SPARCompiler Ada Programmer's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-4862-11
Revision B, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

VADS, VADScross, and Verdix are registered trademarks of Rational Software Corporation (formerly Verdix).

The OPEN LOOK and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Contents

Figures

Tables

"An annotator has his scruples tool."

Wallace Stevens

Ada Formatter 1

SPARCompiler Ada includes the Ada source code formatter `a.pr` ("pretty printer"), which writes to the standard output and is easily redirected to a file. Hereafter, SPARCompiler Ada is called SC Ada.

A range of options can tailor `a.pr` for individual coding standards.

`a.error` and `a.list` provide program listings without formatting. `a.error` lists programs containing errors and intersperses the compiler error messages among the source lines. `a.list` produces a program listing with or without line numbers — but with no other additional formatting — for programs containing no errors.

References

`a.error` and `a.list`, *SPARCompiler Ada Reference Guide*

1.1 Invoking `a.pr`

This command syntax invokes source code formatter:

```
a.pr [options] [ada_source_file]
```

`a.pr` reads as input the specified file or standard input. Specify options on the command line or in the runtime configuration file `.prrc`, located in the user's current or home directory. An option specified on the command line takes precedence over options specified in the `.prrc` file.

This chapter lists options for the command line and configuration file later.

1.2 Default Format Specifications

If no options are given, `a.pr` prints the source file as follows:

- Reserved words are in lower case.
- Identifiers are in upper case.
- Comments remain as given.
- The outer-most level begins in the first column.
- Each subsequent level is indented 8 columns.
- Indentation is performed with tabs rather than spaces.
- A formatted line of source, including comment and indentation, does not extend past the 132nd column without continuing on the next line.
- A maximum of 55 lines is allowed on a page; a new page starts if a program unit (package, subprogram, task, and so forth) does not fit on the current page and it is split among the necessary number of pages.
- Pagination is performed with form feeds rather than blank lines.
- Record type declarations and type representations have the word `record` on the same line as the word `type` or the word `for`.
- When comments align to the right of the Ada source code, comments align 4 spaces to the right of the longest source line containing a comment.

1.3 Fixed-format Specifications

In addition to the format specifications listed above, which are changed by command line or `.prrc` commands, an Ada source file is formatted in the following manner.

- Each Ada statement prints on its own line.
- Any blank lines in the input file produce blank lines in the output except those falling at the beginning of a page other than the first.
- A subprogram or `task` body, `package` specification, body statement or `accept` statement ends with its identifier name.
- `pragma PAGE` is recognized, and the following code starts on a new page.

1.4 Error and Warning Messages

If errors are encountered in the `.prrc` file, an error message prints and `a.pr` quits, leaving the source file unformatted.

All error and warning messages print to `standard error`. To prevent warnings from appearing in the formatted source, use the `-nw` command line option or `set no_warning` in the `.prrc` file.

To place the warning/error messages in a file separate from the formatted source, use the following from `csh(1)`.

```
(a.pr [options] ada_source.a > output_file) >& error_file
```

or the following from `sh(1)`.

```
a.pr [options] ada_source.a 1> output_file 2> error_file
```

1.5 `.prrc` Configuration File

The `.prrc` file contains a list of options that determine the format of a source file when it prints. The configuration file must be located in the user's current working directory or the home directory. The file consists of a single `set` command, followed by a series of options.

`set` has the syntax:

```
set option [argument]
```

Each option specification must be on its own line, with any number of blanks between the different parts and any number of blank lines between specifications.

1.6 Command-line Options

Options on the command line take precedence over options in the `.prrc` file.

1.7 `a.pr` Output

The following sections discuss options for major formatting issues.

1.7.1 Comments

Only comments in a single program unit (package, subprogram or task) align with each other. If, for example, a package body contains a subprogram and a task, the comments in the subprogram align, the comments in the task align, and the comments outside of these two align.

Comments print either to the right or to the left, as described in the following paragraph. Use `set align_cmts` (or `-ac` and `-al`) to specify whether a comment, when aligned at the right, prints to the right of the longest line containing a comment or to the right of the longest line, regardless of whether or not it contains a comment. If printed at the right, comments align four spaces to the right of either the longest line or the longest line containing a comment, depending on the option specified.

To print at the left, comments must be preceded by a blank line or be no farther than one level of indentation to the right of the immediately previous statement. If the line immediately previous to this comment is a comment on its own line, this comment aligns with the previous one.

1.7.2 Pagination

Pagination occurs in a variety of ways. `pragma PAGE` is recognized, which starts the code following the pragma on a new page. This cannot be turned off. If no paging options are specified on the command line or in the `.prrc` file, pagination is performed using form feeds rather than blank lines.

If the `-p` *number* option or the `.prrc` command `set page` *number* is given, pagination is performed with blank lines. The number specified in the option gives the length of a page. For example, if

```
set page 65
```

is specified and a block of code ends on the 54th line, 11 blank lines print by default. Of course, the page size specified must be greater than or equal to the maximum number of lines on a page (set by `-l` *number* or `set lines` *number*).

If the command-line option `-pl` or the `.prrc` command `set page_lu` is used, each library unit (indicated by a `with` clause) begins on a new page.

If a block of code does not fit on the current page, a new one starts. If the block of code does not fit on a single page, that block splits between the necessary number of pages.

If the `-np` option or the `.prrc` option `set no_page` is used, pagination occurs only when a `pragma PAGE` is encountered.

1.7.3 Splitting Lines

A line is split after one of the following keywords or operators.

```
abs             +
and             -
and then        &
is              /=
mod             =
not             <
or              <=
or else         >
renames         >=
rem             :
xor             :=
**              |
*               ,
/
```

A line is not forced to conform to the maximum length if it contains none of the above. If, after splitting a line as much as possible, its length is greater than desired, a warning message issues.

When a line is split, the portion that prints on the next line indents one-half level from the original portion to show that it is a split line. For example, if each level indents 4 spaces and the original line is 2 levels deep (indents 8 spaces), the split portion indents 10 (8+2) spaces. If each level indents a single space, split portions align with the original portion.

Subprograms, `accept` and `entry` headers, or parameter lists are special cases. Parameters lists split as far as possible by aligning parameter declarations.

If warning messages print, the offending line input line number is part of the message. This number is not always the true line number. For example, if the input file contains the following,

```
procedure TEST(ARGUMENT1 : integer;
               ARGUMENT2 : boolean;
               ARGUMENT3 : integer);
```

all three lines are concatenated then processed. If the combined line length is greater than desired, the line splits, looking the same as above. If the length of the split lines is too long, a warning prints, but the line number given for `ARGUMENT2`... and `ARGUMENT3`... is the input line number of `procedure TEST`.

1.7.4 Line Length Specification

The maximum line length includes the comment, if one exists. If the comments align to the right of the longest line (regardless of whether or not it contains a comment), a line of code can be the specified length minus 14. For example, if the specified line length is 80, the maximum length a line of code is 66 (80 minus 4 spaces for indentation of comment and 10 spaces for the comment, regardless of how long the comment is).

The same situation occurs if comments are specified to align to the right of the longest line containing comments and the line in question has a comment. If a line does not contain a comment, its maximum line length is that specified.

1.7.5 Tabs

If `-t` *number* or the `.prrc` command `set tabs` *number* is used, tabs print when the indentation needed for a given line is greater than or equal to the specified number. For example, if `set tabs 4` and `set indent 4` are both specified, each new level indents with an additional tab. If `set indent 5` is given, each new level indents with a tab and a blank.

By default, a tab prints for every 8 characters needed in indentation (as if `set tabs 8` is specified). Specify the `-t 0` option or the `.prrc` command `set tabs 0` to indent with spaces rather than tabs.

1.8 Examples

The following examples show typical use of `a.pr` with options supplied in a `.prrc` file and from the command line.

1.8.1 Unformatted Source File

Figure 1-1 uses the following unformatted file.

```
--file: show_pr.a
Procedure show_pr Is
Type rec
    Is Record
i : integer;     -- “char” decl is the longest line
char : character:=’x’; end Record;

    r   :   rec   := (0, ’z’);             -- Longest line with
comment
        int:integer:=5;-- Declarations and comments
   -- should each be aligned
Begin
  -- Because these comments are indented no more than one
 -- level of indentation from the previous statement, they
-- should be aligned at the left.

r.int :=
i;
        end;
```

Figure 1-1 Example of an Unformatted Source File

1.8.2 `.prrc` File

A `.prrc` file is created with options for indentation, margins, the placement of record block indicators, and other formatting choices. See Figure 1-2.

```
set indent 4
set margin 4
set tabs   0
set record next
set reserved upper
set ident lower
set align_cmts line
```

Figure 1-2 Example of `.prcc` File

1.8.3 Output

Executing the command

```
a.pr show_pr.a
```

reformats the file according to the specifications of the `.prrc` file and places the output on standard output. See Figure 1-3.

```
--file: show_pr.a
PROCEDURE show_pr IS
    TYPE rec IS
      RECORD
        i    : integer;          -- "char" decl is the longest line
        char : character := 'x';
      END RECORD;

    r   : rec     := (0, 'z');   -- Longest line with comment
    int : integer := 5;          -- Declarations and comments
                                 -- should each be aligned
BEGIN
-- Because these comments are indented no more than one
-- level of indentation from the previous statement, they
-- should be aligned at the left.

    r.int := i;
END show_pr;
```

Figure 1-3 Example of a Formatted File

1.8.4 Command-line Options

Command line options override those given the `.prrc` file (if present). The command

```
a.pr -i 4 -m 0 -RS -rl -il -ac show_pr.a
```

in conjunction with the previous `.prrc` file produces the following reformatted listing in Figure 1-4:

```
--file: show_pr.a
procedure show_pr is
    type rec is record
       i    : integer;         -- "char" decl is the longest line
        char : character := 'x';
    end record;

    r   : rec     := (0, 'z');   -- Longest line with comment
    int : integer := 5;          -- Declarations and comments
                                 -- should each be aligned
begin
-- Because these comments are indented no more than one
-- level of indentation from the previous statement, they
-- should be aligned at the left.

    r.int := i;
end show_pr;
```

Figure 1-4 Example of a Reformatted Listing

1

"I made him a visit, hoping to find
That he took better care for improving his mind."

Isaac Watts

Ada Preprocessor 2

`a.app` is an Ada preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Input is Ada source intermixed with preprocessor control lines and macro substitutions. Output is Ada source; all control lines convert to comments, all source files are included, and all macro substitutions are replaced.

Preprocessor control lines begin with a sharp character (#) and are recognized only by `a.app`. Syntax of the Ada preprocessor is Ada-like and can span across control lines. `a.app` achieves conditional compilation by evaluating expressions. The primary components are identifiers defined either in an Ada library or locally in the source. `a.app` supports a simple form of macro substitution; the value of a defined identifier is replaced where that identifier, prefixed by a dollar sign ($), occurs in the Ada source.

2.1 Invocation

The syntax for invoking `a.app` directly is:

`a.app` [*options*] *in_file out_file*

When you specify the `-P` option on the `ada` command line, the compiler invokes `a.app`.

Include the following `INFO` directive in the `ada.lib` file, and cause the compiler to invoke `a.app`:

`APP:INFO:`*boolean_value*`:`

If *boolean_value* is set to TRUE, the compiler invokes a.app automatically before compiling the source; any other value for *boolean_value* has no effect. Default value of the APP INFO directive is FALSE. The -P option to the ada command takes precedence over the APP INFO directive.

When the compiler invokes a.app, a temporary output file is created and is discarded at the end of the compile. All diagnostics are in reference to the original input file.

If an error is encountered, *out_file* is not created.

If no files are specified on the command line, the standard input and standard output are used.

a.app supports these command line options:

-w	(warnings) Suppress warning diagnostics.
-s	(strip) Control and inactive lines are stripped from the output source.
-D *identifier type value*	(define) Define identifier of a specified type and value.
-L *library_name*	(Library) Operate in SC Ada library *library_name* (the current working directory is the default).

Except for -s (strip), which is available only when invoking a.app directly, ada recognizes these options.

2.2 Chapter Conventions

This chapter describes the syntax and semantics of the Ada PreProcessor (APP) language. The APP language is based on a subset of the Ada language.

Each section introduces its subject, gives any necessary syntax rules, and describes the semantics of the corresponding language constructs. *Examples*, *Notes*, *Differences*, and *References* can appear at the end of a section.

Examples illustrate the possible forms of the constructs described. *Notes* emphasize consequences of the rules described in the section or elsewhere. *Differences* highlight differences between the preprocessor language and the Ada language. *References* attract the attention of readers to a term or phrase having a technical meaning defined in another section in the *Ada Reference Manual* or in the *SPARCompiler Ada Reference Guide*

The context-free syntax of the APP language is described using a simple variant of Backus-Naur-Form, as described in section 1.5 of the *Ada Reference Manual*. The sharp character, denoting a control line, does not appear in any of the syntax rules; it is implicit that all constructs must appear on one or more control lines. Rules dealing with how Ada text is intermingled with the constructs and the effects on them are explained in the appropriate sections.

2.2.1 Lexical Elements

APP uses the ISO graphic character set (see section 2.1 in the *Ada Reference Manual*).

The basic lexical elements of the preprocessor consist of delimiters, identifiers, numeric literals, string literals, and comments. Rules of composition are those defined by Ada (see section 2 in the *Ada Reference Manual*).

These lexical elements are recognized on lines that begin with the sharp character (control lines); the sharp has no other effect. A control line contains a sequence of lexical elements (possibly none).

Text on non-control lines consists of a sequence of separate Ada lexical items. An exception to this is a macro substitution, which is composed of the dollar character followed immediately by an identifier.

Intermingling control and non-control lines has no effect on the syntax of either language.

When the possibility exists of interpreting adjacent lexical characters as a single lexical element, you *must* separate the adjacent lexical elements with an explicit separator. An explicit separator is any space character (except in a comment or string literal), format effector or end of line character. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except in a comment.

Use one or more separators between any two adjacent lexical elements. You *must* use at least one separator between an identifier or a numeric literal and an adjacent identifier or numeric literal.

A delimiter is any of the following special characters

& ' ( ) * + , - / : ; < = > |

or one of the following compound delimiters, each composed of two adjacent special characters

=> .. ** := /= >= <=

The following reserved words have significance in the preprocessor language:

```
abs         else        mod         rem
and         elsif
            end         not         then
case
constant                or          when
            if          others
            in
declare     is          pragma      xor
```

The remaining Ada reserved words are reserved but without any significance (see section 2.9 of the *Ada Reference Manual*).

The replacement of the vertical bar, sharp, and quotes are supported, as specified in section 2.10 of the *Ada Reference Manual.*

Figure 2-1 gives examples of lexical elements.

