Ada 95 Reference Manual Supplement

For mission-critical applications

www.xgc.com

Ada 95 Reference Manual Supplement

For mission-critical applications

Order Number: XGC-ADA-RMS-081201

XGC Technology

London
UK
<Web: www.xgc.com>

Ada 95 Reference Manual Supplement: For mission-critical applications

by Chris Nettleton

Publication date December 1, 2008

License

Acknowledgements

XGC Ada is based on technology originally developed by the GNAT team at New York University and now maintained by Ada Core Technologies, Inc., and includes software from the GNU C compiler, debugger and binary utilities developed by and on behalf of the Free Software Foundation, Inc., Cambridge, Massachusetts.

In this manual, the chapters on advice, attributes, characteristics, compatibility, and pragmas are based on text from the GNAT Reference Manual, Version 3.11 and later.

Contents

Tables

About This Manual

This supplement should be read in conjunction with the *Ada 95 Reference Manual* (RM). It describes the differences between the complete Ada 95 programming language, and the mission-critical subset supported by the XGC range of cross compilers. It also includes information on implementation-dependent characteristics of XGC Ada, including all the information required by Annex M of the Reference Manual.

1. Audience

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

2. Related Documents

See the following documents for further information on XGC Ada:

- *The `target` Ada Technical Summary* ,which includes target-dependent information.

- *Getting Started with `target` Ada* describes the steps required to prepare and run a simple program.
- *XGC Ada User's Guide*, which provides information on how to use the XGC Ada compiler system.
- Ada 95 Reference Manual, ANSI/ISO/IEC-8652:1995, which contains all reference material for the Ada 95 programming language.
- The *XGC Libraries* documents the library functions available with all XGC compilers.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC user manuals.

You can send your comments as follows:

- Internet electronic mail: readers_comments@xgc.com

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

Technical support enquiries should be directed to the XGC web site, http://www.xgc.com/ or by email to support@xgc.com.

4. Documentation Conventions

This guide uses the following typographic conventions:

`$`
: A dollar sign represents the system prompt for the Bash shell.

`#`
: A number sign represents the superuser prompt.

`$vi hello.c`
: Boldface type in interactive examples indicates typed user input.

`file`
: Italic or slanted type indicates variable values, place-holders, and function argument names.

[|], { | }
: In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...
: In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

cat(1)
: A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Mb/s
: This symbol indicates megabits per second.

MB/s
: This symbol indicates megabytes per second.

Ctrl+**x**
: This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows. In examples, this key combination is printed in bold type (for example, **Ctrl**+**C**).

Chapter 1

Implementation-Defined Pragmas

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

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

pragma Ada_83

Ada_83

Synopsis

```
pragma Ada_83;
```

Description

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, XGC Ada 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 XGC Ada 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 XGC Ada in Ada 83 mode will in fact compile and execute with an Ada 83 compiler, since XGC Ada does not enforce all the additional checks required by Ada 83.

pragma Ada_95

Ada_95

Synopsis

```
pragma Ada_95;
```

Description

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

Annotate

Synopsis

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

                    ARG ::= NAME | EXPRESSION
```

Description

This pragma is used to annotate programs. *`identifier`* identifies the type of annotation. XGC Ada verifies this is an identifier, but does not otherwise analyze it. The *`arg`* argument 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 XGC Ada compiler. This pragma is intended for use by external tools, including ASIS.

pragma Assert

Assert

Synopsis

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

Description

The effect of this pragma depends on whether the corresponding command line switch is set to activate assertions. If assertions are disabled, 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_EXPRES
                        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 C_Pass_By_Copy

C_Pass_By_Copy

Synopsis

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

Description

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, that is 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

Common_Object

Synopsis

```
pragma Common_Object
                    [Internal =>] LOCAL_NAME,
```

```
                    [, [External =>] EXTERNAL_SYMBOL,
                    [, [Size     =>] EXTERNAL_SYMBOL]

