer positioning (see Section 6.5 [Text_IO], page 91). There is one additional case:

If `Ada.Wide_Text_IO.Look_Ahead` reads a character outside the normal lower ASCII set (i.e. a character in the range:

```
Wide_Character'Val (16#0080#) .. Wide_Character'Val (16#FFFF#)
```

then although the logical position of the file pointer is unchanged by the `Look_Ahead` call, the stream is physically positioned past the wide character sequence. Again this is to avoid the need for buffering or backup, and all `Wide_Text_IO` routines check the internal indication that this situation has occurred so that this is not visible to a normal program using `Wide_Text_IO`. However, this discrepancy can be observed if the wide text file shares a stream with another file.

6.6.2 Reading and Writing Non-Regular Files

As in the case of Text_IO, when a non-regular file is read, it is assumed that the file contains no page marks (any form characters are treated as data characters), and `End_Of_Page` always returns `False`. Similarly, the end of file indication is not sticky, so it is possible to read beyond an end of file.

6.7 Stream_IO

A stream file is a sequence of bytes, where individual elements are written to the file as described in the Ada 95 reference manual. The type `Stream_Element` is simply a byte. There are two ways to read or write a stream file.

- The operations `Read` and `Write` directly read or write a sequence of stream elements with no control information.
- The stream attributes applied to a stream file transfer data in the manner described for stream attributes.

6.8 Shared Files

Section A.14 of the Ada 95 Reference Manual allows implementations to provide a wide variety of behavior if an attempt is made to access the same external file with two or more internal files.

To provide a full range of functionality, while at the same time minimizing the problems of portability caused by this implementation dependence, GNAT handles file sharing as follows:

- In the absence of a '`shared=`*xxx*' form parameter, an attempt to open two or more files with the same full name is considered an error and is not supported. The exception `Use_Error` will be raised. Note that a file that is not explicitly closed by the program remains open until the program terminates.
- If the form parameter '`shared=no`' appears in the form string, the file can be opened or created with its own separate stream identifier, regardless of whether other files sharing the same external file are opened. The exact effect depends on how the C stream routines handle multiple accesses to the same external files using separate streams.
- If the form parameter '`shared=yes`' appears in the form string for each of two or more files opened using the same full name, the same stream is shared between these files, and the semantics are as described in Ada 95 Reference Manual, Section A.14.

When a program that opens multiple files with the same name is ported from another Ada compiler to GNAT, the effect will be that `Use_Error` is raised.

The documentation of the original compiler and the documentation of the program should then be examined to determine if file sharing was expected, and '`shared=`*xxx*' parameters added to `Open` and `Create` calls as required.

When a program is ported from GNAT to some other Ada compiler, no special attention is required unless the '`shared=`*xxx*' form parameter is used in the program. In this case, you must examine the documentation of the new compiler to see if it supports the required file sharing semantics, and form strings modified appropriately. Of course it may be the case that the program cannot be ported if the target compiler does not support the required functionality. The best approach in writing portable code is to avoid file sharing (and hence the use of the '`shared=`*xxx*' parameter in the form string) completely.

One common use of file sharing in Ada 83 is the use of instantiations of Sequential_IO on the same file with different types, to achieve heterogenous input-output. Although this approach will work in GNAT if '`shared=yes`' is specified, it is preferable in Ada 95 to use Stream_IO for this purpose (using the stream attributes)

6.9 Open Modes

`Open` and `Create` calls result in a call to `fopen` using the mode shown in Table 6.1

Table 6-1 `Open` and `Create` Call Modes

	OPEN	**CREATE**
`Append_File`	`"r+"`	`"w+"`
`In_File`	`"r"`	`"w+"`
`Out_File (Direct_IO)`	`"r+"`	`"w"`
`Out_File (all other cases)`	`"w"`	`"w"`
`Inout_File`	`"r+"`	`"w+"`

If text file translation is required, then either '`b`' or '`t`' is added to the mode, depending on the setting of Text. Text file translation refers to the mapping of CR/LF sequences in an external file to LF characters internally. This mapping only occurs in DOS and DOS-like systems, and is not relevant to other systems.

A special case occurs with Stream_IO. As shown in the above table, the file is initially opened in 'r' or 'w' mode for the `In_File` and `Out_File` cases. If a `Set_Mode` operation subsequently requires switching from reading to writing or vice-versa, then the file is reopened in 'r+' mode to permit the required operation.