Examples of identifiers

```
#   COUNT         X     get_symbol    Ethelyn              Marion
#   SNOBOL_4      X1    PageCount     STORE_NEXT_ITEM
```

Examples of numeric Literals:

```
#   12              0       1E6                    123_456          -- integer literals
#   12.0            0.0     0.456                  3.14159_26       -- real literals
#   1.34E-12                1.0E+6                                  -- real literals with exponent
#   2#1111_1111#            16#FF#                 016#OFF#         -- integer literals of value 255
#   16#E#E1                 8#340#                                  -- integer literals of value 224
#   16#F.FF#E+2             2#1.1111_1111_111#E11                   -- real literals of value 4095.0
```

Examples of string Literals

```
#  "Message of the day:"
#  ""                          -- an empty string literal
#  " "     "A"       """"      -- three string literals of length 1
#  "Characters such as $, % and } are allowed in string literals"
```

Examples of comments

```
# end if;                     -- processing of LINE is complete
#-- a long comment can be split onto
#-- two or more control lines

#-------------- the first two hyphens start the comment
```

Figure 2-1 Example of Lexical Elements

Note – The following syntactic categories, defined in the *Ada Reference Manual*, are used throughout this chapter in APP syntactic rules.

The following Ada delimiters are not used in APP.

```
. <<  >>  <>
```

The set of APP reserved words is a proper subset of the reserved words defined in Ada.

Differences

A character literal is not supported currently.

The sharp and dollar character, which by themselves are illegal lexical items in Ada, have significance only in APP.

References

character literal, section 2.5; comment, section 2.7; delimiter, section 2.2; format effector, section 2.1; graphic character, section 2.1; identifier, section 2.3; macro substitution, section 1.3.12; numeric literal, section 2.4; separator, section 2.2; and string literal, section 2.6 in *Ada Reference Manual*

2.2.2 Program Structure

The text of a conditional-compilation program consists of Ada text, interspersed with optional preprocessor control lines. Ada text has no effect on the preprocessing, except for macro substitutions. The legality of the Ada text is checked only to verify that each lexical item is formed correctly.

A conditional compilation consists of a sequence of preprocessor statements. Each preprocessor statement must be specified on one or more control-lines.

```
conditional_compilation ::= statement_sequence

statement_sequence ::= {statement}
```

A statement defines the action to be taken by the preprocessor. The basic statements are:

```
statement ::=
       object_declaration
     |  assignment_statement
     |  if_statement
     |  case_statement
     |  declare_statement
     |  pragma
```

Each control-line, when preprocessed, converts to a comment, that is, the sharp character is preceded by the comment character sequence. If the strip option is in effect, the line (including the end of line character) is discarded.

Note – Only use the strip option to generate a new file that is independent of the original file, because the line numbering no longer corresponds to the original.

References

Section 2.2.1, “Lexical Elements,” on page 2-3
Section 2.2.3.3, “Local Declaration,” on page 2-10
Section 2.2.8.1, “If Statement,” on page 2-24
Section 2.2.7, “Assignment,” on page 2-23
Section 2.2.8.2, “Case Statement,” on page 2-26
Section 2.2.9, “Declare Statement,” on page 2-28
Section 2.2.11, “Pragmas,” on page 2-31
Section 2.2.12, “Macro Substitution,” on page 2-33

2.2.3 Declarations

The preprocessor language defines only one kind of declared entity, an object. An object is an entity that contains a value of a given type. Declare an object in several ways:

- Ada Library Directive
- Command Line
- Local Declaration

Each form of declaration introduces an identifier as the declared entity. After the declaration the identifier is said to be defined.

For each form of declaration, the language rules define a certain region of text called the scope of the declaration. In its scope and only there, are places where the identifier can refer to the associated declared entity. At such places, the identifier is said to be a name of the entity (its simple name); the name is said to denote the associated entity.

Note – Use the attribute `P'DEFINED` to query whether the identifier is defined.

References

“P’DEFINED” on page 2-16

2.2.3.1 Ada Library Directive

Define an object in an Ada library via a `DEFINE` directive. The form is:

```
identifier:DEFINE:type:value:
```

A type is defined as:

```
type ::= BOOLEAN | INTEGER | REAL | STRING | TEXT
```

A value is one of the following:

```
value ::=
      boolean_value

    | numeric_value

    | rational_value

    | string_literal

    | text_literal

boolean_value   ::= FALSE | TRUE
numeric_value   ::= [+|-]numeric_literal
rational_value  ::= numeric_value/numeric_value
text_literal    ::= {graphic_character}
```

An optional sign prefixes `numeric value`. A `rational value` is a pair of `numeric values`, separated by a divide character. Each must be a real literal.

A string literal is as defined in section 2.6 of the *Ada Reference Manual*.

A text literal is a sequence of graphic characters (possibly `none`).

An `INFO` directive is recognized and treated as if it is declared as a `DEFINE` directive of the `type TEXT`.

An `INFO` or `DEFINE` directive definition is equivalent to a constant object declaration.

The SC Ada tool `a.info` provides operations to add, modify, and delete directives. Figure 2-2 is an example of library directives.

```
DEBUG:DEFINE:BOOLEAN:TRUE:
LEVEL:DEFINE:INTEGER:13:
HERTZ:DEFINE:REAL:100.0:
MESSAGE:DEFINE:STRING:“Hello World!”:
COMPILER:DEFINE:TEXT:VADS:
VERSION:INFO:2.1:
```

Figure 2-2 Example of Ada Library Directives

Note – An `INFO` directive is reserved for use by SC Ada components and are predefined. An `INFO` directive is similar in definition to the following `DEFINE` directive:

```
identifier:DEFINE:TEXT:value:
```

A `DEFINE` directive cannot be substituted for an `INFO` directive.

Only use text literal in a directive; a string literal must be used in the source.

Differences

The definition of a text literal and rational value have no counterparts in Ada.

References

“Local Declaration” on page 2-10
directive, *SPARCompiler Ada User’s Guide*
`a.info` and `APP INFO` directive, *SPARCompiler Ada Reference Guide*
graphic character, section 2.1; numeric literal, section 2.4; real literal, section 2.4; string literal, section 2.6. *Ada Reference Manual*

2.2.3.2 Command Line

Define an object on the command line with the `-D` option. The form is:

```
-D identifier type value
```

A command line definition is equivalent to a constant object declaration. See Figure 2-3.

```
-D DEBUG BOOLEAN TRUE
-D LEVEL INTEGER 13
-D HERTZ REAL 100.0
-D MESSAGE STRING '"Hello World!"'
-D COMPILER TEXT VADS
```

Figure 2-3 Example of Command Line

Note – Specifying a string value on the command line requires that the string be enclosed by apostrophes. The shell restricts embedded apostrophes.

2.2.3.3 Local Declaration

An object is defined locally and has the following syntax:

```
object_declaration ::=
    identifier : [constant] type [:= expression];
```

An object declaration declares an object whose type is specified by the identifier type. If the object declaration includes the assignment compound delimiter, `:=`, followed by an expression, the expression specifies an initial value for the object; the type of the expression must be that of the object.

The declared object is a constant if the reserved word `constant` appears in the object declaration; the declaration must then include an explicit initialization. The value of a constant cannot be modified after initialization.

An object that is not a constant is a variable. The value of a variable must be changed by an assignment. Figure 2-4 gives examples of local declarations.

Examples of variable declarations

```
#  COUNT        : INTEGER;
#  SORTED       : BOOLEAN          := FALSE;
#  MESSAGE      : STRING           := “Hello World!”;
```

Examples of constant declarations

```
#  LIMIT        : constant INTEGER  := 10;
#  VERSION      : constant REAL     := 2.1;
```

Figure 2-4 Example of Local Declarations

Note – A local declaration is the only means to define a variable, since definitions introduced by a directive or command line are constants.

Differences

An identifier list is not supported.

No constraints are checked during assignment.

2.2.4 Types

A type is characterized by a set of values and operations. All types are predefined and are not extensible.

Supported types are boolean, numeric, and string.

All types share the following basic operations which consist of assignment, relational operators, and explicit type conversions.

Note – A character type is not supported yet.

2.2.4.1 BOOLEAN

The type BOOLEAN contains the two values FALSE and TRUE, ordered with the relation FALSE < TRUE.

Additional operations consist of logical operators, short-circuit control forms, logical negation, and image attribute.

2.2.4.2 INTEGER

The type INTEGER is an integer type whose set of values range from some lower bound to some upper bound, where the lower bound is some value < 0 and the upper bound is some value > 0. Integer literals are of the type INTEGER.

Additional operations consist of membership tests, binary adding operators, unary adding operators, multiplying operators, absolute value, exponentiation, and image attribute.

Note – The type INTEGER corresponds to the Ada predefined type UNIVERSAL_INTEGER.

The lower and upper bounds of the type INTEGER depend on the amount of available memory required for the internal representation.

References

UNIVERSAL_INTEGER, section 3.5.4 in *Ada Reference Manual*.

2.2.4.3 REAL

The type REAL is a real type comprised of rational values. Real literals are of the type REAL.

Additional operations consist of binary adding operators, unary adding operators, multiplying operators, absolute value, exponentiation, and image attribute.

Notes – The `type REAL` corresponds to the Ada predefined `type UNIVERSAL_REAL`.

With rational values, one can assume that (1.0 / X)*X = 1.0.

References

Section 3.5.6 in *Ada Reference Manual*

2.2.4.4 STRING

The `type STRING` is a variable length array of a character component in the range 1 .. N, where N is some nonnegative integer value.

String literals are basic operations applicable to the `type STRING` and `TEXT`.

Additional operations consist of concatenation, slice operation, length attribute, and value attribute.

Notes – A character type is no supported yet.

When `N` is zero, this denotes the null string `""`.

Access a single character component by a slice operation.

Differences

A `STRING` object, in Ada, must always be constrained in an object declaration.

2.2.4.5 TEXT

The `type TEXT` is a derived type of `STRING`. The main difference is that the external value is a sequence of graphic characters.

Note – An external `TEXT` value is specified in an `INFO` or `DEFINE` directive or is used in a macro substitution.

2.2.5 Names

Names denote objects, slices of objects or attributes of objects.

```
name::=
    simple_name
  | slice
  | attribute

simple_name::=identifier
```

A simple name for an entity is the identifier associated with the entity by its declaration. The evaluation of a simple name consists of evaluating its definition. The simple name must be defined and have a value.

Note – The use of an undefined identifier is legal in certain contexts that prevent the evaluation. These include the `if` statement, short-circuit forms, and any context that is inactive.

Differences

In Ada, the use of an undefined identifier is an error.

An object must contain a value, whereas in Ada, the evaluation yields an erroneous program.

References

Note on page 2-19
“Conditional Processing” on page 2-23
“If Statement” on page 2-24

2.2.5.1 Slice

A slice denotes a sequence of consecutive characters of a STRING or TEXT object.

```
slice ::= simple_name(range)

range ::= simple_expression .. simple_expression
```

A range specifies a subset of values of the type INTEGER. The range L .. R specifies the values from L to R inclusive if the relation L <= R is TRUE. A value V belongs to the range if the relations L <= V and V <= R are both TRUE. A null range is range for which the relation R < L is TRUE.

Except for a null range, the bounds of the range must belong to the range of the sliced object.

An implicit conversion adjusts the bounds from L .. R to 1 .. R-L+1. Figure 2-5 shows an example of slices.

```
#MESSAGE(6..10)    -- a slice of 5 characters
#MESSAGE(1..0)     -- a null slice
```

Figure 2-5 Example of Slices

Notes – The implicit conversion occurs during assignment and when part of any expression.

The syntax does not allow slicing of a slice, as in the example FOO(1..10)(3..6).

2.2.5.2 Attributes

An attribute denotes a basic operation of an entity given by a prefix.

```
attribute ::= prefix'attribute_designator
prefix ::= identifier | simple_name
attribute_designator ::= simple_name [(expression)]
```

The applicable attribute designators depend on the prefix.

The following attributes are defined:

P'DEFINED
For a prefix P that is an identifier:
Yields the value TRUE if the identifier P is defined; yields the value FALSE otherwise. The value of this attribute is of the type BOOLEAN.

P'LENGTH
For a prefix P that denotes a STRING or TEXT object:
Yields the number of character components in the object P. The value of this attribute is of the type INTEGER.

P'IMAGE
For a prefix P that is the type INTEGER, REAL, or BOOLEAN:
This attribute is an operation with a single argument. The actual argument X must be a value of the type P. The result type is the type STRING or TEXT. The result is the image of the value in display form.

The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent or trailing spaces; but with a one character prefix that is either a minus sign or a space.

For a real value, the image is the corresponding real decimal literal if the fractional part of the value is zero. Otherwise, the image is a pair of real decimal literals separated by the divide character, with no underlines, leading zeros, exponents or trailing spaces but with a single-character prefix (either a minus sign or a space).

The image of a boolean value is the corresponding identifier in upper case.

P'VALUE
For a prefix P that is the type INTEGER, REAL, or BOOLEAN:
This attribute is an operation with a single argument. The actual argument X must be a value of the type STRING or TEXT. the result type is the type P. Any leading and any trailing spaces that correspond to X are ignored.

For the type INTEGER, if the sequence of characters has the syntax of a numeric value, the result is this value.

For the type REAL, if the sequence of characters has the syntax of a rational value, the result is this value.

For the type BOOLEAN, if the sequence of characters correspond to the identifiers TRUE or FALSE, the result is this value.

For any other case, it is an error. See Figure 2-6.

```
#   COUNT'DEFINED              -- TRUE if COUNT is defined
#   MESSAGE'LENGTH             -- yields the value 12
#   REAL'IMAGE(VERSION)        -- yields the value "2.1"
#   BOOLEAN'VALUE(MESSAGE)     -- erroneous
```

Figure 2-6 Example of Attributes

Differences

The attribute `P'DEFINED` has no counterpart in Ada.

The attributes `P'IMAGE` and `P'VALUE`, where `P` is `REAL`, have no counterparts in Ada.

The attributes `P'FIRST`, `P'LAST`, and `P'RANGE` are not supported for the string types.

References

Section 2.2.3, "Declarations," on page 2-7

2.2.6 Expressions

An expression is a formula that defines the computation of a value.

```
expression ::=
      relation {and relation}
   |  relation {or relation}
   |  relation {xor relation}
   |  relation {and then relation}
   |  relation {or else relation}

relation ::=
      simple_expression [relational_operator
simple_expression]
   |  simple_expression [not] in range

simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator
term}

term ::=
  factor {multiplying_operator factor}

factor ::=
      primary [** primary]
   |  abs primary
   |  not primary

primary ::=
      numeric_literal
   |  string_literal
   |  name
   |  type_conversion
   |  (expression)
```

Each primary has a value and a type. See Figure 2-7.