                    EXTERNAL_SYMBOL ::=
                    IDENTIFIER
                    | static_string_EXPRESSION
```

Description

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 XGC Ada, but otherwise ignored.

pragma Complex_Representation

Complex_Representation

Synopsis

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

Description

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 the compiler 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

Component_Alignment

Synopsis

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

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

Description

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.

`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, that is 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 underlying hardware or operating system or both.

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 representation clause), the XGC Ada uses the default alignment as described previously.

pragma Debug

Debug

Synopsis

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

Description

If assertions are not enabled on the command line, this pragma has no effect. If assertions 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 Export_Function

Export_Function

Synopsis

```
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
```

Description

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`. XGC Ada 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.

pragma Export_Object

Export_Object

Synopsis

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

                   EXTERNAL_SYMBOL ::=
                   IDENTIFIER
                   | static_string_EXPRESSION
```

Description

This pragma designates an object as exported, and apart from the extended rules for external 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 XGC Ada.

pragma Export_Procedure

Export_Procedure

Synopsis

```
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
```

Description

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

Export_Valued_Procedure

Synopsis

```
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
```

Description

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. XGC Ada 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 Ident

Ident

Synopsis

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

Description

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.

pragma Import_Function

Import_Function

Synopsis

```
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
```

Description

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.

pragma Import_Object

Import_Object

Synopsis

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

                    EXTERNAL_SYMBOL ::=
                    IDENTIFIER
                    | static_string_EXPRESSION
```

Description

This pragma designates an object as imported, and apart from the extended rules for external 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 XGC Ada.

pragma Import_Procedure

Import_Procedure

Synopsis

```
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
```

Description

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

Import_Valued_Procedure

Synopsis

```
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
```

Description

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 Interface_Name

Interface_Name

Synopsis

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

Description

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

Linker_Alias

Synopsis

```
pragma Linker_Alias (
                    [Entity =>] LOCAL_NAME
```

```
                    [Alias  =>] static_string_EXPRESSION);
```

Description

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

pragma Linker_Section

Linker_Section

Synopsis

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

Description

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

pragma Normalize_Scalars

Normalize_Scalars

Synopsis

```
pragma Normalize_Scalars;
```

Description

This is a language defined pragma which is fully implemented in XGC Ada. 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 type'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, that is 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 Machine_Attribute

Machine_Attribute

Synopsis

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

Description

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 No_Return

No_Return

Synopsis

```
pragma No_Return (procedure_Local_Name);
```

Description

`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 Profile

Profile

Synopsis

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

                    IDENTIFIER ::=
                    XGC | Ravenscar | Restricted_Run_Time | No_Run_Time
```

Description

This pragma specifies a restriction profile. It is a configuration pragma, and so has the usual applicability of configuration pragmas (that is 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. See Appendix A, *Restrictions and Profiles* [99] .

pragma Psect_Object

Psect_Object

Synopsis

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

                    EXTERNAL_SYMBOL ::=
                    IDENTIFIER
                    | static_string_EXPRESSION
```

Description

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

pragma Pure_Function

Pure_Function

Synopsis

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

Description

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 re-computation).

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

Share_Generic

Synopsis

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

Description

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

pragma Source_File_Name

Source_File_Name

Synopsis

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

                    FNAME_DESIG => Body_File_Name | Spec_File_Name
```

Description

Use this to override the normal naming convention. It is a configuration pragma, and so has the usual applicability of configuration pragmas (that is 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

Source_Reference

Synopsis

```
pragma Source_Reference (INTEGER_Literal, STRING_Literal);
```

Description

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 Subtitle

Subtitle

Synopsis

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

Description

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

pragma Suppress_All

Suppress_All

Synopsis

```
pragma Suppress_All;
```

Description

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 XGC Ada.

pragma Title

Title

Synopsis

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

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

Description

Syntax checked but otherwise ignored by XGC Ada. 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 XGC Ada does not have titles or subtitles.