6.10 Operations on C Streams

The package `Interfaces.C_Streams` provides an Ada program with direct access to the C library functions for operations on C streams:

```
package Interfaces.C_Streams is
  -- Note: the reason we do not use the types that are in
  -- Interfaces.C is that we want to avoid dragging in the
  -- code in this unit if possible.
  subtype chars is System.Address;
  -- Pointer to null-terminated array of characters
  subtype FILEs is System.Address;
  -- Corresponds to the C type FILE*
  subtype voids is System.Address;
  -- Corresponds to the C type void*
  subtype int is Integer;
  subtype long is Long_Integer;
  -- Note: the above types are subtypes deliberately, and it
  -- is part of this spec that the above correspondences are
  -- guaranteed. This means that it is legitimate to, for
  -- example, use Integer instead of int. We provide these
  -- synonyms for clarity, but in some cases it may be
  -- convenient to use the underlying types (for example to
  -- avoid an unnecessary dependency of a spec on the spec
  -- of this unit).
  type size_t is mod 2 ** Standard'Address_Size;
  NULL_Stream : constant FILEs;
  -- Value returned (NULL in C) to indicate an
  -- fdopen/fopen/tmpfile error
  ----------------------------------
  -- Constants Defined in stdio.h --
  ----------------------------------
  EOF : constant int;
  -- Used by a number of routines to indicate error or
  -- end of file
  IOFBF : constant int;
  IOLBF : constant int;
  IONBF : constant int;
  -- Used to indicate buffering mode for setvbuf call
  SEEK_CUR : constant int;
  SEEK_END : constant int;
  SEEK_SET : constant int;
```

```
-- Used to indicate origin for fseek call
function stdin return FILEs;
function stdout return FILEs;
function stderr return FILEs;
-- Streams associated with standard files
--------------------------
-- Standard C functions --
--------------------------
-- The functions selected below are ones that are
-- available in DOS, OS/2, UNIX and Xenix (but not
-- necessarily in ANSI C). These are very thin interfaces
-- which copy exactly the C headers. For more
-- documentation on these functions, see the Microsoft C
-- "Run-Time Library Reference" (Microsoft Press, 1990,
-- ISBN 1-55615-225-6), which includes useful information
-- on system compatibility.
procedure clearerr (stream : FILEs);
function fclose (stream : FILEs) return int;
function fdopen (handle : int; mode : chars) return FILEs;
function feof (stream : FILEs) return int;
function ferror (stream : FILEs) return int;
function fflush (stream : FILEs) return int;
function fgetc (stream : FILEs) return int;
function fgets (strng : chars; n : int; stream : FILEs)
    return chars;
function fileno (stream : FILEs) return int;
function fopen (filename : chars; Mode : chars)
    return FILEs;
-- Note: to maintain target independence, use
-- text_translation_required, a boolean variable defined in
-- a-sysdep.c to deal with the target dependent text
-- translation requirement. If this variable is set,
-- then  b/t should be appended to the standard mode
-- argument to set the text translation mode off or on
-- as required.
function fputc (C : int; stream : FILEs) return int;
function fputs (Strng : chars; Stream : FILEs) return int;
function fread
   (buffer : voids;
    size : size_t;
    count : size_t;
    stream : FILEs)
    return size_t;
function freopen
   (filename : chars;
    mode : chars;
    stream : FILEs)
    return FILEs;
```

```
function fseek
   (stream : FILEs;
    offset : long;
    origin : int)
    return int;
function ftell (stream : FILEs) return long;
function fwrite
   (buffer : voids;
    size : size_t;
    count : size_t;
    stream : FILEs)
    return size_t;
function isatty (handle : int) return int;
procedure mktemp (template : chars);
-- The return value (which is just a pointer to template)
-- is discarded
procedure rewind (stream : FILEs);
function rmtmp return int;
function setvbuf
   (stream : FILEs;
    buffer : chars;
    mode : int;
    size : size_t)
    return int;

function tmpfile return FILEs;
function ungetc (c : int; stream : FILEs) return int;
function unlink (filename : chars) return int;
---------------------
-- Extra functions --
---------------------
-- These functions supply slightly thicker bindings than
-- those above. They are derived from functions in the
-- C Run-Time Library, but may do a bit more work than
-- just directly calling one of the Library functions.
function is_regular_file (handle : int) return int;
-- Tests if given handle is for a regular file (result 1)
-- or for a non-regular file (pipe or device, result 0).
---------------------------------
-- Control of Text/Binary Mode --
---------------------------------
-- If text_translation_required is true, then the following
-- functions may be used to dynamically switch a file from
-- binary to text mode or vice versa. These functions have
-- no effect if text_translation_required is false (i.e. in
-- normal UNIX mode). Use fileno to get a stream handle.
procedure set_binary_mode (handle : int);
procedure set_text_mode (handle : int);
```

```
    ----------------------------
    -- Full Path Name support --
    ----------------------------
    procedure full_name (nam : chars; buffer : chars);
    -- Given a NUL terminated string representing a file
    -- name, returns in buffer a NUL terminated string
    -- representing the full path name for the file name.
    -- On systems where it is relevant the   drive is also
    -- part of the full path name. It is the responsibility
    -- of the caller to pass an actual parameter for buffer
    -- that is big enough for any full path name. Use
    -- max_path_len given below as the size of buffer.
    max_path_len : integer;
    -- Maximum length of an allowable full path name on the
    -- system, including a terminating NUL character.
  end Interfaces.C_Streams;
```