Examples of primaries

```
#   100                          -- integer literal
#   4.0                          -- real literal
#   "Hello World!"               -- string literal
#   LIMIT                        -- constant
#   COUNT                        -- variable
#   MESSAGE'LENGTH               -- attribute
#   MESSAGE(1..5)                -- slice
#   REAL(1)                      -- conversion
#   (LIMIT + 1)                  -- parenthesized expression
```

Examples of expressions

```
#   VERSION                      -- primary
#   not SORTED                   -- factor
#   2*COUNT                      -- term
#   -4.0                         -- simple expression
#   -4.0 + VERSION               -- simple expression
#   B**2 - 4.0*A*C               -- simple expression
#   PASSWORD(1..3) = "BWV"       -- relation
#   LIMIT in 1..10               -- relation
#   LIMIT not in 1..10           -- relation
#   COUNT = 0 or ITEM_HIT        -- expression
#   (COLD and SUNNY) or WARM     -- expression (parentheses are required)
#   A**(B**C)                    -- expression (parentheses are required)
```

Figure 2-7 Example of Expressions

Note – The short-circuit forms prevent the evaluation of the right-hand expression. This is useful when referencing an undefined object. For example:

```
#  DEBUGGING: BOOLEAN := DEBUG'DEFINED and then DEBUG;
```

Differences

In APP, erroneous expressions are detected immediately, whereas in Ada an exception is raised, for example, divide by zero.

2.2.6.1 Operators and Expression Evaluation

The preprocessor language defines the following classes of predefined operators. The basic properties of these operators are identical to their counterparts in the Ada language (see section 4.5 in the *Ada Reference Manual*). The evaluation of an expression delivers a value or causes an error to be detected.

```
logical_operator::= and | or | xor
relational_operator::= = | /= | < | <= | > | >=
binary_adding_operator::= + | - | &
unary_adding_operator::= + | -
multiplying_operator::= * | / | mod | rem
highest_precedence_operator::= ** | abs | not
```

The operations on numeric types either yield the mathematically correct or an erroneous result. See Table 2-1.

Table 2-1 Operators and Expression Evaluation

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
and	conjunction	BOOLEAN	BOOLEAN	BOOLEAN
or	inclusive disjunction	BOOLEAN	BOOLEAN	BOOLEAN
xor	exclusive disjunction	BOOLEAN	BOOLEAN	BOOLEAN
=	equality	any type	same type	BOOLEAN
/=	inequality	any type	same type	BOOLEAN
< <=> >=	test for ordering	any type	same type	BOOLEAN
+	identity		INTEGER REAL	INTEGER REAL
-	negation		INTEGER REAL	INTEGER REAL
+	addition	INTEGER REAL	INTEGER REAL	INTEGER REAL

Table 2-1 Operators and Expression Evaluation *(Continued)*

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
-	subtraction	INTEGER REAL	INTEGER REAL	INTEGER REAL
&	concatenation	STRING TEXT	STRING TEXT	STRING TEXT
*	multiplication	INTEGER REAL REAL INTEGER	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
/	division	INTEGER REAL REAL	INTEGER REAL INTEGER	INTEGER REAL REAL
mod	modulus	INTEGER	INTEGER	INTEGER
rem	remainder	INTEGER	INTEGER	INTEGER
abs	absolute value	INTEGER REAL	INTEGER REAL	INTEGER REAL
not	logical negation		BOOLEAN	BOOLEAN
**	exponentiation	INTEGER REAL	INTEGER INTEGER	INTEGER REAL

The concatenation of two string objects X and Y yields a new string expression with bounds 1.. X'LENGTH + Y'LENGTH.

For integer division, rem, and mod, the right operand must be nonzero.

Exponentiation of an integer requires that the exponent be non-negative.

2.2.6.2 Type Conversions

The evaluation of a type conversion evaluates the expression given as the operand and converts the resulting value to a specified target type.

```
type_conversion ::= type(expression)
```

A conversion of an operand of a given type to the type itself is allowed. No special restrictions limit the form of the expression.

The allowed type conversions correspond to the following cases:

- Numeric types
  The operand is either `type INTEGER` or `REAL`; the value of the operand converts to the target type, which is either of the `type INTEGER` or `REAL`. The conversion of a real value to an integer value rounds to the nearest integer; if the operand is halfway between two integers, the rounding mode is round-to-nearest-even.

- String types
  The operand is either `type STRING` or `TEXT`; the value of the operand converts to the target type, which is either `type STRING` or `TEXT`. The representation does not change.

See Figure 2-8.

Examples of numeric type conversions

```
#   REAL(1)                 -- value is 1.0

#   INTEGER(1.6)            -- value is 2
#   INTEGER(1.5)            -- value is 2 (round-to-nearest-even)
#   INTEGER(2.5)            -- value is 2 (round-to-nearest-even)
#   INTEGER(-1.5)           -- value is -2 (round-to-nearest-even)
#   INTEGER(-2.5)           -- value is -2 (round-to-nearest-even)
```

Examples of conversions between string types

```
#   STRING(COMPILER)        -- bounds are those of COMPILER
#   TEXT(MESSAGE(6..10))    -- bounds are 1 and 5
#   TEXT("")                -- bounds are 1 and 0
```

Figure 2-8 Example of Type Conversions

Differences

A string literal is allowed as an operand.

In Ada, the rounding mode for the conversion of a real to an integer is implementation-dependent.

2.2.7 Assignment

An assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand expression must be of the same type.

```
assignment_statement ::=
    variable_simple_name := expression;
```

See Figure 2-9.

```
#   SORTED:= TRUE;
#   CONTROL_LINE := "# " & SOURCE_LINE;
#   CELSIUS := (FAHRENHEIT-32.0) * (5.0/9.0);
```

Figure 2-9 Example of Assignment Statements

2.2.8 Conditional Processing

Conditional processing activates or inactivates a section of source text. The source text is a sequence of Ada and preprocessing source lines. Each line is either active or inactive. An active line is preprocessed and any constituents are evaluated; an inactive line is not evaluated. An inactive Ada source line converts to a comment; the character sequence "`--*`" precedes the line.

Note – For a section of text that is inactive, the APP syntax is still checked.

2.2.8.1 `If` *Statement*

An `if` statement selects conditionally the enclosed statement sequence, depending on the (truth) value of one or more corresponding conditions.

```
if_statement ::=
       if condition then
          statement_sequence
     { elsif condition then
          statement_sequence}
     [ else
         statement_sequence]
       end if;

condition ::= boolean_expression
```

An expression specifying a condition must be of `type BOOLEAN`.

For the evaluation of an `if` statement, the condition specified after `if` and any conditions specified after `elsif`, are evaluated in succession (treating a final `else` as `elsif then`), until one evaluates to `TRUE` or all conditions are evaluated and yield `FALSE`. If one condition evaluates to `TRUE`, then the corresponding statement sequence and all Ada source between `then–elsif`, `then–else` or `else–end`, is active; otherwise the remaining statement sequences and Ada text are inactive. See Figure 2-10.

```
#  if DEBUG'DEFINED and then DEBUG then
       TEXT_IO.PUT("Debugging ....");
#  end if;

   X :=
#  if BIAS >= 0 then
        X+1
#  else
        X-1
#  end if;
   ;

#  if COND1 then
       -- cond1 part
#  elsif COND2 then
       -- cond2 part
#  else
       -- else part
#  end if;
```

Figure 2-10 Example of Conditional Processing — `if` Statement

---

Note – An `if` statement achieves conditional evaluation. Evaluation of an object checks that the object is defined and has a value. In the above example, if `COND1` is true, `COND2` is not evaluated; if `COND2` is undefined, an evaluation error is prevented.

Differences

In APP, use an undefined identifier in expression, as long as it is not evaluated.

2.2.8.2 `Case` *Statement*

A `case` statement selects conditionally one of a number of alternative statement sequences; the chosen alternative is defined by the value of an expression.

```
case_statement ::=
    case expression is
     case_statement_alternative
    {case_statement_alternative}
    end case;

case_statement_alternative ::=
    when choice {| choice} =>
       statement_sequence

choice ::=
        simple_expression
    |   range
    |   others
```

The expression is any one of the available types. Each choice in a `case` statement alternative must be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen (possibly none).

A value of the type of the expression must be represented once, and only once, in the set of choices. A choice defined by a range stands for all values in the corresponding range (none if a null range). The choice `others` is allowed only for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives.

The preprocessing of a `case` statement consists of the evaluation of the expression followed by the evaluation of each choice and the preprocessing of the chosen (possibly none) statement sequence.

For a chosen alternative, the corresponding statement sequence and all Ada source between the arrow and to the next alternative or end, is active; otherwise, the remaining alternatives and Ada text are inactive. See Figure 2-11.

```
#  case DEBUGGING is
#   when TRUE =>
      TEXT_IO.PUT("Debugging ...");
#  end case;

#  case TARGET is
#   when “rt” =>
      rt_specific;
#   when “sparc” =>
      sparc_specific;
#   when “mc68020” =>
      mc68020_specific;
#   when others =>
#     pragma ERROR(“unknown TARGET: “ & TARGET);
#  end case;
```

Figure 2-11 Example of Conditional Processing — `Case` Statement

---

Note – An `others` choice is not required; it is possible that no alternative is chosen.

Differences

In Ada, the type expression must be a discrete type and all choices specified must represent all possible values of the expression.

2.2.9 `Declare` *Statement*

A `declare` statement encloses a statement sequence, which in effect is a declarative region.

```
declare_statement ::=
    declare
        statement_sequence
    end declare;
```

Figure 2-12 is an example of a `declare` statement.

```
#  declare
#      KIND: constant STRING := STRING(COMPILER);
       TEXT_IO.PUT("Compiler: " & $KIND);
#  end declare;
```

Figure 2-12 Example of a `Declare` Statement

References

Section 2.2.10.1, “Declarative Region,” on page 2-28

2.2.10 *Visibility Rules*

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the preprocessor text are described.

2.2.10.1 *Declarative Region*

A declarative region is a portion of the program text that encompasses external declarations defined outside of the program text. A single declarative region is formed by the text of each of the following:

- A statement sequence of a conditional compilation program
- A statement sequence of a `declare` statement

The following two regions are implicitly part of the program text and extend to the end of the program and are said to enclose a region:

- An Ada library
- A command line

The Ada libraries visible to the program are those that are defined on the Ada path in the local Ada library, excluding any closure. An Ada library encloses the predecessor Ada library on the Ada path. The local Ada library encloses the command line, which encloses the program text.

References

Ada path, *SPARCompiler Ada Reference Guide*

2.2.10.2 Scope of Declarations

The scope of a declaration that occurs immediately in a declarative region extends from the beginning of the declaration to the end of the declarative region. This implies that a library directive or command line declaration extends from its declaration to the end of the program text.

2.2.10.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text, Ada library or command line, is defined by the visibility rules.

A declaration is visible only in a certain part of its scope; this part starts at the end of its declaration and extends to the end of the immediate scope of the declaration but excludes places where the declaration is hidden.

A declaration is said to be hidden in an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden in the immediate scope of the inner homograph. Each of the two declarations is said to be a homograph of the other if both declarations have the same identifier.

Two declarations that occur immediately in the same declarative region must not be homographs, unless the declarative region is an Ada library or command line, in which case the second declaration is ignored.

No homograph is allowed for any of the predefined types and boolean values. See Figure 2-13.

Examples of Ada library directives

```
MESSAGE:DEFINE:STRING:“Hello World!”:
MESSAGE:DEFINE:TEXT:Hello World!: -- ignored; redeclaration
```

Examples of command line

```
-D DEBUG BOOLEAN TRUE
-D DEBUG BOOLEAN FALSE              -- ignored; redeclaration

/DEFINE=("DEBUG:BOOLEAN=TRUE")
/DEFINE=("DEBUG:BOOLEAN=FALSE")     -- ignored; redeclaration
```

Examples of local declarations

```
#  if not NAME'DEFINED then
#      NAME: constant TEXT := "undefined";
#  end if;

#  LEVEL: INTEGER := 1;
#  if TRUE then
#      LEVEL: INTEGER := 2          -- illegal; redeclaration
#  end if;
#  declare
#     LEVEL: INTEGER := 3;          -- an inner homograph of LEVEL
#  end declare;

#  I: INTEGER := I + 1;             -- illegal; the declaration of I
                                    -- hides the use of I in the expression

#  FOO: BOOLEAN := FOO'DEFINED;     -- illegal; the declaration of FOO
                                    -- is not yet complete

#  BOOLEAN: INTEGER := 1;           -- illegal; a predefined type cannot
                                    -- be redefined
```

Figure 2-13 Example of Visibility

Differences

In Ada, predefined identifiers can be redefined.

2.2.11 Pragmas

A pragma conveys information to the preprocessor. The syntax is

```
pragma ::= pragma identifier [(expression)];
```

The identifier must be recognized by the preprocessor.

Differences

An unrecognized or illegal pragma is considered an error and not ignored as is the case in Ada.

2.2.11.1 Pragma INCLUDE

`pragma INCLUDE` includes files in a compilation. The form of this pragma is

`pragma INCLUDE` (*string_expression*);

This includes the entire contents of the named file at the point of the pragma and preprocesses it. The filename expression must be of the `type STRING` or `TEXT` and must be a well-formed filename specification.

The named file is first searched for in the local Ada library and in the sequence of Ada libraries contained on the Ada path. An `include` file that is obtained from a non-local Ada library uses the Ada path of the non-local Ada library to search for an `include` file.

See Figure 2-14.

```
#  HOME: constant string :="/usr/home/";

#  pragma INCLUDE ("file.a");-- is searched for in the
#   -- Ada libraries

#  pragma INCLUDE (HOME & "file.a");-- is searched for in the
#   -- directory HOME
```

Figure 2-14 Example of `pragma INCLUDE`

Note – The text of an `include` file is processed as if it is part of the original source, except that diagnostics refer back to the `include` file. It can contain preprocessor control lines, possibly even additional `INCLUDE` pragmas.

Obtain an `include` file from an Ada library that is not directly visible from the current Ada library.

Restriction

The `include` facility is not available when `a.app` is used indirectly during an Ada compile; it is restricted solely to the direct use of `a.app`.

2.2.11.2 Pragma WARNING

`pragma WARNING` issues a warning. The form of this pragma is

```
pragma WARNING [(string_expression)];
```

The pragma takes an optional `STRING` or `TEXT` argument, which if provided, is used as the warning message. See Figure 2-15.

```
#  pragma WARNING;
#  pragma WARNING ("this text is the warning message");
```

Figure 2-15 Example of `pragma WARNING`

Note – If the `-w` option is specified on the command line, `pragma WARNING` has no effect.