Unlike other pragmas, the full flexibility of named notation is allowed for this pragma, that is 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

Unchecked_Union

Synopsis

```
pragma Unchecked_Union (first_subtype_LOCAL_NAME)
```

Description

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 XGC Ada, a direct comparison is illegal, but XGC Ada 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, that is 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

Unimplemented_Unit

Synopsis

```
pragma Unimplemented_Unit;
```

Description

If this pragma occurs in a unit that is processed by the compiler, XGC Ada 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 Unsuppress

Unsuppress

Synopsis

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

Description

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 Warnings

Warnings

Synopsis

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

Description

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

Weak_External

Synopsis

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

Description

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.

Chapter 2

Implementation-Defined Attributes

The Ada 95 Reference Manual defines a set of attributes that provide useful additional functionality in all areas of the language. These language defined attributes are implemented in XGC Ada and work as described in the Manual.

In addition, Ada 95 allows implementations to define additional attributes whose meaning is defined by the implementation. XGC Ada 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 reference manual describes these additional attributes.

`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.

`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.

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. XGC Ada 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, for example 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, for example 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 Univers
```

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 (that is 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)
```

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.

`Integer_Value`

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

```
function S'Integer_Value (Arg : Universal_Fixed) retur
```

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 other Ada compilers.

`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 meaning the first parameter) of *`subprogram`*. The code returned is 1 for by copy and 2 for by reference.

`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 XGC Ada 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 95 Reference Manual, 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 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 a type that 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.

`Unrestricted_Access`

The `Unrestricted_Access` attribute is similar to `Access` except that all accessibility and aliased view checks are omitted. This is very much a user-beware attribute. It is very 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 XGC Ada, 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).

`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`.

Chapter 3

Implementation Advice

The main text of the Ada 95 Reference Manual describes the required behavior of all Ada 95 compilers, and subject to the restrictions described in Appendix A, *Restrictions and Profiles* [99] ,the XGC Ada 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, that is they do not specify requirements that compilers must follow. Rather they provide advice on generally desirable behavior.

Using a question and answer format, this chapter gives the reference manual section number, paragraph number and several keywords for each piece of advice. Each entry consists of the text of the advice followed by the XGC Ada interpretation of this advice. Most often, this simply says “followed”, which means that XGC Ada follows the advice. However, in a number of cases, XGC Ada deliberately deviates from this advice, in which case the text describes what XGC Ada does and why.

3.1. Section 1: General

1.1.3(20): Error Detection

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.

A:

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.

A:

Followed.

1.1.5(12): Bounded Errors

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

A:

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

3.2. Section 2: Lexical Elements

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.

A:

The following implementation defined pragmas are exceptions to this rule:

`Ada_83`
: Affects legality

`Assert`
: Affects semantics

`Debug`
: Affects semantics

`Interface_Name`
: Affects semantics

`Machine_Attribute`
: Affects semantics

`Unimplemented_Unit`
: Affects legality

`Unchecked_Union`
: Affects semantics

Not followed. In each of the above cases, it is essential to the purpose of the pragma that this advice not be followed. For details see Chapter 1, *Implementation-Defined Pragmas* [1] .

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`.

A: See response to paragraph 16 of this same section.

3.3. Section 3: Declarations and Types

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`.

A: 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).

A: `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 non-binary modules up to `Integer'Last`.

A: *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

function 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 non-contiguous internal codes specified by an enumeration_representation_clause.

A: *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).

A: `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.

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).

A: *Followed.*

3.4. Section 9: Tasking

9.6(30-31): Duration'Small

9.6(30-31): Duration'Small	Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds.
A:	*Followed*. Duration'Small is one microsecond.
(continued)	The time base for delay_relative_statements should be monotonic; it need not be the same time base as used for Calendar.Clock.
A:	*Not applicable*. delay_relative_statements are prohibited by the built-in restriction No_Relative_Delay.

3.5. Section 10: Program Structure and Compilation Issues

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.
A:	*Followed*.

3.6. Section 11: Exceptions

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`.
A:	*Followed for Exception_Message*. Exception_Information is not supported.
11.5(28): Suppression of Checks	The implementation should minimize the code executed for checks that have been suppressed.
A:	*Followed*.