6.11 Interfacing to C Streams

The packages in this section permit interfacing Ada files to C Stream operations.

```
with Interfaces.C_Streams;
package Ada.Sequential_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Sequential_IO.C_Streams;

 with Interfaces.C_Streams;
 package Ada.Direct_IO.C_Streams is
    function C_Stream (F : File_Type)
       return Interfaces.C_Streams.FILEs;
    procedure Open
      (File : in out File_Type;
       Mode : in File_Mode;
       C_Stream : in Interfaces.C_Streams.FILEs;
       Form : in String := "");
 end Ada.Direct_IO.C_Streams;

 with Interfaces.C_Streams;
 package Ada.Text_IO.C_Streams is
    function C_Stream (F : File_Type)
       return Interfaces.C_Streams.FILEs;
    procedure Open
      (File : in out File_Type;
```

```
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Text_IO.C_Streams;
```

```
with Interfaces.C_Streams;
package Ada.Wide_Text_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Wide_Text_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Stream_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Stream_IO.C_Streams;
```

In each of these five packages, the `C_Stream` function obtains the `FILE` pointer from a currently opened Ada file. It is then possible to use the `Interfaces.C_Streams` package to operate on this stream, or the stream can be passed to a C program which can operate on it directly. Of course the program is responsible for ensuring that only appropriate sequences of operations are executed.

One particular use of relevance to an Ada program is that the `setvbuf` function can be used to control the buffering of the stream used by an Ada file. In the absence of such a call the standard default buffering is used.

The `Open` procedures in these packages open a file giving an existing C Stream instead of a file name. Typically this stream is imported from a C program, allowing an Ada file to operate on an existing C file.

7 Interfacing to Other Languages

The facilities in annex B of the Ada 95 Reference Manual are fully implemented in GNAT, and in addition, a full interface to C++ is provided.

7.1 Interfacing to C

Interfacing to C with GNAT can use one of two approaches:

1. The types in the package `Interfaces.C` may be used.
2. Standard Ada types may be used directly. This may be less portable to other compilers, but will work on all GNAT compilers, which guarantee correspondence between the C and Ada types.

Pragma `Convention C` maybe applied to Ada types, but mostly has no effect, since this is the default. The following table shows the correspondence between Ada scalar types and the corresponding C types.

`Integer` `int`

`Short_Integer`
`short`

`Short_Short_Integer`
`signed char`

`Long_Integer`
`long`

`Long_Long_Integer`
`long long`

`Short_Float`
`float`

`Float` `float`

`Long_Float`
`double`

`Long_Long_Float`
This is the longest floating-point type supported by the hardware. Sometimes, this is the same as `Long_Float`, i.e. as the C type `double`. Otherwise, it is a wider type which is also available as `long double` in GNU C.

- Ada enumeration types map to C enumeration types directly if pragma `Convention C` is specified, which causes them to have int length. Without pragma `Convention C`, Ada enumeration types map to 8, 16, or 32 bits (i.e. C types signed char, short, int respectively) depending on the number of values passed. This is the only case in which pragma `Convention C` affects the representation of an Ada type.
- Ada access types map to C pointers, except for the case of pointers to unconstrained types in Ada, which have no direct C equivalent.
- Ada arrays map directly to C arrays.