2.2.11.3 Pragma ERROR

pragma ERROR issues an error. The form of this pragma is:

pragma ERROR [(*string_expression*)];

The pragma takes an optional STRING or TEXT argument, which if provided, is used as the error message.

The effect of this pragma is to cause the preprocessing to fail.

See Figure 2-16.

```
#  pragma ERROR;
#  pragma ERROR ("this text is the error message");
```

Figure 2-16 Example of pragma ERROR

2.2.12 Macro Substitution

A macro substitution is identified by the following lexical item in the Ada text.

$identifier

The identifier must be defined and have a value.

Table 2-2 specifies the output format for each type.

Table 2-2 Macro Substitution

Type	Syntax
BOOLEAN	boolean_value
INTEGER	numeric_value
REAL	rational_value
STRING	string_literal
TEXT	text_literal

Decimal notation is used for integer and real literals. See Figure 2-17.

Input	Output
`# DEBUG: BOOLEAN := TRUE;` `if $DEBUG then`	`--# DEBUG: BOOLEAN := TRUE;` `if TRUE then`
`# BIAS: INTEGER := -16#ff#;` `ADJUST := ADJUST + ($BIAS);`	`--# BIAS: INTEGER := -16#ff#;` `ADJUST := ADJUST + (-255);`

`# PI: CONSTANT REAL := 3.14;`
`RADIANS := 180.0/($PI);`
	`--# PI: CONSTANT REAL :=`
	`3.14;`
	`RADIANS := 180.0/(`
	`157.0/50.0);`

`# S: STRING := "abc""def";` `if PATTERN = $s then`	`--# S: STRING := "abc""def";` `if PATTERN = "abc" "def" then`

`# NOP: TEXT :=`
`# "code_0'(op => nop)";`
`$NOP;`
	`--# NOP: TEXT :=`
	`--# "code_0'(op => nop)";`
	`code_0'(op => nop);`

Figure 2-17 Example of Macro Substitutions

Note – Macro substitution occurs only in the Ada text, not in APP control lines. The macro substitution cannot exceed the current line length limit. A macro substitution cannot extend to the next line. A macro substitution cannot be embedded in a comment or string literal, since these are lexical items. A numeric macro substitution must be parenthesized if part of a larger expression.

2.3 Example

The following example illustrates some of the possible declarations and the visibility rules involved.

For the following example, the two `ada.lib` files are defined as follows:

```
       file: /usr2/ada_2.1/self/standard/ada.lib

1)     !ada library
       ADAPATH=
       TARGET:INFO:SELF_TARGET:
       VERSION:INFO:2.1:
       VADS:INFO:/usr2/ada_2.1:
       HOST:INFO:some_host:

       file: /usr2/ada_2.1/test/ada.lib

2)     !ada library
       ADAPATH= /usr2/ada_2.1/self/standard
3)     HOST:DEFINE:STRING:"some_host":
       DEBUG:DEFINE:BOOLEAN:FALSE:
4)     DEBUG:DEFINE:BOOLEAN:TRUE:
```

With the following invocation in directory `/usr2/ada_2.1/test`:

```
5)     a.app -D DEBUG BOOLEAN TRUE example_in.a example_out.a
       file: /usr2/ada_2.1/test/example_in.a

6)     #
7)     # if not DEBUG'DEFINED then
       #  DEBUG: constant BOOLEAN := FALSE;
       # end if;
       #
8)     # HOST: constant STRING := "some_other_host";
       #
       # if DEBUG then
       with TEXT_IO;
       with DEBUG;
       # end if;
       procedure EXAMPLE is
       begin
       # if DEBUG then
       #  declare
9)     #     HOST: constant STRING := "yet_another_host";
          TEXT_IO.PUT("Debugging host: " & $HOST);
          DEBUG;
       #  end declare;
       # else
          null;
       # end if;
       end EXAMPLE;
```

1. At this point, the only defined objects are the predefined types and boolean values.
2. After processing the outermost Ada library, the objects TARGET, VERSION, VADS, and HOST are defined.
3. The declaration of HOST hides the definition in /usr2/ada_2.1/self/standard/ada.lib.
4. The redeclaration of DEBUG is ignored and a warning issues.
5. The command line declaration of DEBUG hides the definition in /usr2/vads6_10/test/ada.lib.
6. At this point, the user-defined non-local identifiers that are visible are:

```
TARGET:  CONSTANT TEXT := "SELF_TARGET";
VERSION: CONSTANT TEXT := "2.1";
VADS:    CONSTANT TEXT := "/usr2/ada_2.1";
HOST:    CONSTANT STRING := "some_host":
DEBUG:   CONSTANT BOOLEAN := TRUE;
```

7. Checks whether DEBUG is defined; if not, a local declaration is defined with a default value. This avoids any potential evaluation error in the following code. An alternative is to use the expression (DEBUG'DEFINED and then DEBUG) instead of (DEBUG).
8. Introduces the local constant HOST which hides the outer definition defined at 3). Its definition extends to the end of the source.
9. Introduces an inner definition of HOST which hides the outer definition defined at 8). Its definition extends to the end of the end declare;.

The resulting file is:

```
file: /usr2/vads6_10/test/example_out.a

--#
--# if not DEBUG'DEFINED then
--#DEBUG: constant BOOLEAN := FALSE;
--# end if;
--#
--# HOST: constant STRING := "some_other_host";
--#
--# if DEBUG then
with TEXT_IO;
with DEBUG;
--# end if;
procedure EXAMPLE is
begin
--# if DEBUG then
--#declare
--#   HOST: constant STRING := "yet_another_host";
   TEXT_IO.PUT("Debugging host: " & "yet_another_host");
   DEBUG;
--#end declare;
--# else
--*null;
--# end if;
end EXAMPLE;
```

If the strip option is used instead, as in the following invocation:

```
a.app -s -D DEBUG BOOLEAN TRUE example_in.a example_out.a
```

The generated file is:

```
file: /usr2/vads6_10/test/example_out.a

with TEXT_IO;
with DEBUG;
procedure EXAMPLE is
begin
    TEXT_IO.PUT("Debugging host: " & "yet_another_host");
    DEBUG;
end EXAMPLE;
```

“Figures won’t lie, but liars will figure.”

Charles Grosvenor

Statistical Profiler 3

The Statistical Profiler enhances the standard self-host product by providing an accurate description of the CPU usage in all parts of an Ada program, including time spent in the Ada Runtime System. The basic approach is to examine the program counter at regular intervals and keep track of where the program is executing. Solaris kernel support for profiling (via the Solaris profil subroutine, see the Solaris Programmer's Guide, `profil(2)`) makes this method effective and non-intrusive.

This profiling data is displayed using the Solaris prof tool. (see `prof(1)` `man` page). Alternatively, use the `a.prof` tool included in the profiler. Since `a.prof` understands the SC Ada subprogram naming convention, it does a better job of formatting the subprogram names.

Figure 3-1 is an example of profiling output from a.prof.

```
%time     cumsecs name

71.6    0.27    test_prof.tp1

21.1    0.34    test_prof

7.4   0.37  test_prof.tp2
```

Figure 3-1 Example of Profiling Output

Figure 3-2 is an example of source line profiling from a.list.

```
********************** test_prof.a ************************
      1:                function test_prof return integer is
      2:
      3:                        tp_cnt: integer;
      4:
     5:       2.11     function tp2(i: integer) return integer is
      6:                 begin
      7:        1.05         return i + 1;
      8:        4.21       end tp2;
      9:
    10:       5.26    function tp1 return integer is
    11:                 tp1_cnt: integer;
    12:             begin
    13:                 for i in 1..10 loop
    14:       6.32              tp1_cnt := i + 1;
    15:       58.95             end loop;
    16:                 return tp1_cnt;
    17:       1.05    end tp1;
    18:
    19:             begin
    20:                 for i in 1..100000 loop
  21:     5.26                  tp_cnt := tp1;
    22:       8.42              tp_cnt := tp2(tp_cnt);
    23:       6.32          end loop;
  24:           return tp_cnt;
    25:             end test_prof;
```

Figure 3-2 Example of Source Line Profiling from a.list

Figure 3-3 is an example of source line profiling from a.das.

```
Unit:           test_prof
Library:        .
Object file:    /vc/brp/.objects/test_prof01
Source_file:    /vc/brp/test_prof.a

Text Section:

   5 2.11     function tp2(i: integer) return integer is
      00000:  addiu      t0,sp,0fff0       t0 <- sp - 16
      00004:  sltu       t1,t0,s7          t1 <- t0 < s7
      00008:  beq        t1,$0,8           -> 014
      0000c: #nop
      00010:  break      02400
      00014:  addu       sp,t0,$0          sp <- t0

 7 1.05         return i + 1;
      00018:  addi       v0,a0,01          v0 <- a0 + 1

8 4.21     end tp2;
      0001c:  addiu      t0,sp,010         t0 <- sp + 16
      00020:  nop
      00024:  addu       sp,t0,$0          sp <- t0
      00028:  jr         ra
      0002c: #nop

 10 5.26     function tp1 return integer is
      00030:  addiu      t0,sp,0fff0       t0 <- sp - 16
      00034:  sltu       t1,t0,s7          t1 <- t0 < s7
      00038:  beq        t1,$0,8           -> 044
      0003c: #nop
      00040:  break      02400
      00044:  addu       sp,t0,$0          sp <- t0

  13          for i in 1..10 loop
      00048:  addi       t0,$0,01          t0 <- 1

  14 6.32             tp1_cnt := i + 1;
      0004c:  addi       v0,t0,01          v0 <- t0 + 1
```

```
(Continued)

 15 58.95         end loop;
     00050:  addu       t0,v0,$0          t0 <- v0
     00054:  addi       t2,$0,0a          t2 <- 10
     00058:  slt        t1,t2,v0          t1 <- t2 < v0
     0005c:  beq        t1,$0,-20         -> 04c
     00060: #nop
 17 1.05     end tp1;
     00064:  addiu      t0,sp,010         t0 <- sp + 16
     00068:  nop
     0006c:  addu       sp,t0,$0          sp <- t0
     00070:  jr         ra
     00074: #nop
  1  function test_prof return integer is
     00078:  addiu      t0,sp,0ffe8       t0 <- sp - 24
     0007c:  sltu       t1,t0,s7          t1 <- t0 < s7
     00080:  beq        t1,$0,8           -> 08c
     00084: #nop
     00088:  break      02400
     0008c:  addu       sp,t0,$0          sp <- t0
     00090:  sw         s0,00(sp)         0(sp) <- s0
     00094:  sw         fp,04(sp)         4(sp) <- fp
     00098:  sw         ra,08(sp)         8(sp) <- ra
     0009c:  addiu      fp,sp,018         fp <- sp + 24
     000a0:  sw         fp,010(sp)        16(sp) <- fp

 20      for i in 1..100000 loop
     000a4:  addi       s0,$0,01          s0 <- 1

 21 5.26         tp_cnt := tp1;
     000a8:  jal        030
     000ac: #nop

 22 8.42         tp_cnt := tp2(tp_cnt);
     000b0:  addu       a0,v0,$0          a0 <- v0
     000b4:  jal        00
     000b8: #nop
```

```
(Continued)

  23 6.32     end loop;
      000bc:  addiu      s0,s0,01          s0 <- s0 + 1
      000c0:  lui        t0,02             t0 <- 020000
      000c4:  addiu      t0,t0,086a0       t0 <- t0 - 31072
      000c8:  slt        t1,t0,s0          t1 <- t0 < s0
      000cc:  beq        t1,$0,-40         -> 0a8
      000d0: #nop

  25  end test_prof;
      000d4:  addu       t0,fp,$0          t0 <- fp
      000d8:  lw         s0,-018(t0)       s0 <- -24(t0)
      000dc:  lw         fp,-014(t0)       fp <- -20(t0)
      000e0:  lw         ra,-010(t0)       ra <- -16(t0)
      000e4:  nop
      000e8:  addu       sp,t0,$0          sp <- t0
      000ec:  jr         ra
      000f0: #nop
      000f4:  nop
      000F14:  nop
      000fc:  nop
```

Figure 3-3 Example of Source Line Profiling from a.das

References:

`a.prof`, *SPARCompiler Ada Reference Guide*

Table 3-1 is an example of profiling output for the Dhrystone benchmark.

Table 3-1 Profiling Output for Dhrystone Benchmarks

%time	cumsecs	name
20.6	5.57	pack_1.proc_1
20.5	11.09	pack_1.proc_0
13.6	14.75	pack_2.proc_8
13.3	18.34	pack_2.func_2
7.2	20.29	pack_2.proc_7
5.3	21.71	pack_2.func_1
4.6	22.94	pack_2.proc_6
4.2	24.08	pack_1.proc_3
3.3	24.97	pack_1.proc_2
2.8	25.73	pack_1.proc_4
2.3	26.35	global_def.record_type..SIZE
1.3	26.69	pack_2.func_3
0.9	26.92	pack_1.proc_5
0.0	26.93	_getrusage
0.0	26.94	c_strings.c_length
0.0	26.95	file_support.putchar
0.0	26.96	text_io.put_string_on_one_line
0.0	26.97	_write
0.0	26.98	_ioctl

3.1 Linking and Running Profiled Programs

Before you can profile your program execution, the `profile_conf` Ada library must be inserted at the beginning of your ADAPATH. Use the `a.path` tool in your application directory as follows:

```
% a.path -i ada_location/self/profile_conf
```

Now link using `a.ld` and execute your program. At the conclusion of program execution, the file, `mon.out`, is created or overwritten in your current directory. Execute `a.prof` to display the profile performance measurements.

For example:

```
% a.ld my_prog
% a.out
% a.prof
```

3.2 Profiling, How To Do It

Because profiling is statistical in nature, you do not have to recompile your program in order to get profiling results.

1. **Run your program. When your program finishes executing it creates a file called `mon.out`.**

```
% test_prof
```

2. **Use a.prof to perform the analysis of the profiling information. (A sample analysis is provided in Table 1.)**

```
% a.prof test_prof mon.out
```

You can obtain profiling data on a source line basis. The `-d` (disassembly) option to `a.prof` generates source line profiling information and leaves it in an ASCII file called `mon.out`. The `mon.out` file is interpreted by `a.das` and `a.list`.

3.3 `profile_conf` *Directory*

The default user library configuration in the self host product, `usr_conf`, has been enhanced to capture and write profiling data. `v_usr_conf_b.a` contains calls to the subprograms in the profile package for starting, stopping, and writing profiling information.