3.7. Section 13: Representation Issues

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.

A: *Followed*. XGC Ada does not support non-static expressions in representation clauses unless they are constants declared before the entity.

(continued) 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.

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

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

A: *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 implementation 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.

A: *Partly followed*. Tight packing of arrays is supported for component sizes of 1, 2, 4, 8, 16 and 32 bits.

(continued) An implementation should support Address clauses for imported subprograms.

A: *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.

A: *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.

(continued)	An implementation should support `Address` clauses for imported subprograms.
A:	*Followed.*
(continued)	Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries.
A:	*Followed.*
(continued)	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.
A:	*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:
A:	*Followed.*
(continued)	An implementation need not support specified `Alignments` for combinations of `Sizes` and `Alignments` that cannot be easily loaded and stored by available machine instructions.
A:	*Followed.*
(continued)	An implementation need not support specified `Alignments` that are greater than the maximum `Alignment` the implementation ever returns by default.
A:	*Followed.*

	The recommended level of support for the `Alignment` attribute for objects is:
(continued)	Same as above, for subtypes, but in addition:
A:	*Followed*.
(continued)	For stand-alone library-level objects of statically constrained subtypes, the implementation should support all `Alignments` supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.
A:	*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).
A:	*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).
A:	*Followed*.
(continued)	`Size` clause on a composite subtype should not affect the internal layout of components.
A:	*Followed*.

(continued)	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.
A:	*Followed.*
(continued)	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.
A:	*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.
A:	*Followed.*
(continued)	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.
A:	*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 minimum support the internal codes in the range `System.Min_Int.System.Max_Int`.

A: *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.

A: *Followed.*

(continued)

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.

A: *Followed.*

(continued)

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.

A: *Followed.*

(continued)

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

A: *Followed.*

(continued)	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.
A:	*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.
A:	*Followed*. There are no such components in XGC Ada.
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.
A:	*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.
A:	*Not Followed*. The type Address is visible.
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`.

A: *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.

A: *Followed*.

(continued) 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.

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

(continued) 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.
A:	*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.
A:	*Followed*, the only other points at which heap storage is dynamically allocated are as follows: • To allocate space for a task when a task is created.
(continued)	A default (implementation-provided) storage pool for an access-to- constant type should not have overhead to support de-allocation of individual objects.
A:	*Not applicable*.
(continued)	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.
A:	*Followed*.
13.11.2(17): Unchecked De-allocation	For a standard storage pool, `Free` should actually reclaim the storage.
A:	*Not supported*.
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.

A: *Not supported.*

3.8. Annex A: Predefined Language Environment

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.

A:	*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.
A:	*Followed*. XGC Ada provides no such localized definitions.
A.4.5(106): Bounded-Length String Handling	Bounded string objects should not be implemented by implicit pointers and dynamic allocation.
A:	*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.
A:	*Followed*.
(continued)	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.
A:	*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`.

A: *Followed.*

3.9. Annex B: Interface to Other Languages

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.

A: *Followed.*

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

A: *Followed.*

(continued) 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.

A: *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`.

A: *Followed.*

(continued)

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

A: *Not Followed.*

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

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

A: *Followed.*

(continued)

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

A: *Followed.*

(continued)

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

A: *Followed.*

(continued)

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

A: *Followed.*

(continued)

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*.
A:	*Followed*.
(continued)	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.
A:	*Followed*.
(continued)	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*.
A:	*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.
(continued)	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*.
A:	*Followed*.
(continued)	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.
A:	*Followed*.

B.4(95-98): Interfacing with COBOL An Ada implementation should support the following interface correspondences between Ada and COBOL.

A: *Not Followed*. COBOL is not supported by XGC Ada.

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

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

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.

A: *Not applicable*. COBOL is not supported by XGC Ada.

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.

A: *Followed*.

(continued) An Ada function corresponds to a FORTRAN function.

A: *Followed*.

(continued) 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.
A:	*Followed*.
(continued)	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.
A:	*Followed*.