- Ada records map directly to C structures.
- Packed Ada records map to C structures where all members are bit fields of the length corresponding to the *type*'Size value in Ada.

7.2 Interfacing to C++

The interface to C++ makes use of the following pragmas, which are usually constructed automatically using the binding generator tool. Using these pragmas it is possible to achieve complete inter-operability between Ada tagged types and C class definitions. See Chapter 1 [Implementation Defined Pragmas], page 3 for more details.

`pragma CPP_Class ([Entity =>] `*local˙name*`)`
: The argument denotes an entity in the current declarative region that is declared as a tagged or untagged record type. It indicates that the type corresponds to an externally declared C++ class type, and is to be laid out the same way that C++ would lay out the type.

`pragma CPP_Constructor ([Entity =>] `*local˙name*`)`
: This pragma identifies an imported function (imported in the usual way with pragma `Import`) as corresponding to a C++ constructor.

`pragma CPP_Vtable ...`
: One `CPP_Vtable` pragma can be present for each component of type `CPP.Interfaces.Vtable_Ptr` in a record to which pragma `CPP_Class` applies.

7.3 Interfacing to COBOL

Interfacing to COBOL is achieved as described in section B.4 of the Ada 95 reference manual.

7.4 Interfacing to Fortran

Interfacing to Fortran is achieved as described in section B.5 of the reference manual. The pragma `Convention Fortran`, applied to a multi- dimensional array causes the array to be stored in column-major order as required for convenient interface to Fortran.

7.5 Interfacing to non-GNAT Ada code

It is possible to specify the convention Ada in a pragma Import or pragma Export. However this refers to the calling conventions used by GNAT, which may or may not be similar enough to those used by some other Ada 83 or Ada 95 compiler to allow interoperation.

If arguments types are kept simple, and if the foreign compiler generally follows system calling conventions, then it may be possible to integrate files compiled by other Ada compilers, provided that the elaboration issues are adequately addressed (for example by eliminating the need for any load time elaboration).

In particular, GNAT running on VMS is designed to be highly compatible with the DEC Ada 83 compiler, so this is one case in which it is possible to import foreign units of this type, provided that the data items passed are restricted to simple scalar values or simple record types without variants, or simple array types with fixed bounds.

8 Machine Code Insertions

Package `Machine_Code` provides machine code support as described in the Ada 95 Reference Manual in two separate forms:

- Machine code statements, consisting of qualified expressions that fit the requirements of RM section 13.8.
- An intrinsic callable procedure, providing an alternative mechanism of including machine instructions in a subprogram.

The two features are similar, and both closely related to the mechanism provided by the asm instruction in the GNU C compiler. Full understanding and use of the facilities in this package requires understanding the asm instruction as described in *Using and Porting GNU CC* by Richard Stallman. Calls to the function `Asm` and the procedure `Asm` have identical semantic restrictions and effects as described below. Both are provided so that the procedure call can be used as a statement, and the function call can be used to form a code_statement.

The first example given in the GNU CC documentation is the C `asm` instruction:

```
asm ("fsinx %1 %0" : "=f" (result) : "f" (angle));
```

The equivalent can be written for GNAT as:

```
Asm ("fsinx %1 %0",
     My_Float'Asm_Output ("=f", result),
     My_Float'Asm_Input  ("f",  angle));
```

The first argument to `Asm` is the assembler template, and is identical to what is used in GNU CC. This string must be a static expression. The second argument is the output operand list. It is either a single `Asm_Output` attribute reference, or a list of such references enclosed in parentheses (technically an array aggregate of such references).

The `Asm_Output` attribute denotes a function that takes two parameters. The first is a string, the second is the name of a variable of the type designated by the attribute prefix. The first (string) argument is required to be a static expression and designates the constraint for the parameter (e.g. what kind of register is required). The second argument is the variable to be updated with the result. The possible values for constraint are the same as those used in the RTL, and are dependent on the configuration file used to build the GCC back end. If there are no output operands, then this argument may either be omitted, or explicitly given as `No_Output_Operands`.

The second argument of *my˙float*'`Asm_Output` functions as though it were an `out` parameter, which is a little curious, but all names have the form of expressions, so there is no syntactic irregularity, even though normally functions would not be permitted `out` parameters. The third argument is the list of input operands. It is either a single `Asm_Input` attribute reference, or a list of such references enclosed in parentheses (technically an array aggregate of such references).