`profile.a` and `profile_b.a` contain the subprograms for creating the `mon.out` file. These subprograms call the Solaris profile routine for starting and stopping profiling. Refer to the Solaris Programmer's Manual, `profil(2)`

References.

changing configuration parameters, *SPARCompiler Ada User's Guide*, Appendix A User Library Configuration

"But I was thinking of a way
To multiply by ten,
And always, in the answer, get
The question back again."

Anonymous

Machine Code Insertions 4

We provide rolo `package MACHINE_CODE` described in Section 13.8 of the *Ada Reference Manual*. The Reference Manual specifies the format of a machine code insertion procedure, but leaves the actual implementation to the compiler vendor. `package MACHINE_CODE` introduces minimal memory overhead. `package MACHINE_CODE` optimizes offsets, but does you cannot select the form. Rather, the compiler chooses the optimal form.

Machine code insertions provide low-level access to the processor from in Ada, which is normally available only from assembly language.

Using machine code insertions is necessary in programs that must reference hardware directly. Machine code insertions are useful for optimizing time-critical sections beyond the capabilities of the compiler.

SC Ada machine code insertions provide features such as the `X'REF` attribute, a full range of addressing modes and parameters that enable the programmer to integrate the machine code into surrounding code with minimal effort.

However, machine code insertions are a non-portable, processor-dependent feature of the language. The compiler cannot perform certain types of error checks that are performed for normal procedures (such as the enforcement of strong-typing). Use machine code insertions with discretion and only where absolutely necessary.

4

4.1 Machine Code Procedures

A machine code procedure is restricted to the following (as imposed by the *Ada Reference Manua*l):

- Machine code insertions are allowed only in a procedure body.
- The only declarations that can occur in the declarative section of a machine code procedure are `use` clauses.
- The only statements that can occur in the body of a machine code procedure are code-statements (labeled or not).
- A machine code procedure cannot have an exception handler.

The syntactic form is:

```
procedure identifier [formal_part] is
    {use_clause}
begin
    {label} code_statement {{label} code_statement}
end [identifier];
```

The body of a machine code procedure must be in the context of a `with` of `package MACHINE_CODE` (provided in the `standard` library).

Even though this is only a limited form of a subprogram body, use it where a subprogram body is allowed, even as a generic or as a separate body. No restrictions exist on the parameter profile. As with other subprograms, `pragma INLINE` can be applied.

4.2 Code statements

A code statement specifies the machine instruction and any operands needed by the instruction. The form of this construct is:

```
type_mark'record_aggregate;
```

The `type_mark` must be a record type defined in the predefined `package MACHINE_CODE`. All the usual rules apply in forming an aggregate and all type checking is enforced. For single components, the aggregate must use named notation.

Ada `package MACHINE_CODE` provides the variant record types `CODE_0`, `CODE_1`, `CODE_2`, `CODE_3`, `CODE_4`, `DATA_1`, and `DATA_N`. The `CODE_`*x* types have a variant of `type OPCODE` and *x* components of `type OPERAND`. The

DATA_*x* types have a variant of type SIZE. DATA_1 has one OPERAND component, while DATA_N has a component that is a variable number of OPERANDs. The types OPCODE, OPERAND, and SIZE are declared in package MACHINE_CODE also.

References

aggregates, section 4.3(4) in *Ada Reference Manual*

4.2.1 Opcodes

type OPCODE is an enumeration type declared in package MACHINE_CODE. This type provides all of the instructions for the SPARC machine, including the additional pseudo instructions described later. Instructions are named by their standard mnemonics, except for the instructions and, or, not, and xor, which are appended by _op, since each root is identical to an Ada reserved word.

4.2.2 Operands

type OPERAND is a private type declared in package MACHINE_CODE. This package provides constants of type OPERAND and functions that return values of type OPERAND. The SC Ada-defined attribute X'REF can be applied to most Ada objects and denotes a value of type OPERAND.

All SPARC registers are provided as OPERAND constants.

All SPARC addressing modes are made available by functions that denote values of type OPERAND. The following table summarizes the available addressing modes and the functions used to support them. See Table 4-1.

Table 4-1 Machine Code Operands

Mode	Notation	Function
Register	%reg	N/A
Memory Address	[address]	ADDR(address)
Memory Address ASI	[regaddr] asi	ADDR(regaddr, asi)
Register Displacement	%reg + %reg %reg + const13 %reg - const13	"+"(reg, reg) "+"(reg, const13) "-"(reg, const13)

Table 4-1 Machine Code Operands (Continued)

Mode	Notation	Function

Immediate	`value`	`IMMED(value)`
		`"+"(value)`
		`"-"(value)`

Extract High 22 Bits	`%hi(value)` `%hi(value + disp)`	`HI(value)` `HI(name, disp)`
Extract Low 10 Bits	`%lo(value)` `%lo(value + disp)`	`LO(value)` `LO(name, disp)`
External Symbol	`name` `name + disp`	`EXT(name)` `EXT(name, disp)`

A `regaddr` and `address` are formed as follows:

```
regaddr  :  reg
         |  reg + reg
address  :  regaddr
         |  reg + const13
         |  reg - const13
         |  const13
```

const13 is a signed constant that fits in 13 bits:

```
const13  :  IMMED(value)
         |  +value
         |  -value
         |  LO(value [, disp])
```

The unary operators `HI` and `LO` accept the following kinds of arguments:

- String (denoting an external name)
- X'REF
- Integer expression

The unary operators `HI`, `LO`, and `EXT`, allow an additional displacement to be added to an external symbol.

The unary operator HI is allowed only in a sethi instruction. The unary operator EXT is allowed only in a call and set instruction. See Figure 4-1.

```
CODE_2'(LD,         ADDR(16#FFF#), G2);
CODE_2'(LD,         ADDR(G3), G4);
CODE_2'(LDA,        ADDR(G3, 3), G4);
CODE_2'(LD,         ADDR(G1+G3), G2);
CODE_2'(LD,         ADDR(G1+16#FFF#), G2);
CODE_2'(LD,         ADDR(G1+LO("_main")), G2);
CODE_2'(LD,         ADDR(G1+LO(LABEL'REF)), G2);
CODE_2'(SETHI,      +16#3FFFFF#, G1);
CODE_2'(SETHI,      HI(16#FFFFFC01#), G1);
CODE_3'(OR_OP,      G1, LO(16#FFFFFC01#), G2);
CODE_1'(CALL,       EXT("_main"));
CODE_2'(SET,        EXT("_main"), R1);
CODE_1'(RETT,       G1+G2);
```

Figure 4-1 Example of Unary Operators

All arguments to machine code functions must be one of the following:

- Static expression
- Type conversion (the expression operand must be a static expression)
- String literal
- Representation attribute
- X'REF attribute
- Entity defined in package MACHINE_CODE

References

static expression, section 4.9, and representation attribute, section 4.3(4), in *Ada Reference Manual*

4.2.3 Ada Entities as Operands

Sometimes, you must reference Ada constants and variables from a machine code insertion procedure. It is very tedious and error-prone for a programmer to attempt to calculate these references by hand. SC Ada provides `X'REF`, which generates a reference to the entity X. The definition is similar to the attribute `X'ADDRESS`.

For a prefix X that denotes an object, a program unit, a label or an enumeration literal, `X'REF`, yields the reference of the first of the storage units allocated to X. For a constant object with a static expression, the value refers to the static expression. For a subprogram or label, the value refers to the machine code associated with the corresponding body or statement. For an enumeration literal, the value refers to the position number. The value of this attribute is of the `type OPERAND` defined in `package MACHINE_CODE`. It is allowed only in the context of a machine code procedure.

In some cases, using `X'REF` generates more than a single instruction.
See Figure 4-2.

Given:	`type STRING_POINTER is access STRING(1 .. 3);` `X: STRING_POINTER := new STRING'("ABC");`
Then:	`CODE_2'(LDSB, X.all(2)'REF, G1);`

Generates:	`ld [%fp-04], %g1`
	`ld [%g1-08], %g1`
	`subcc %gl, %g0, %g0`
	`bne .L`
	`#nop`
	`call RAISE_CONSTRAINT_ERROR`
	`#nop`
	`L:`
	`ldsb [%g1+01], %g1`
	
	When X is constant static, the value used is an immeadiate literal.

Given:	`X : constant integer := 5;`
Then:	`CODE_2'(SET, X'REF, G1);`
Generates:	or %g0, +05, %g1
	Notes - If `pragma EXTERNAL_NAME` is applied to `X`, then a reference to `X` is used instead. Use `X'REF` to generate references to the procedure parameters

Given:	`procedure SHIFT_RIGHT(`
	`ELEMENT: in INTEGER;`
	`COUNT : in INTEGER;`
	`RESULT : out INTEGER) is`
	`begin`
	`CODE_2'(SRL, COUNT'REF, ELEMENT'REF, RESULT'REF);`
	`end SHIFT_RIGHT;`

Generates	`(preamble and postamble code generation omitted):` `srl %g2, %g3, %g1`

	When the X'REF attribute is applied to labels, the addressing mode generated is absolute or a branch displacement, depending on the instruction. For example, the following instructions generate an absolute addressing mode for LABEL'REF.
CODE_2'	(SET, LABEL'REF, R1); -- address of LABEL
	The following instruction uses a branch displacement.
CODE_1'	(BE, LABEL'REF); -- branch to LABEL
	When the X'REF attribute is applied to subprograms, the addressing mode generated is absolute.
CODE_1'	(CALL, SUBP'REF);

Figure 4-2 Example of Ada Entities as Operands

4.3 Program Control

OPERANDs control the flow of execution with instructions such as Bicc and CALL. Use labels and subroutine names in conjunction with the X'REF attribute to form destinations for these instructions.

The following example illustrates a typical start-up routine for an Ada program. Its function is to call an initialization routine, elaborate the library units, call the main program, and call an exit routine.

See Figure 4-3.

```
with MACHINE_CODE;
procedure START is
   use MACHINE_CODE;
   pragma implicit_code(OFF);
begin
   -- Set stack limit in %G4
   CODE_2'(SETHI,  HI("USER_STACK_SIZE"), G1);
   CODE_2'(LD,     ADDR(G1+LO("USER_STACK_SIZE")), G4);
   CODE_3'(SUB,    SP, G4, G4);

   -- Save a pointer to the args and environment, which starts
   -- at %sp+64
   CODE_3'(ADD,    SP, +64, G3);
   CODE_2'(SETHI,  HI("__u_mainp"), G1);
   CODE_2'(ST,     G3, ADDR(G1+LO("__u_mainp")));
   CODE_2'(LD,     ADDR(G3), G1);  -- argc
   CODE_3'(SLL,    G1, +2, G1);
   CODE_2'(INC,    +8, G1);
   CODE_3'(ADD,    G1, G3, G1);
   CODE_2'(SETHI,  HI("_environ"), G2);
   CODE_2'(ST,     G1, ADDR(G2+LO("_environ")));

  -- Call the package elaboration routines in ELABORATION_TABLE
   -- The address of __stop is the end of the call stack.

   CODE_2'(SET,    STOP_LAB'REF, G1);
   CODE_2'(SETHI,  HI("__stop"), G2);
   CODE_2'(ST,     G1, ADDR(G2+LO("__stop")));
   CODE_2'(SET,    EXT("ELABORATION_TABLE"), L0);

<<ELAB>>
   CODE_2'(LD,     ADDR(L0), L1);
   CODE_2'(CMP,    L1, +0);
   CODE_1'(BE,     DONE'REF);
   CODE_0'(OP => NOP);
   CODE_1'(CALL,   L1);
   CODE_0'(OP => NOP);
<<STOP_LAB>>
   CODE_2'(INC,    +4, L0);
   CODE_1'(BA,     ELAB'REF);
<<DONE>>
   CODE_1'(CLR,    O0);
   CODE_1'(CALL,   EXT("__exit"));
end START;
```

Figure 4-3 Example of Typical Start-up Routine

4.4 Subprogram Call

Make a general form of a subroutine call from in a machine code procedure by using a `CALL` statement. `CALL_0`, `CALL_1`, and `CALL_N` handle the various argument lists needed by a subroutine. The first argument to the `CALL` statement is the name of the subroutine suffixed by `'REF` to yield a value of `type OPERAND`. All subsequent arguments, if any, must be of `type OPERAND`. Number of arguments and their type compatibility are checked. The three forms are:

```
CALL_0'(SUBP => FOO'REF);

CALL_1'(FOO'REF, ARG'REF);

CALL_N'(FOO'REF, (ARG1'REF, ..., ARGn'REF));
```

If `foo` is a procedure, the following Ada procedure calls are equivalent to the statements shown above.

```
FOO;

FOO(ARG);

FOO(ARG1, ..., ARGn);
```

If `foo` is a function, the forms are similar, but the result is not assigned.

If `foo` is overloaded, an error generates when using `'REF`. The subroutine must renamed to provide a distinct name.

The current limitation on the arguments is that they must be of the form `NAME'REF`. No expression is supported, although the expression can be assigned to the name before the call.

The effect is to push the arguments on the stack (if any), make the call, copy the out parameters (if any), and adjust the `sp` (if necessary). The return value of a function is not copied.

If `pragma INLINE` is indicated for `foo`, `foo` is expanded inline.

4.5 Parameter Passing in Machine Code Subprograms

On RISC machines, Ada passes one or more parameters in registers. You must understand exactly how registers are used in parameter passing, especially if you are implementing `machine_code` subprograms using `'REF` on parameters. Using `'REF` on a parameter in a register in an instruction and a memory reference is required, causes the compiler to flag an error. Likewise, using `'REF` on a parameter in a memory location in an instruction and a register is required, causes the compiler to flag an error.

`package MACHINE_CODE` expects references to parameters via `'REF` to be consistent with the register usage rules outlined in Appendix F, "Implementation-Dependent Characteristics." For example, on SPARC systems, the compiler passes the first 6 scalar parameters in registers `o0-o5`. `ld` moves a value from a memory location a register while the SC Ada `mov:` is the equivalent of moving a value from one register to a another.

See Figure 4-4.

```
procedure test_machine_code (p1, p2, p3, p4, p5, p6, p7 : integer) is
begin
    code_2'(ld, p1'ref, g4);              -- (A) put p1 into register g1
    code_2'(mov, p1'ref, g4);             -- (B) put p1 into register g1
    code_2'(ld, p7'ref, g2);              -- (C) put p7 into register g2
    code_2'(mov, p7'ref, g2);             -- (D) put p7 into register g2
     ...
```

Figure 4-4 Example of Parameter Passing in Machine Code Insertions

Since the first 6 scalar parameters are passed in registers, `p1` is in a register, while `p7` is on the stack. Therefore, (B) and (C) are legal, while (A) and (D) flag `p1`/`p7` as being illegal operands.