3.10. Annex C: Systems Programming

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.

A:

Followed.

(continued)

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

A:	*Followed*.
(continued)	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.
A:	*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.
A:	*Followed* for both intrinsics and machine-code subprograms.
(continued)	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.
A:	*Followed*. A full set of machine operation intrinsic subprograms is provided.
(continued)	Atomic read-modify-write operations -- e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.
A:	*Followed* on any target supporting such operations.
(continued)	Standard numeric functions -- e.g., sin, log.
A:	*Followed* on any target supporting such operations.

(continued)	String manipulation operations -- e.g., translate and test.
A:	*Followed* on any target supporting such operations.
(continued)	Vector operations -- e.g., compare vector against thresholds.
A:	*Followed* on any target supporting such operations.
(continued)	Direct operations on I/O ports.
A:	*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.
A:	*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.
A:	*Followed* on any target where the underlying operating system permits such direct calls.
(continued)	Whenever practical, violations of any implementation-defined restrictions should be detected before run time.
A:	*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.
A:	*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.
A:	*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.
A:	*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.
A:	*Not followed*. This implementation is not targeted to such a domain.

3.11. Annex D: Real-Time Systems

D.3(17): Locking Policies

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

A:

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.

A:

Followed. No such implementation-defined queuing 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.
A:	*Not applicable*.
(continued)	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.
A:	*Not applicable*.
D.7(21): Tasking Restrictions	When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation.
A:	*Followed*.
D.8(47-49): Monotonic Time	When appropriate, implementations should provide configuration mechanisms to change the value of `Tick`.
A:	*Not Followed*.
(continued)	It is recommended that `Calendar.Clock` and `Real_Time.Clock` be implemented as transformations of the same time base.
A:	*Not Followed*. Package Calendar is prohibited by the built-in restriction No_Calendar.
(continued)	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.
A:	*Followed*.

3.12. Annex E: Distributed Systems

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.

A:

Not applicable.

(continued)

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.

A:

Not applicable.

3.13. Annex F: Information Systems

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.

A: *Not applicable*. COBOL is not supported by XGC Ada.

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.

A: *Not followed*. XGC Ada ignores *S*'Machine_Radix and always uses binary representations.

3.14. Annex G: Numerics

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.

A:

Not applicable.

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.

A: *Not followed.*

(continued) 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.

A: *Not followed.*

(continued) 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.

A: *Not applicable.*

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.

A: *Not applicable.*

G.2.4(19): Accuracy Requirements

The versions of the forward trigonometric functions 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. 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`.

A:

Not applicable.

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.

A:

Not applicable.