The `Asm_Input` attribute denotes a function that takes two parameters. The first is a string, the second is an expression of the type designated by the prefix. The first (string) argument is required to be a static expression, and is the constraint for the parameter, (e.g. what kind of register is required). The second argument is the value to be used as the input argument. The possible values for the constrant are the same as those used in the RTL, and are dependent on the configuration file used to built the GCC back end.

If there are no input operands, this argument may either be omitted, or explicitly given as `No_Input_Operands`. The fourth argument, not present in the above example, is a list of register names, called the *clobber* argument. This argument, if given, must be a static string expression, and is a space or comma separated list of names of registers that must be considered destroyed as a result of the `Asm` call. If this argument is the null string (the default value), then the code generator assumes that no additional registers are destroyed.

The fifth argument, not present in the above example, called the *volatile* argument, is by default `False`. It can be set to the literal value `True` to indicate to the code generator that all optimizations with respect to the instruction specified should be suppressed, and that in particular, for an instruction that has outputs, the instruction will still be generated, even if none of the outputs are used. See the full description in the GCC manual for further details.

The `Asm` subprograms may be used in two ways. First the procedure forms can be used anywhere a procedure call would be valid, and correspond to what the RM calls "intrinsic" routines. Such calls can be used to intersperse machine instructions with other Ada statements. Second, the function forms, which return a dummy value of the limited private type `Asm_Insn`, can be used in code statements, and indeed this is the only context where such calls are allowed. Code statements appear as aggregates of the form:

```
Asm_Insn'(Asm (...));
Asm_Insn'(Asm_Volatile (...));
```

In accordance with RM rules, such code statements are allowed only within subprograms whose entire body consists of such statements. It is not permissible to intermix such statements with other Ada statements.

Typically the form using intrinsic procedure calls is more convenient and more flexible. The code statement form is provided to meet the RM suggestion that such a facility should be made available. The following is the exact syntax of the call to `Asm` (of course if named notation is used, the arguments may be given in arbitrary order, following the normal rules for use of positional and named arguments)

```
ASM_CALL ::= Asm (
                 [Template =>] static_string_EXPRESSION
               [,[Outputs  =>] OUTPUT_OPERAND_LIST      ]
               [,[Inputs   =>] INPUT_OPERAND_LIST       ]
               [,[Clobber  =>] static_string_EXPRESSION ]
               [,[Volatile =>] static_boolean_EXPRESSION] )
OUTPUT_OPERAND_LIST ::=
  No_Output_Operands
| OUTPUT_OPERAND_ATTRIBUTE
| (OUTPUT_OPERAND_ATTRIBUTE {,OUTPUT_OPERAND_ATTRIBUTE})
OUTPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Output (static_string_EXPRESSION, NAME)
INPUT_OPERAND_LIST ::=
  No_Input_Operands
| INPUT_OPERAND_ATTRIBUTE
| (INPUT_OPERAND_ATTRIBUTE {,INPUT_OPERAND_ATTRIBUTE})
```

```
INPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Input (static_string_EXPRESSION, EXPRESSION)
```

9 Specialized Needs Annexes

Ada 95 defines a number of specialized needs annexes, which are not required in all implementations. However, as described in this chapter, GNAT implements all of these special needs annexes:

Systems Programming (Annex C)
: The systems programming annex is fully implemented.

Real-Time Systems (Annex D)
: The real-time systems annex is fully implemented.

Distributed Systems (Annex E)
: Stub generation is fully implemented, but no PCS is provided yet, so distributed systems cannot yet be constructed with this version of GNAT.

Information Systems (Annex F)
: The information systems annex is fully implemented.

Numerics (Annex G)
: The numerics annex is fully implemented.

Safety and Security (Annex H)
: The safety and security annex is fully implemented.

Obsolescent Features (Annex I)
: The obsolescent features annex is fully implemented.

Language Defined Attributes (Annex J)
: The language defined attributes annex is fully implemented.

Language Defined Pragmas (Annex K)
: The language defined pragmas annex is fully implemented.

10 Compatibility Guide

This chapter contains sections that describe compatibility issues between GNAT and other Ada 83 and Ada 95 compilation systems, to aid in porting applications developed in other Ada environments.

10.1 Compatibility with Ada 83

Ada 95 is designed to be highly upwards compatible with Ada 83. In particular, the design intention is that the difficulties associated with moving from Ada 83 to Ada 95 should be no greater than those that occur when moving from one Ada 83 system to another.