Caution – If you inline a machine code procedure, the parameters must be referenced using the `'REF` attribute.

4.6 Local Data

Reference variables that are visible to a machine code procedure (either in packages or enclosing subprograms) with `X'REF`. In some cases, however, it is necessary to intermix data and generated code. The `DATA_1` code-statement places a single data item in the code, while the `DATA_`*N* code-statement places multiple data items.

An operand is restricted to the following:

- Immediate
- Absolute
- External symbol
- Label reference
- Subprogram reference

See Figure 4-5

```
DATA_1'(WORD, IMMED(ASCII.LF));
DATA_1'(WORD, ABSOL(16#EFFFF0#));
DATA_N'(WORD, (LABEL1'REF, LABEL2'REF, LABEL3'REF));
```

Figure 4-5 Example of Local Data

4.6.1 Jump Table via Absolute Addresses

A jump table is constructed by building a table of absolute addresses. The table is built by using the data statement, where the operands consist of label references to the selected entry points. Use an absolute address mode specifying the physical address also. Figure 4-6 illustrates the technique.

```
procedure EXAMPLE (INDEX: INTEGER) is
begin

   -- Assume INDEX has the values 0, 4, ..., n*4

   CODE_2'(LD,   INDEX'REF, O0);
   CODE_2'(SET,  TABLE'REF, O1);
   CODE_2'(LD,   ADDR(O0+O1), O0);
   CODE_1'(JMP,  O0);
   CODE_0'(OP => NOP);

<<TABLE>>
   DATA_1'(WORD,     L0'REF);
   DATA_1'(WORD,     L1'REF);
   ...
   DATA_1'(WORD,     Ln'REF);

<<L0>>
   ...
   CODE_1'(BA,   DONE'REF);
<<L1>>
   ...
   CODE_1'(BA,   DONE'REF);
<<Ln>>
   ...
   CODE_1'(BA,   DONE'REF);

<<DONE>>
   CODE_0'(OP => NOP);

end EXAMPLE;
```

Figure 4-6 Example of Jump Table via Absolute Address

Notice how the last statement of each entry code segment is a branch to "DONE." Replacing the branch with a return instruction is not correct, since the epilogue code does not execute.

4.7 Pragmas

pragmas INLINE, SUPPRESS, IMPLICIT_CODE, and OPTIMIZE_CODE directly affect the generation of machine code.

4.7.1 pragma INLINE

This pragma has its normal effect of causing the routine to be expanded inline where called, rather than generating call/return instructions to a single body of code.

4.7.2 pragma SUPPRESS

pragma SUPPRESS has its normal effect of suppressing the generation of runtime checks. Generate runtime checks to perform an elaboration check at the start of the procedure and constraint checks on objects accessed using X'REF. These checks ensure only that the reference is valid; they do not check whether the assigned value is valid. For example, the code-statement

```
CODE_2'(STB, G1, X.all'REF);
```

performs an access check to ensure that X is not null, but does not check to ensure that the value moved to the referenced location is in the range of type X.all.

4.7.3 pragma IMPLICIT_CODE

pragma IMPLICIT_CODE controls the generation of implicit code. Implicit code is code generated for procedure entry and exit to support the calling conventions used by the compiler. (This does not include the return instruction, which always generates unless pragma INLINE is used.) Implicit code includes any additional code generated because of the use of the X'REF attribute (such as code to load a base register).

When pragma IMPLICIT_CODE(OFF) is specified, any stack allocation and the storage_check normally generated for the stack allocation are not generated.

Implicit code always generates for a X'REF attribute that requires it. A warning message generates if IMPLICIT_CODE(OFF) is specified in such a case.

4.7.4 *x* pragma OPTIMIZE_CODE

pragma OPTIMIZE_CODE enables the programmer to specify whether the compiler should attempt to optimize through the machine code insertions. When pragma OPTIMIZE_CODE(OFF) is specified, the compiler generates the code as specified.

Place the pragma in the declarative section of the machine code procedure.

4.8 Debugging Machine Code

The SC Ada debugger supports source-level debugging of machine code insertions. Set breakpoints at code-statements just like any other statements.

Determine register values with reg or by preceding the register name with a dollar sign and using either p or a word dump raw memory command. For example, examine register r1 as a word decimal value using the line-mode command shown below.

```
$r1:Ld
```

Use li and wi to disassemble the generated code. In addition, the debugger attempts to disassemble DATA_*x* statements as SPARC instructions, producing meaningless results.

4.9 Pseudo Instructions

A set of pseudo instructions are supported that map to hardware instructions, as Table 4-2 describes.

Table 4-2 Pseudo Instruction Mapping

Pseudo Instruction	Hardware Equivalent(s)
nop	sethi 0, %g0
cmp reg, reg_or_imm	subcc reg, reg_or_imm, %g0
jmp address	jmpl address, %g0
call reg_or_imm	jmpl reg_or_imm, %o7
jmpl label, %o7	call label
tst reg	orcc reg, %g0, %g0

Table 4-2 Pseudo Instruction Mapping (Continued)

Pseudo Instruction	**Hardware Equivalent(s)**
`ret`	`jmpl j%i7+8, %g0`
`retl`	`jmpl j%o7+8, %g0`
`restore`	`restore %g0, %g0, %g0`
`save`	`save %g0, %g0, %g0`
`not reg1, reg2`	`xnor reg1, %g0, reg2`
`not reg`	`xnor reg, %g0, reg`
`neg reg1, reg2`	`sub %g0 , reg1, reg2`
`neg reg`	`sub %g0, reg, reg`
`inc reg`	`add reg, 1, reg`
`inc const13, reg`	`add reg, const13, reg`
`inccc reg`	`addcc reg, 1, reg`
`dec reg`	`sub reg, 1, reg`
`dec const13, reg`	`sub reg, const13, reg`
`deccc reg`	`subcc reg, 1, reg`
`btst reg_or_imm, reg`	`andcc reg, reg_or_imm, %g0`
`bset reg_or_imm, reg`	`or reg, reg_or_imm, reg`
`bclr reg_or_imm, reg`	`andn reg, reg_or_imm, reg`
`btog reg_or_imm, reg`	`xor reg, reg_or_imm, reg`
`clr reg`	`or %g0, reg_or_imm, reg`
`clrb [address]`	`stb %g0, [address]`
`clrh [address]`	`sth %g0, [address]`
`clr [address]`	`st %g0, [address]`
`mov reg_or_imm, reg`	`or %g0, reg_or_imm, reg`
`mov %y, reg`	`rd %y, reg`
`mov %psr, reg`	`rd %psr, reg`
`mov %wim, reg`	`rd %wim, reg`
`mov %tbr, reg`	`rd %tbr, reg`

Table 4-2 Pseudo Instruction Mapping (Continued)

Pseudo Instruction		Hardware Equivalent(s)	
mov	reg_or_imm, %y	wr	%g0, reg_or_imm, %y
mov	reg_or_imm, %psr	wr	%g0, reg_or_imm, %psr
mov	reg_or_imm, %wim	wr	%g0, reg_or_imm, %wim
mov	reg_or_imm, %tbr	wr	%g0, reg_or_imm, %tbr
set	value, reg (if -4096 <= value <= 4095)	or	%g0, value, reg
set	value, reg (if ((value&0x1ff) == 0))	sethi	%hi(value), reg
set	value,reg	sethi	%hi(value), reg;
		or	reg, %lo(value), reg
	(otherwise)		

4.10 package MACHINE_CODE

Figure 4-7 shows an example of package MACHINE_CODE.

```
package MACHINE_CODE is

  -- Description for the SPARC.

type opcode is (

      bn, fbn, cbn, be, fbne, cb123, ble, fblg,cb12, bl, fbul, cb13,
      bleu, fbl, cb1, bcs, fbug, cb23, bneg, fbg, cb2, bvs, fbu,
      cb3, ba, fba, cba, bne, fbe, cb0, bg, fbue, cb03, bge, fbge,
      cb02, bgu, fbuge, cb023, bcc, fble, cb01, bpos, fbule, cb013,
      bvc, fbo, cb012, bn_a, fbn_a, cbn_a, be_a, fbne_a, cb123_a,
      ble_a, fblg_a, cb12_a, bl_a, fbul_a, cb13_a, bleu_a, fbl_a,
      cb1_a, bcs_a, fbug_a, cb23_a, bneg_a, fbg_a, cb2_a, bvs_a,
      fbu_a, cb3_a, ba_a, fba_a, cba_a, bne_a, fbe_a, cb0_a, bg_a,
      fbue_a, cb03_a, bge_a, fbge_a, cb02_a, bgu_a, fbuge_a,
      cb023_a, bcc_a, fble_a, cb01_a, bpos_a, fbule_a, cb013_a,
      bvc_a, fbo_a, cb012_a, call, cpop1, cpop2, nop, ret, retl,
      clrb, clrh, jmp, clr, dec, set, inc, deccc, inccc, tst, neg,
      not_op, rd, cmp, wr, bclr, bset, btog, btst, mov, fmovs,
      fnegs, fabss, fints, fintd, fintx, fintrzs, fintrzd, fintrzx,
      fsqrts, fsqrtd, fsqrtx, fadds, faddd, faddx, fsubs, fsubd,
      fsubx, fmuls, fmuld, fmulx, fdivs, fdivd, fdivx, frems, fremd,
```

```
      fremx, fquots, fquotd, fquotx, fscales, fscaled, fscalex,
      fstoir, fdtoir, fxtoir, fitos, fdtos, fxtos, fitod, fstod,
      fxtod, fitox, fstox, fdtox, fstoi, fdtoi, fxtoi, fclasss,
      fclassd, fclassx, fexpos, fexpod, fexpox, fcmps, fcmpd,
      fcmpx, fcmpes, fcmped, fcmpex, add, and_op, or_op, xor_op,
      sub, andn, orn, xnor, addx, subx, addcc, andcc, orcc, xorcc,
      subcc, andncc, orncc, xnorcc, addxcc, subxcc, tadd, tsub,
      taddcctv, tsubcctv, mulscc, sll, srl, sra, rdy, rdpsr, rdwim,
      rdtbr, wry, wrpsr, wrwim, wrtbr, jmpl, rett, iflush, save,
      restore, ld, ldub, lduh, ldd, st, stb, sth, std, ldsb, ldsh,
      ldstub, swap, lda, lduba, lduha, ldda, sta, stba, stha, stda,
      ldsba, ldsha, ldstuba, swapa, ldf, ldfsr, lddf, stf, stfsr,
      stdfq, stdf, ldc, ldcsr, lddc, stc, stcsr, stdcq, stdc,
      unimpl, sethi, tn, te, tle, tl, tleu, tcs, tneg, tvs, ta, tne,
      tg, tge, tgu, tcc, tpos, tvc);
type size is (word);

type operand is private;

type operand_seq is array (positive range <>) of operand;
n: positive;

 --
 --  Instruction formats.
 --

type code_0 (op: opcode) is
   record
     null;
   end record;

type code_1 (op: opcode) is
   record
     oprnd_1: operand;
   end record;

type code_2 (op: opcode) is
   record
     oprnd_1: operand;
     oprnd_2: operand;
   end record;

type code_3 (op: opcode) is
   record
     oprnd_1: operand;
```

```
      oprnd_2: operand;
      oprnd_3: operand;
    end record;

 type code_4 (op: opcode) is
    record
      oprnd_1: operand;
      oprnd_2: operand;
      oprnd_3: operand;
      oprnd_4: operand;
    end record;

  --
  --  Data formats.
  --

 type data_1 (sz: size) is
    record
      oprnd_1: operand;
    end record;


 type data_n (sz: size) is
    record
      oprnd_n: operand_seq (1..n);
    end record;

    --
  --  Call formats.
  --

 type call_0 is
    record
      subp: operand;
    end record;

type call_1 is
    record
      subp:    operand;
      oprnd_1: operand;
    end record;

 type call_n is
    record
      subp:    operand;
      oprnd_n: operand_seq (1..n);
```

```
  end record;

  --
  --  Registers.
  --

-- Integer registers.
r0:  constant operand;
r1:  constant operand;
r2:  constant operand;
r3:  constant operand;
r4:  constant operand;
r5:  constant operand;
r6:  constant operand;
r7:  constant operand;
r8:  constant operand;
r9:  constant operand;
r10: constant operand;
r11: constant operand;
r12: constant operand;
r13: constant operand;
r14: constant operand;
r15: constant operand;
r16: constant operand;
r17: constant operand;
r18: constant operand;
r19: constant operand;
r20: constant operand;
r21: constant operand;
r22: constant operand;
r23: constant operand;
r24: constant operand;
r25: constant operand;
r26: constant operand;
r27: constant operand;
r28: constant operand;
r29: constant operand;
r30: constant operand;
r31: constant operand;

-- Global registers.
g0:  constant operand;
g1:  constant operand;
g2:  constant operand;
g3:  constant operand;
g4:  constant operand;
```

```
g5:  constant operand;
g6:  constant operand;
g7:  constant operand;

-- In registers.
i0:  constant operand;
i1:  constant operand;
i2:  constant operand;
i3:  constant operand;
i4:  constant operand;
i5:  constant operand;
i6:  constant operand;
i7:  constant operand;

-- Local registers.
l0:  constant operand;
l1:  constant operand;
l2:  constant operand;
l3:  constant operand;
l4:  constant operand;
l5:  constant operand;
l6:  constant operand;
l7:  constant operand;

-- Out registers.
o0:  constant operand;
o1:  constant operand;
o2:  constant operand;
o3:  constant operand;
o4:  constant operand;
o5:  constant operand;
o6:  constant operand;
o7:  constant operand;

fp:  constant operand;  -- i6
sp:  constant operand;  -- o6

-- Floating point registers.
f0:  constant operand;
f1:  constant operand;
f2:  constant operand;
f3:  constant operand;
f4:  constant operand;
f5:  constant operand;
f6:  constant operand;
f7:  constant operand;
```

```
f8:  constant operand;
f9:  constant operand;
f10: constant operand;
f11: constant operand;
f12: constant operand;
f13: constant operand;
f14: constant operand;
f15: constant operand;
f16: constant operand;
f17: constant operand;
f18: constant operand;
f19: constant operand;
f20: constant operand;
f21: constant operand;
f22: constant operand;
f23: constant operand;
f24: constant operand;
f25: constant operand;
f26: constant operand;
f27: constant operand;
f28: constant operand;
f29: constant operand;
f30: constant operand;
f31: constant operand;
-- Coprocessor registers.
c0:  constant operand;
c1:  constant operand;
c2:  constant operand;
c3:  constant operand;
c4:  constant operand;
c5:  constant operand;
c6:  constant operand;
c7:  constant operand;
c8:  constant operand;
c9:  constant operand;
c10: constant operand;
c11: constant operand;
c12: constant operand;
c13: constant operand;
c14: constant operand;
c15: constant operand;
c16: constant operand;
c17: constant operand;
c18: constant operand;
c19: constant operand;
```

```
  c20: constant operand;
  c21: constant operand;
  c22: constant operand;
  c23: constant operand;
  c24: constant operand;
  c25: constant operand;
  c26: constant operand;
  c27: constant operand;
  c28: constant operand;
  c29: constant operand;
  c30: constant operand;
  c31: constant operand;

  -- Special registers.
  fsr: constant operand;
  fq : constant operand;
  csr: constant operand;
  cq : constant operand;
  psr: constant operand;
  tbr: constant operand;
  wim: constant operand;
  y  : constant operand;


--
-- Addressing modes.
--

function addr (
     expr: operand)
   return operand;

function addr (
     expr: integer)
    return operand;

    -- Assembler Notation:
    --   [expr]
    --
    -- Description:
    --   The expr denotes the effective address.

function addr (
     expr: operand;
     asi : integer)
    return operand;
```

```
    -- Assembler Notation:
    --   [expr]asi
    --
    -- Description:
    --   The expr denotes the effective address.  The asi specifies
    --   the alternate address space identifier.
function "+" (
     base: operand;
     disp: operand)
    return operand;

function "+" (
     base: operand;
     disp: integer)
    return operand;

function "-" (
     base: operand;
     disp: integer)
    return operand;

    -- Assembler Notation:
    --   reg + reg
    --   reg + const
    --   reg - const
    --
    -- Description:
    --   Displacement is added to the register to form the address.
    --   The base operand must be a general register.  The disp
    --   operand can be a register, a signed immediate constant
    --   (13 bits), or the lo operator.

function immed (
     val: integer)
    return operand;

function immed (
     val: character)
    return operand;

function "+" (
     val: integer)
    return operand;
```

```
function "-" (
     val: integer)
    return operand;

    -- Description:
    --   Immediate literal.

function hi (
     name: string;
     disp: integer := 0)
    return operand;

function hi (
     name: operand;
     disp: integer := 0)
    return operand;

function hi (
     addr: integer)
    return operand;

    -- Assembler Notation:
    --   %hi(name)
    --   %hi(addr)
    --
    -- Description:
    --   Unary operator that extracts high 22 bits of its operand.
    --   The name must either be a string denoting the external name,
    --   an Ada entity that is relocatable, or an integer value.
    --   A displacement is only allowed for a relocatable entity.
    --   This function is only allowed in the sethi instruction.

function lo (
     name: string;
     disp: integer := 0)
    return operand;

function lo (
     name: operand;
     disp: integer := 0)
    return operand;

function lo (
     addr: integer)
    return operand;
```

```
    -- Assembler Notation:
    --   %lo(name)
    --   %lo(addr)
    --
    -- Description:
    --   Unary operator that extracts low 10 bits of its operand.
    --   The name must either be a string denoting the external name,
    --   an Ada entity that is relocatable, or an integer value.
    --   A displacement is only allowed for a relocatable entity.

function ext (
     name: operand;
     disp: integer := 0)
    return operand;

function ext (
     name: string;
     disp: integer := 0)
    return operand;


    -- Description:
    --   The name denotes external symbol. The displacement is added
    --   to the value of name.

private

    --
    -- Implementation specific.
    --

end MACHINE_CODE;
```