Chapter 4

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 in Ada 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 constraint 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)
```

4.1. Constraints for Operands

Here are specific details on what constraint letters you can use with Asm statement operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

4.1.1. Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

“ m ”

A memory operand is allowed, with any kind of address that the target computer supports in general.

“ o ”

A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an auto-increment or auto-decrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter “ o ” is valid only when accompanied by both “ < ” (if the target machine has pre-decrement addressing) and “ > ” (if the target machine has pre-increment addressing).

“ V ”

A memory operand that is not offsettable. In other words, anything that would fit the “ m ” constraint but not the “ o ” constraint.

“ < ”

A memory operand with auto-decrement addressing (either pre-decrement or post-decrement) is allowed.

“ > ”

A memory operand with auto-increment addressing (either pre-increment or post-increment) is allowed.

“ r ”

A register operand is allowed provided that it is in a general register.

“ d ” , “ a ” , “ f ” ,...
Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. “ d ” , “ a ” and “ f ” are defined on the 68000/68020 to stand for data, address and floating point registers.

“ i ”

An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

“ n ”

An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use “ n ” rather than “ i ” .

“ I ” , “ J ” , “ K ” ,... “ P ”

Other letters in the range “ I ” through “ P ” may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, “ I ” is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

“ E ”

An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

“ F ”

An immediate floating operand (expression code const_double) is allowed.

“ G ” , “ H ”

“ G ” and “ H ” may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

“ s ”

An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use “ s ” instead of “ i ” ?Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a “ moveq ” instruction. We arrange for this to happen by defining the letter “ K ” to mean “any integer outside the range -128 to 127 ” ,and then specifying “ Ks ” in the operand constraints.

“ g ”

Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

“ X ”

Any operand whatsoever is allowed.

“ 0 ” , “ 1 ” , “ 2 ” ,... “ 9 ”

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which `asm` distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

“ `p` ”

An operand that is a valid memory address is allowed. This is for “load address” and “push address” instructions.

“ `p` ” in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

“ `Q` ” , “ `R` ” , “ `S` ” ,... “ `U` ”

Letters in the range “ `Q` ” through “ `U` ” may be defined in a machine-dependent fashion to stand for arbitrary operand types.

4.1.2. Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the " ? " and " ! " characters:

?

Disparage slightly the alternative that the " ? " appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each " ? " that appears in it.

!

Disparage severely the alternative that the " ! " appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

4.1.3. Constraint Modifier Characters

Here are constraint modifier characters.

" = "

Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

" + "

Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. " = " identifies an output; " + " identifies an operand that is both input and output; all other operands are assumed to be input only.

" & "

Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is modified before the instruction

is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

“ & ” applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires “ & ” while others do not. See, for example, the “ movdf ” insn of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows the compiler to produce better code when only some of the inputs can be affected by the earlyclobber.

“ & ” does not obviate the need to write “ = ” .

“ % ”

Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints.

“ # ”

Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

Chapter 5

Compatibility Guide

This chapter contains sections that describe compatibility issues between XGC Ada and other Ada 83 and Ada 95 compilation systems, to aid in porting applications developed in other Ada environments.