However, there are a number of points at which there are minor incompatibilities. The Ada 95 Annotated Reference Manual contains full details of these issues, and should be consulted for a complete treatment. In practice the following are the most likely issues to be encountered.

Character range
: The range of Standard.Character is now the full 256 characters of Latin-1, whereas in most Ada 83 implementations it was restricted to 128 characters. This may show up as compile time or runtime errors. The desirable fix is to adapt the program to accomodate the full character set, but in some cases it may be convenient to define a subtype or derived type of Character that covers only the restricted range.

New reserved words
: The identifiers `abstract`, `aliased`, `protected`, `requeue`, `tagged`, and `until` are reserved in Ada 95. Existing Ada 83 code using any of these identifiers must be edited to use some alternative name.

Freezing rules
: The rules in Ada 95 are slightly different with regard to the point at which entities are frozen, and representation pragmas and clauses are not permitted past the freeze point. This shows up most typically in the form of an error message complaining that a representation item appears too late, and the appropriate corrective action is to move the item nearer to the declaration of the entity to which it refers.

: A particular case is that representation pragmas (including the extended DEC Ada 83 compatibility pragmas such as Export_Procedure), cannot be applied to a subprogram body. If necessary, a separate subprogram declaration must be introduced to which the pragma can be applied.

Optional bodies for library packages
: In Ada 83, a package that did not require a package body was nevertheless allowed to have one. This lead to certain surprises in compiling large systems (situations in which the body could be unexpectedly ignored). In Ada 95, if a package does not require a body then it is not permitted to have a body. To fix this problem, simply remove a redundant body if it is empty, or, if it is non-empty, introduce a dummy declaration into the spec that makes the body

required. One approach is to add a private part to the package declaration (if necessary), and define a parameterless procedure called Requires_Body, which must then be given a dummy procedure body in the package body, which then becomes required.

Numeric_Error is now the same as Constraint_Error

In Ada 95, the exception Numeric_Error is a renaming of Constraint_Error. This means that it is illegal to have separate exception handlers for the two exceptions. The fix is simply to remove the handler for the Numeric_Error case (since even in Ada 83, a compiler was free to raise Constraint_Error in place of Numeric_Error in all cases).

Indefinite subtypes in generics

In Ada 83, it was permissible to pass an indefinite type (e.g. String) as the actual for a generic formal private type, but then the instantiation would be illegal if there were any instances of declarations of variables of this type in the generic body. In Ada 95, to avoid this clear violation of the contract model, the generic declaration clearly indicates whether or not such instantiations are permitted. If a generic formal parameter has explicit unknown discriminants, indicated by using (<>) after the type name, then it can be instantiated with indefinite types, but no variables can be declared of this type. Any attempt to declare a variable will result in an illegality at the time the generic is declared. If the (<>) notation is not used, then it is illegal to instantiate the generic with an indefinite type. This will show up as a compile time error, and the fix is usually simply to add the (<>) to the generic declaration.

All implementations of GNAT provide a switch that causes GNAT to operate in Ada 83 mode. In this mode, some but not all compatibility problems of the type described above are handled automatically. For example, the new Ada 95 protected keywords are not recognized in this mode. However, in practice, it is usually advisable to make the necessary modifications to the program to remove the need for using this switch.

10.2 Compatibility with Other Ada 95 Systems

Providing that programs avoid the use of implementation dependent and implementation defined features of Ada 95, as documented in the Ada 95 reference manual, there should be a high degree of portability between GNAT and other Ada 95 systems. The following are specific items which have proved troublesome in moving GNAT programs to other Ada 95 compilers, but do not affect porting code to GNAT.

Ada 83 Pragmas and Attributes

Ada 95 compilers are allowed, but not required, to implement the missing Ada 83 pragmas and attributes that are no longer defined in Ada 95. GNAT implements all such pragmas and attributes, eliminating this as a compatibility concern, but some other Ada 95 compilers reject these pragmas and attributes.

Special-needs Annexes

GNAT implements the full set of special needs annexes. At the current time, it is the only Ada 95 compiler to do so. This means that programs making use of these features may not be portable to other Ada 95 compilation systems.

Representation Clauses
: Some other Ada 95 compilers implement only the minimal set of representation clauses required by the Ada 95 reference manual. GNAT goes far beyond this minimal set, as described in the next section.