Figure 4-7 Example of `package MACHINE_CODE`

> “He that is but able to express no sense at all in several languages will pass for learneder than he that’s known to speak the strongest reason in his own.”
>
> Samuel Butler

Interface Programming 5

Translating programs from other languages into Ada is usually straightforward if the source language is one of the block-structured languages, such as Pascal or C. Clearly structured programs in other languages are not difficult to translate but may require more work because of differences between the source and target languages. However, it is often desirable to make use of sbprograms or libraries developed in some other language from inside Ada programs without having to translate everything into Ada.

This chapter presents (1) an approach to making existing libraries and programs written in C useful from Ada and (2) a discussion of a modular approach to program conversion into Ada. The types of declarations used in the examples in this chapter are similar to those used to solve interface problems on many operating systems.

Note – This chapter discusses a number of Ada restrictions. Unless otherwise noted, these restrictions apply to SPARCompiler Ada, and not necessarily to standard Ada.

5.1 Ada Interface to `curses`

This section presents an Ada interface to the OS-derived `curses` library of compiled C functions for screen formatting. The Ada package for calling the library functions gives the programmer the same functional entities and objects as the original.

We assume that the reader has some familiarity with the `curses` library of cursor motion optimization routines for video terminals. This library provides text input, text output, functions for creating windows, and functions for altering terminal characteristics. It is helpful to have a copy of the operating system manual section on the `curses` library available for reference. The complete source code for the `curses` interface package described in this example is in the `publiclib` library supplied with SC Ada.

The goal in this example is to provide a complete Ada interface to `curses` using the same subprogram and variable names provided in the original C version. Further, a programmer can use the standard documentation for `curses` so no additional effort is needed in order to make use of the library from Ada.

The following five steps are necessary to accomplish this goal:

1. Create parallel data types.
2. Declare external subprograms.
3. Access global variables declared in the original language.
4. Mapping to Parallel Data Structures.
5. Reduce the call overhead of the interface.

5.1.1 Create Parallel Data Types

When access to a subprogram or variable declared in an alternate language is required, it is the programmer's responsibility to ensure that any Ada variables used in conjunction with the subroutine or variable are of a compatible data representation in both languages. For example, the programmer cannot assume that a data structure declared in Ada is identical to a data structure declared in C. Although the type names may be identical, the composition, length or alignment of the object or component may not be. Type definitions in different languages can, if taken at face value, cause erroneous results. For instance, SC Ada `type FLOAT` is not the same as the C `type float` for 32-bit CPUs.

The two basic approaches for creating parallel data types are using *a priori* knowledge and using Ada representation specifications. The programmer knows that some types are parallel between two language implementations from reading the vendor's documentation. However, remember that neither

Ada nor C compilers are required to use a particular size to represent any particular type, and an implementation is free to choose a representation based on hardware considerations.

Ada representation clauses, on the other hand, enable the Ada programmer to define an exact duplicate of the physical layout of any data type in another language once it is known. Ada allows type specifications that are largely independent of the implementation. Type, storage, record layout, and alignment can all be controlled.

When the underlying representation of a type has no analogue in one language (or for which the usage in one language is significantly different from Ada), define the data type using Ada representation specifications and `UNCHECKED_CONVERSION`s.

Simple Types — Some samples of the more common C simple types and their corresponding Ada predefined types are given in Table 5-1. For example, the `type SHORT_INTEGER` is `pcc`-derived C compiler `type short`, both representing a 16-bit integer.

Table 5-1 Simple Types

C	FORTRAN	Ada
`int`	`INTEGER*4`	`INTEGER`
`long`	`INTEGER*4`	`INTEGER`
`short`	`INTEGER*2`	`SHORT_INTEGER`
`char`	`CHARACTER`	`CHARACTER`
		`TINY_INTEGER`
`float`	`REAL`	`SHORT_FLOAT`
`double`	`DOUBLE PRECISION`	`FLOAT`

For the declaration of a C unsigned integer

```
unsigned short u_var;
```

no predefined Ada equivalent exists, and the type must be created using representation clauses.

```
-- an Ada version of the C type: unsigned short
type C_UNSIGNED_SHORT is range 0 .. (2 ** 16) - 1;
for C_UNSIGNED_SHORT'LENGTH use 16;

u_var : C_UNSIGNED_SHORT;
```

The first statement creates a type whose range includes all the values the C type encompasses. The second ensures that at most 16 bits of storage are allocated to every object of this type.

The C `type char` represents a character (usually by ASCII value) or a byte integer value. Some C implementations treat `char` as unsigned and some as signed quantities. No exact Ada analogue exists to this type exists; use SC Ada `TINY_INTEGER` for numeric representations and `type CHARACTER` to represent a character value as illustrated in Table 5-2. When C programs contain ambiguous assignments or use such types or integer/address conversions, the `generic function UNCHECKED_CONVERSION` offers a method for controlled easing of type conversions.

Table 5-2 Type Conversions

C Source	Ada Source

`char number = 20;`	`with UNCHECKED_CONVERSION;`
	`function CONVERT_NO_TO_CHAR is new`
	`  UNCHECKED_CONVERSION(TINY_INTEGER,`
	`  CHARACTER);`
	`function CONVERT_CHAR_TO_NO is new`
	`  UNCHECKED_CONVERSION(CHARACTER,`
	`  TINY_INTEGER);`

`char ch;`	`NUMBER: TINY_INTEGER := 20;` `CH : CHARACTER;`
`ch = number;` `ch += 2;`	`CH := CONVERT_NO_TO_CHAR(NUMBER);` `CH := (CONVERT_CHAR_TO_NO(CH) +2);`

Record Types — The same two basic approaches can be taken to the representation of record types as with simple types. Again, *a priori* knowledge can be used. Both C and SC Ada associate the record label with a base address from which offsets to access individual components of the record are

calculated. For the Ada programmer, as long as the record is composed of equivalent simple data types, the offsets are calculated similarly, and the record structures are identical.

However, occasions arise where storage conventions are not arranged so conveniently. In such cases, Ada representation specification can be given for constructing records. For example, `curses` uses the following C structure.

```
struct sgttyb{
  char sg_ispeed;
  char sg_ospeed;
  char sg_erase;
  char sg_kill;
  short sg_flags;
};
```

Ensure a parallel structure using the following statements in conjunction with the type definitions for `short` and `char` of the type mentioned above:

```
type C_SHORT is range -(2 ** 15) .. (2 ** 15) - 1;
type C_TINY is range -(2 ** 7) .. (2 ** 7) - 1;
type SGTTYB is
record
  SG_ISPEED : C_TINY;
  SG_OSPEED : C_TINY;
  SG_ERASE  : C_TINY;
  SG_KILL   : C_TINY;
  SG_FLAGS  : C_SHORT;
end record;

for SGTTYB use
record
  SG_ISPEED at 0 range 0..7;
  SG_OSPEED at 1 range 0..7;
  SG_ERASE at 2 range 0..7;
  SG_KILL at 3 range 0..7;
  SG_FLAGS at 4 range 0..15;
end record;
```

In this example, the record is first defined in terms of its simple type components and then in terms of storage allocation characteristics. The value that appears directly after the `at` indicates the number of `STORAGE_UNITS` (in this case bytes) from the base address that the element should be placed. The range specifies the bits the type should occupy.

Ada supports an optional clause to specify object alignment. For instance, if you must align objects on two-byte boundaries, use the statement `at mod 2` as shown here:

```
for SGTTYB use
record at mod 2;
  SG_ISPEED at 0 range 0..7;
  SG_OSPEED at 0 range 8..15;
  SG_ERASE at 2 range 0..7;
  SG_KILL at 2 range 8..15;
  SG_FLAGS at 4 range 0..15;
end record;
```

Array Types — When defining Ada array types that are parallel to C array types, remember that the standard representation of an array in both languages is to associate the array label with the first component. Use this location to calculate an offset. This means that as long as the programmer ensures that the individual components are compatible structures, the basic array structure type is the same. When in doubt, use representation specifications to ensure that individual component representations are identical.

Dynamic Array Types — Several languages, including Ada and C, define dynamically-sized arrays. In C, you calculate the size of a dynamic array based on data known only to you. (An exception is the C string type.) SC Ada implements dynamic arrays by keeping “dope vectors” created with each dynamic subtype. The dope vector is placed in memory immediately preceding the array value, so an access to a dynamic array can be used. The dope vector for a single-dimensioned array contains:

```
type dope vector is record
    element_size: implementation-defined;
    low_bound:   implementation-defined;
    high_bound:  implementation-defined;
    array_size: implementation-defined;
end record;
```

We recommend that the SYSTEM.ADDRESS of the first element of an Ada array be used to pass the array to C. C arrays always start at index 0, while Ada arrays start with any index. *element(1)* can name entirely different true elements.

FORTRAN multidimensional arrays are stored in column-major storage format. C and Ada arrays are stored in row-major format. Reverse multidimensional indices between Ada and FORTRAN.

To pass arrays from C to Ada, create an appropriate subtype for the value. If the array is referenced only in a closed scope in Ada, the `for... use at...;` representation control is convenient.

```
procedure deal_with(C_array: system.address; length:integer)is
   the_array: array(integer range)..length-1) of integer;
      for the_array use at C_array;
   -- the_array(0) names C_array[0] of the C caller
```

If a C array is passed and must be preserved over an open scope, for example library, use a fixed-length array in the Ada program, making the Ada array at least as large as any possible C parameter. Suppress index checks since they are redundant. While it is sometimes desirable to create a dynamically-typed reference to the C array, it is impossible without copying the array unless you build an appropriate dope vector. The dope vector is placed in memory immediately preceding the array value, so an access to a dynamic array can be used.

Pointers and Address Types — Although pointer and address types are implementation-specific, the Ada tactic of using host conventions usually allows the use of Ada pointer and address types parallel to their C counterparts. If, for some reason, host conventions are not followed, or if the Ada compiler supports several host implementations of a particular type and must choose a particular representation. Use representation specifications to customize the size and range of the data type.

String Types — String types in C present a special case of the C array. A character string in C is represented by a pointer to the first character in an array of bytes. By convention, strings in C are terminated by a null character (`16#00#`) and store no explicit length. In Ada, however, a string is represented by a packed array of `type CHARACTER` with the maximum number of components specified as part of the type.

A parallel type using the declaration in Ada is shown here:

```
type C_STRING is access STRING (1..INTEGER'LAST);
```

Although this type declaration enables access to C strings, exercise caution when transferring strings between C and Ada variables. For example, a C string is an array of characters, but an Ada string (or any aggregate) has additional array information represented in a dope vector.

5.1.2 Declare External Subprograms

After parallel types are established, the next step is to gain access to subprograms and macros provided in the interface target package. Do this by using functions or macros and then by using the predefined `pragma INTERFACE_NAME` to establish a link from the Ada procedure or function name to the corresponding C function or macro.

`pragma INTERFACE` enables Ada programs to call subroutines defined in other languages: C, ADA, PASCAL, FORTRAN, and UNCHECKED. The definition of `pragma INTERFACE` does not allow access to subprograms whose names are Ada reserved words or cannot be expressed as an Ada identifier.