5.1. 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

accommodate 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.

The compiler provides a switch that causes XGC Ada 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.

5.2. Compatibility with Other Ada 95 Systems

Providing that programs avoid the use of restricted, implementation dependent and implementation defined features of XGC Ada, there should be a high degree of portability between XGC Ada and other Ada 95 systems. The following are specific items which have proved troublesome in moving XGC Ada programs to other Ada 95 compilers, but do not affect porting code to XGC Ada.

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. XGC Ada 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
: XGC Ada implements the a restricted set of special needs annexes. Other Ada compilers may support a different set and that means that 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. XGC Ada goes far beyond this minimal set, as described in the next section.

5.3. 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.

XGC Ada 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 XGC Ada 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 XGC Ada 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, XGC Ada 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.

Appendix A

Restrictions and Profiles

This Appendix defines how the Ada 95 restrictions, accessible through the pragma Restrictions, are supported. Unsafe features such as run-time dispatching and heap management are not supported in the run-time system, so all the restrictions that are relevant for these features are set to True by default.

The following restrictions are built in. That is, they cannot be turned off and are exploited by the compiler to offer better-quality generated code than would otherwise be possible.

- No_Abort_Statements
- No_Dispatch
- No_Local_Protected_Objects
- No_Requeue
- No_Task_Attributes
- No_Task_Hierarchy
- No_Terminate_Alternatives

The implementation-defined pragma Profile may also be used to set and unset restrictions that correspond to a certain application area. The profiles supported are as follows:

Table A.1. Supported Profiles

Profile Name	Description
XGC	This is the default profile and offers the least restrictions.
Ravenscar	This allows a limited form of tasking that includes static tasks, protected objects, the delay until statement and interrupts.
Restricted_Run_Time	This severely restricts the use of non-deterministic language features (including tasking) and is suitable for general avionics applications.
No_Run_Time	This profile prohibits all calls to the predefined Ada library and is useful for safety-critical applications. Calls to the compiler support library are not restricted.

Table A.2, “Profiles and Restrictions” [100] gives the individual restrictions for each profile. Note that the built-in restrictions apply to all profiles.

Table A.2. Profiles and Restrictions

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_ Run_Time
Boolean_Entry_Barriers	XGC (Ravenscar)	False	True	True
Immediate_Reclamation	RM H.4(10)	False	False	False
No_Abort_Statements	RM D.7(5), H.4(3)	True	True	True
No_Access_Subprograms	RM H.4(17)	False	True	True
No_Allocators	RM H.4(7)	False	False	True
No_Asynchronous_Control	RM D.9(10)	False	True	True
No_Calendar	XGC	False	True	True
No_Delay	RM H.4(21)	False	False	True
No_Dispatch	RM H.4(19)	True	True	True
No_Dynamic_Interrupts	XGC	True	True	True
No_Dynamic_Priorities	RM D.9(9)	False	True	True

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_Run_Time
No_Elaboration_Code	XGC	False	False	True
No_Entry_Calls_In_Elaboration_Code	XGC	False	True	True
No_Entry_Queue	XGC	True	True	True
No_Enumeration_Maps	XGC	False	False	True
No_Exception_Handlers	XGC	False	False	True
No_Exceptions	RM H.4(12)	False	False	False
No_Fixed_Point	RM H.4(15)	False	False	False
No_Floating_Point	RM H.4(14)	False	False	False
No_Implementation_Attributes	XGC	False	False	True
No_Implementation_Pragmas	XGC	False	False	True
No_Implementation_Restrictions	XGC	False	False	True
No_Implicit_Conditionals	XGC	False	False	True
No_Implicit_Heap_Allocations	RM D.8(8), H.4(3)	False	True	True
No_Implicit_Loops	XGC	False	False	False
No_IO	RM H.4(20)	False	True	True
No_Local_Allocators	RM H.4(8)	False	True	True
No_Local_Protected_Objects	XGC	True	True	True
No_Nested_Finalization	RM D.7(4)	True	True	True
No_Protected_Type_Allocators	XGC	True	True	True
No_Protected_Types	RM H.4(5)	False	False	True