10.3 Representation Clauses

The Ada 83 reference manual was quite vague in describing both the minimal required implementation of representation clauses, and also their precise effects. The Ada 95 reference manual is much more explicit, but the minimal set of capabilities required in Ada 95 is quite limited.

GNAT implements the full required set of capabilities described in the Ada 95 reference manual, but also goes much beyond this, and in particular an effort has been made to be compatible with existing Ada 83 usage to the greatest extent possible.

A few cases exist in which Ada 83 compiler behavior is incompatible with requirements in the Ada 95 reference manual. These are instances of intentional or accidental dependence on specific implementation dependent characteristics of these Ada 83 compilers. The following is a list of the cases most likely to arise in existing legacy Ada 83 code.

Implicit Packing
: Some Ada 83 compilers allowed a Size specification to cause implicit packing of an array or record. This is specifically disallowed by implementation advice in the Ada 83 reference manual (for good reason, this usage can cause expensive implicit conversions to occur in the code). The problem will show up as an error message rejecting the size clause. The fix is simply to provide the explicit pragma Pack.

Meaning of Size Attribute
: The Size attribute in Ada 95 for discrete types is defined as being the minimal number of bits required to hold values of the type. For example, on a 32-bit machine, the size of Natural will typically be 31 and not 32 (since no sign bit is required). Some Ada 83 compilers gave 31, and some 32 in this situation. This problem will usually show up as a compile time error, but not always. It is a good idea to check all uses of the 'Size attribute when porting Ada 83 code. The GNAT specific attribute Object_Size can provide a useful way of duplicating the behavior of some Ada 83 compiler systems.

Size of Access Types
: A common assumption in Ada 83 code is that an access type is in fact a pointer, and that therefore it will be the same size as a System.Address value. This assumption is true for GNAT in most cases with one exception. For the case of a pointer to an unconstrained array type (where the bounds may vary from one value of the access type to another), the default is to use a `"fat pointer"`, which is represented as two separate pointers, one to the bounds, and one to the array. This representation has a number of advantages, including improved efficiency. However, it may cause some difficulties in porting existing Ada 83 code which makes the assumption that, for example, pointers fit in 32 bits on a machine with 32-bit addressing.

To get around this problem, GNAT also permits the use of "thin pointers" for access types in this case (where the designated type is an unconstrained array type). These thin pointers are indeed the same size as a System.Address value. To specify a thin pointer, use a size clause for the type, for example:

```
type X is access all String;
for X'Size use System.Address'Size;
```

which will cause the type X to be represented using a single pointer. When using this representation, the bounds are right behind the array. This representation is slightly less efficient, and does not allow quite such flexibility in the use of foreign pointers or in using the Unrestricted_Access attribute to create pointers to non-aliased objects. But for any standard portable use of the access type it will work in a functionally correct manner and allow porting of existing code. Note that another way of forcing a thin pointer representation is to use a component size clause for the element size in an array, or a record representation clause for an access field in a record.

10.4 Compatibility with DEC Ada 83

The VMS version of GNAT fully implements all the pragmas and attributes provided by DEC Ada 83, as well as providing the standard DEC Ada 83 libraries, including Starlet. In addition, data layouts and parameter passing conventions are highly compatible. This means that porting existing DEC Ada 83 code to GNAT in VMS systems should be easier than most other porting efforts. The following are some of the most significant differences between GNAT and DEC Ada 83.

Default floating-point representation
: In GNAT, the default floating-point format is IEEE, whereas in DEC Ada 83, it is VMS format. GNAT does implement the necessary pragmas (Long_Float, Float_Representation) for changing this default.

System
: The package System in GNAT exactly corresponds to the definition in the Ada 95 reference manual, which means that it excludes many of the DEC Ada 83 extensions. However, a separate package Aux_DEC is provided that contains the additional definitions, and a special pragma, Extend_System allows this package to be treated transparently as an extension of package System.

Task_Id values
: The Task_Id values assigned will be different in the two systems, and GNAT does not provide a specified value for the Task_Id of the environment task, which in GNAT is treated like any other declared task.

For full details on these and other less significant compatibility issues, see appendix E of the Digital publication entitiled "DEC Ada, Technical Overview and Comparison on DIGITAL Platforms".

For GNAT running on other than VMS systems, all the DEC Ada 83 pragmas and attributes are recognized, although only a subset of them can sensibly be implemented. The description of pragmas in this reference manual indicates whether or not they are applicable to non-VMS systems.

Index

(Index is nonexistent)

Table of Contents