No_Recursion	RM H.4(22)	False	True	True
No_Reentrancy	RM H.4(23)	False	False	False
No_Relative_Delay	XGC	False	True	True
No_Requeue	XGC	True	True	True
No_Select_Statements	XGC (Ravenscar)	False	True	True
No_Standard_Storage_Pools	XGC	True	True	True
No_Streams	XGC	True	True	True
No_Task_Allocators	RM D.7(7)	False	True	True
No_Task_Attributes	XGC	True	True	True
No_Task_Hierarchy	RM D.7(3), H.4(3)	True	True	True
No_Task_Termination	XGC	True	True	True
No_Terminate_Alternatives	RM D.7(6)	True	True	True

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_Run_Time
No_Unchecked_Access	RM H.4(18)	False	True	True
No_Unchecked_Conversion	RM H.4(16)	False	False	True
No_Unchecked_Deallocation	RM H.4(9)	True	True	True
No_Wide_Characters	XGC	False	True	True
Static_Priorities	XGC	False	True	True
Static_Storage_Size	XGC	False	True	True

Table A.3, “Profiles and Numerical Restrictions” [102] gives the restrictions concerning numerical limits.

Table A.3. Profiles and Numerical Restrictions

Restriction	*Ada 95 Reference Manual* Section	Default	Ravenscar	Restricted_Run_Time
Max_Asynchronous_Select_Nesting	RM D.7(18), H.4(2)	0	0	0
Max_Protected_Entries	RM D.7(14)	1	1	1
Max_Select_Alternatives	RM D.7(12)	Undefined	0	0
Max_Storage_At_Blocking	RM D.7(17)	0	0	0
Max_Task_Entries	RM D.7(13), H.4(2)	Undefined	0	0
Max_Tasks	RM D.7(19), H.4(2)	Undefined	Undefined	Undefined
Max_Entry_Queue_Depth	Ravenscar specific	1	1	1

Violation of the restriction Max_Entry_Queue_Depth is detected at run time and raises the predefined exception Program_Error.

Appendix B

The Predefined Library

This appendix lists the units in the Ada 95 predefined library, and indicates whether a unit is supported or not. The answer “Yes” means the unit is supported in the default profile, and maybe in the other profiles. The answer “Restricted...” means the unit is not supported in any profile because of a built-in restriction.

Table B.1. Predefined Library Units

Unit Name	Supported?
Ada	Yes
Ada.Asynchronous_Task_Control	Yes
Ada.Calendar	Yes [a] [b]
Ada.Characters	Yes
Ada.Characters.Handling	Yes
Ada.Characters.Latin_1	Yes
Ada.Characters.Wide_Latin_1	Yes
Ada.Command_Line	Yes
Ada.Decimal	Yes
Ada.Direct_IO	Yes[b]
Ada.Dynamic_Priorities	Yes

Unit Name	Supported?
Ada.Exceptions	Yes
Ada.Finalization	Restricted No_Implicit_Heap_Allocations
Ada.Interrupts	Yes
Ada.Interrupts.Names	Yes
Ada.IO_Exceptions	Yes
Ada.Numerics	Yes
Ada.Numerics.Complex_Elementary_Functions	Yes
Ada.Numerics.Complex_Types	Yes
Ada.Numerics.Discrete_Random	Yes
Ada.Numerics.Elementary_Functions	Yes
Ada.Numerics.Float_Random	Yes
Ada.Numerics.Generic_Complex_Elementary_Functions	Yes
Ada.Numerics.Generic_Complex_Types	Yes
Ada.Numerics.Generic_Elementary_Functions	Yes
Ada.Real_Time	Yes
Ada.Sequential_IO	Yes[b]
Ada.Storage_IO	Yes
Ada.Streams	Restricted No_Dispatch
Ada.Streams.Stream_IO	Restricted No_Dispatch
Ada.Strings	Yes
Ada.Strings.Bounded	Yes
Ada.Strings.Fixed	Yes
Ada.Strings.Maps	Yes
Ada.Strings.Maps.Constants	Yes
Ada.Strings.Unbounded	Not available
Ada.Strings.Wide_Bounded	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Fixed	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Maps	Restricted No_Implicit_Heap_Allocations
Ada.Strings.Wide_Maps.Wide_Constants	Restricted No_Implicit_Heap_Allocations

Unit Name	Supported?
Ada.Strings.Wide_Unbounded	Restricted No_Implicit_Heap_Allocations
Ada.Synchronous_Task_Control	Yes
Ada.Tags	Restricted No_Dispatch
Ada.Task_Attributes	No
Ada.Task_Identification	Yes
Ada.Text_IO	Yes[b]
Ada.Text_IO.Complex_IO	Not applicable
Ada.Text_IO.Editing	Not applicable
Ada.Text_IO.Text_Streams	Not applicable
Ada.Unchecked_Conversion	Yes
Ada.Unchecked_Deallocation	Restricted No_Unchecked_Deallocation
Ada.Wide_Text_IO	Not applicable
Ada.Wide_Text_IO.Complex_IO	Not applicable
Ada.Wide_Text_IO.Editing	Not applicable
Ada.Wide_Text_IO.Text_Streams	Not applicable
Calendar	Yes[ab]
Direct_IO	Yes[b]
IO_Exceptions	Yes
Interfaces	Yes
Interfaces.C	Yes
Interfaces.C.Pointers	Yes
Interfaces.C.Strings	Yes
Interfaces.COBOL	Not applicable
Interfaces.FORTRAN	Not applicable
Machine_Code	Yes
Sequential_IO	Yes[b]
System	Yes
System.Address_to_Access_Conversions	Yes
System.Machine_Code	Yes
System.RPC	Not available (depends on Ada.Streams)

Unit Name	Supported?
System.Storage_Elements	Yes
System.Storage_Pools	Not available (depends on Ada.Finalization)
Text_IO	Yes
Unchecked_Conversion	Yes
Unchecked_Deallocation	Restricted No_Unchecked_Deallocation

[a]Restricted to POSIX date range, which is Jan 1, 1970 to Jan 19, 2038

[b]When supported by appropriate system calls

Index