We added `pragma INTERFACE_NAME`, which gives the exact name of the external subprogram and uses this format:

```
pragma INTERFACE_NAME (Ada_name, link_name);
```

The first parameter is the name of the Ada subprogram, the second is the name of the target subprogram as known to the linker. Form the *link_name* argument from a string literal, a constant string object or a catenation of these operands. This allows a system-independent interface to common routines for different versions of an operating system. For example, we supply the following:

```
pragma INTERFACE_NAME (c_exit, C_SUBP_PREFIX & “exit”)

where

package LANGUAGE is
   C_PREFIX:      constant string := "_";
   C_SUBP_PREFIX: constant string := ".";

   FORTRAN_PREFIX: constant string := "_";
   FORTRAN_SUFFIX: constant string := "_";

   OBJECT_EXTENSION: constant string := ".o";
   LIBRARY_EXTENSION: constant string := ".a";
   EXECUTABLE_EXTENSION: constant string := "";

end LANGUAGE;
```

allows inter-language communication ability with a minimum of effort.

If *Ada_name* denotes a subprogram, a `pragma INTERFACE` must be specified already for the subprogram.

The Ada compiler handles parameter pushing and target language compiler naming conventions and checks to make sure the parameters are allowed in the target language. For example, Ada fixed-point types do not have a C equivalent. Generally, C compilers push parameters from right-to-left and generate assembly instructions in which C subprogram names are usually preceded with an underscore.

Use SC Ada `pragma INTERFACE_NAME` to enable Ada procedures to directly access variables defined in other language modules. For example:

```
pragma INTERFACE_NAME (Ada_variable, link_name);
```

Provide access to external variables by extending the external library with procedural access to these variables, but this pragma allows faster-executing code and does not require additional code in the language of the external library.

Finally, because the interface between an Ada routine and an external routine can involve an intermediate Ada function call, use `pragma INLINE` to eliminate the overhead of the extra call.

Because Ada does not support macros and preprocessing, the C macros must be defined as Ada procedures. However, `pragma INLINE` treats procedures as if they are macros with the benefit of semantic checking. Default parameters are available in Ada so instances of C functions that use macros solely for the purpose of providing for default parameters are written more easily in Ada.

With `curses`, most cursor movement is done using window-specific functions. These window-specific routines define a set of macros that act on the default window `stdscr`, and these routine macros define more macros. For example, the procedure to add a character to a window, `waddch()`, is shown in the following code:

```
waddch(win,ch)
WINDOW win;
char ch;

#define addch(ch) VOID(waddch(stdscr,ch))
#define mvwaddstr(win,y,x,str)
VOID (wmove (win,y,x) == ERR ? ERR : waddstr(win, str))
#define mvaddstr (y, x, str) mvwaddstr (stdscr, y, x, str)
```

Still some problems exist in using `pragma INTERFACE` for subprograms of this type that may not be immediately apparent. First, all parameters must be of mode `in` or “call by value” and functions can return only results that are scalar, access or `SYSTEM.ADDRESS` types. Second, this naming convention enables the Ada programmer to use only the C label for an interface routine. This becomes awkward when the language interfaced to supports different naming conventions than those allowed in Ada or when the creation of an intermediate routine is required to overcome the first restriction while preserving the original calling and naming conventions.

The most effective way to circumvent the first restriction is to pass other types of parameters than those listed by reference, using the predefined address attribute (section 13.7.2 of the *Ada Reference Manual*). Overcome the second restriction using `pragma INTERFACE_NAME` to map between the Ada and C subprogram names.

The use of this pragma and the address attribute are both useful for `waddstr` (window add string). This subprogram involves conflicting data structures and parameter passing techniques. The Ada string is passed as two units, the array itself and the length or dope vector. In contrast, its C counterpart is passed by transferring a pointer to the first character of the string.

References

Find `pragma INTERFACE` usage examples in `examples` directory.

5.1.2.1 Using Intermediate Routines

The most obvious solution is to create an intermediate routine that converts an Ada string input into a C-string format before calling the C routine. `pragma INTERFACE_NAME` facilitates this operation while allowing the procedure to remain unchanged from your standpoint.

Figure 5-1 shows an example using `waddstr`.

```
-- package spec
procedure WADDSTR( WIN: WINDOW; S: STRING );
procedure C_WADDSTR( win: WINDOW; STR: ADDRESS );

pragma INTERFACE( C, C_waddstr);
pragma INTERFACE_NAME(C_waddstr, C_SUBP_PREFIX & "waddstr" );

-- package body
procedure WADDSTR( WIN: WINDOW; S: STRING ) is
  T: STRING(1..(S'LAST + 1));
begin
  T(1..S'LAST) := S;
  T(S'LAST + 1) := ASCII.NUL;
  C_WADDSTR( win, T'ADDRESS );
end WADDSTR;
```

Figure 5-1 Example of Using Intermediate Routines

From the compiler standpoint, the effect of `pragma INTERFACE_NAME` is to substitute the *link_name* given when generating the reference instead of the Ada label.

After the interface correspondence between the C functions is established, begin defining macros in terms of these functions. In our example using the file `curses.h` as a template in conjunction with the interface declarations for `WADDSTR` declared in the last section, we can create the default window counterpart, the Ada version of the macro `ADDSTR`.

A C macro takes the form of a `#define` followed by the macro specifics. In this case, `addstr` is defined as the following.

```
#define addstr(str)  VOID(waddstr(stdscr, str))
```

The variable `str` is already defined as a C string (`*char`) type and `stdscr` is the default window used by `curses`. Using this definition, our Ada procedure counterpart becomes the following.

```
-- package spec
procedure ADDSTR( S: STRING );

-- package body
procedure ADDSTR( S: STRING ) is
begin
  WADDSTR( STDSCR, S );
end ADDSTR;
```

Notice that the type declaration of S is parallel to the Ada `procedure WADDSTR` type declaration defined earlier, and not the C interface type declaration for `str` from `C_waddstr`.

Create the majority of the `curses` screen routines in this manner using intermediate routines and passing addresses or simulating macros when appropriate. A disadvantage to intermediate routines is in the form of additional overhead generated from the call intensive execution of these simulated macros.

References

Section 5.1.5, “Reduce the Overhead,” on page 5-19

5.1.3 Access Global Variables

The third step when building a complete package from a C library is to gain access to global variables declared in C from Ada. Although not supported in the standard language definition, access to these items is essential in a package like `curses` where a number of global variables are provided for programmer use or use by internal routines. This connection is established by the SC Ada compiler with `pragma INTERFACE_NAME`.

An example of this pragma used to access the `curses` global variables `LINES` and `COLS` is shown here:

```
LINES, COLS : INTEGER;
pragma INTERFACE_NAME( LINES, C_prefix & "LINES");
pragma INTERFACE_NAME( COLS , C_prefix & "COLS" );
```

Use `pragma INTERFACE_NAME` to make structures that normally are inaccessible, available to you. For example, the two variables `curscr` and `stdscr` are pointers to `curses` window structures, and are the key to implementing changes on a screen in most of the curses routines. Using this pragma, we can declare the following:

```
Stdscr : SYSTEM.ADDRESS;
Curscr : SYSTEM.ADDRESS;

Pragma  INTERFACE_NAME(STDSCR, C_prefix & “stdscr”);
Pragma  INTERFACE_NAME(CURSCR, C_prefix & “curscr”);
```

The first two statements declare the variables `curscr` and `stdscr` as predefined Ada variables; the latter two associate the same memory locations for the Ada variables as those occupied by the C variables. As noted earlier, the C record access method enables the programmer access to these structures by base address as long as access to internal structure values is not necessary. Any subsequent reference to these labels from Ada refers to the identical data structure currently used by the C subprograms. As far as the compiler is concerned, both forms of this statement are equivalent to generating a reference in the object module for a variable with the linker name `_stdscr` or `_curscr`.

5.1.4 Map to Parallel Data Structures

The fourth step in the Ada interface process is the ability to map parallel data structures.

Accomplish this with `pragma`s `INTERFACE` and `INTERFACE_NAME`. The ability to map parallel data structures is illustrated using the `curses` library package. In this package, the window structure is declared in C as a record containing such information as the current cursor position, certain terminal attributes, and pointers to the actual screen structure.

See Figure 5-2.

```
struct _win_st{
  short _cury,  _curx;
  short _maxy,  _maxx;
  short _begy,  _begx;
  short _flags;
  bool  _clear;
  bool  _leave;
  bool  _scroll;
  char  **_y;
  short *_firstch;
  short *_lastch;
};
```

Figure 5-2 Example of Window Structure in C

To implement the macros `flushok` (`win, bf`), `getyx` (`win, y, x`), and `winch` (`win`) correctly, you must gain access to fields that reside in the window structure. The first step is to create a parallel Ada record to use as a template to place over the C structure. See Figure 5-3.

```
Type WIN_STRUCTURE;
Type WIN_POINTER is access WIN_STRUCTURE;
Type WIN_STRUCTURE is
record
  CURY, CURX   :  SHORT;
  MAXY, MAXX   :  SHORT;
  BEGY, BEGX   :  SHORT;
  FLAGS        :  SHORT;
  CLEAR        :  BOOL;
  LEAVE        :  BOOL;
  SCROLL       :  BOOL;
  YBAR         :  STRING_POINTER;
  FIRST_CH     :  SHORT_POINTER;
  LAST_CH      :  SHORT_POINTER;
  NEXT_TP      :  WIN_POINTER;
  ORIG         :  WIN_POINTER;
end record;

type WINDOW is access WIN_STRUCTURE;
```

Figure 5-3 Example of Parallel Ada Record to C Window Structure

This declaration is followed by type and `INTERFACE_NAME` declarations that essentially create labels for all values stored in the C structure, as illustrated in the following examples:

```
CURSCR : WINDOW
Pragma INTERFACE_NAME(CURSCR, LANGUAGE.C_prefix & “curscr”);

STDSCR : WINDOW
Pragma INTERFACE_NAME(STDSCR, LANGUAGE.C_prefix & “stdscr”);
```

This enables the programmer to declare any subsequent windows that are required as `type WINDOW` and gain similar structure access. Now write procedures like `getyx` as the following.

```
Procedure GETYX(WIN:WINDOW; Y, X: out INTEGER) is
begin
  Y:=INTEGER(WIN.CURY);
  X:=INTEGER(WIN.CURX);
end GETYX;
```

A procedure like `scrollok` becomes the following.

```
procedure SCROLLOK(WIN:WINDOW; BF : BOOLEAN) is
begin
  WIN.SCOLL:=BF;
end SCROLLOK;
```

However, this process does get more complex when implementation of a C macro like `winch` is required. This routine requires extracting a character from the screen at the current cursor location, making access to the screen character storage structure necessary. Using the C WINDOW structure for reference, it becomes clear that the screen is represented by a two-dimensional C array of characters. In Ada, this translates to a pointer to an array of `C_Strings`, which are represented by pointers to arrays of characters terminated by nulls. Although hard to visualize, create this structure with the following declaration.

```
type CHAR_STRING is array(0..149) of TINY_INTEGER;
type STRING_ACCESS is access CHAR_STRING;
type SCREEN_ARRAY is array(0..149) of STRING_ACCESS;
type STRING_POINTER is access SCREEN_ARRAY;
```

The final type declaration shown above (`STRING_POINTER`) is used as the type declaration for the `YBAR` field in the Ada WINDOW structure. When the Ada structure is then overlaid on the C window structure, it creates a parallel character storage structure by capturing the pointer stored in the C variable `_y` in the Ada `YBAR` field. A diagram of this structure is shown in Figure 5-4.

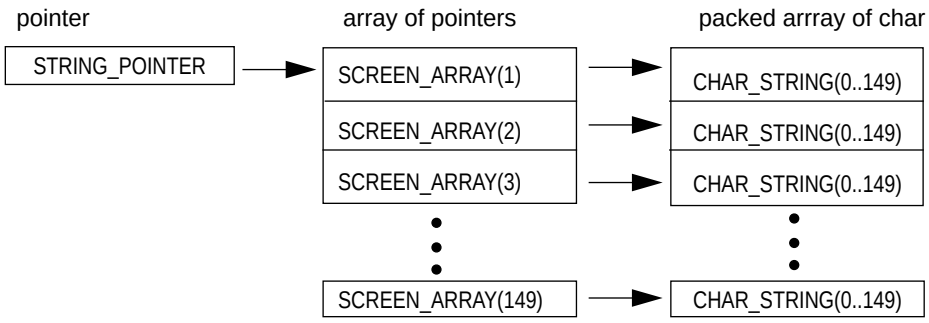

Figure 5-4 Parallel Character Storage Structure

Ada requires limits on arrays at compilation time, hence the 150 component restriction on the arrays declared. However, this does not affect the usefulness of our mapping as long as `SCREEN_ARRAY(0..ROWS-1)` and `CHAR_STRING(0..COLS-1)` are overlaid correctly on the corresponding C structures in memory. The call to `WINCH` becomes the following.

```
function WINCH( WIN: WINDOW ) return CHARACTER is
  T: TINY_INTEGER;
begin
  T := WIN.YBAR.ALL( INTEGER( WIN.CURY )).ALL( INTEGER(WIN.CURX ));
  T := ABS( T );
  return CHARACTER'VAL( T );
end WINCH;
```

Another interesting problem arises when implementing terminal characteristic macros. These operations are performed by masking bits contained in the short integer sg_flags of the C structure sgttyb given in the following code segment. This field is used as a packed array of boolean bit flags indicating the status of different terminal characteristics.

```
struct sgttyb {
  char  sg_ispeed;      /* input speed */
  char  sg_ospeed;      /* output speed */
  char  sg_erase;       /* erase character */
  char  sg_kill;        /* kill character */
  short sg_flags;       /* mode flags */
};
```

The operations on the sg_flags field are accomplished by means of bitwise ANDs and ORs setting or clearing flags as needed. The Ada language, in contrast, provides no provision for bitwise ANDs and ORs on integers. However, it does provide for such operations on arrays of booleans. The key to resolution of this problem is to define an array of booleans mapped to the integer storage defined by the C program. After a connection is established using pragma INTERFACE_NAME, use unchecked conversion to convert the SHORT_INTEGER field to a packed array of boolean. In this form, alter the fields by using the predefined Ada and or operations in conjunction with the appropriate mask values. After flags are altered, the process of conversion can be reversed. Using this technique, a procedure like echo becomes the following:

```
procedure ECHO is
begin
  TTY.SG_FLAGS := PACK( UNPACK( TTY.SG_FLAGS ) OR UNPACK( M_ECHO ));
  ECHOIT := TRUE;
  TTY_POINTER := TTY'ADDRESS;
  STTY( TTY_CH, TTY_POINTER );
end ECHO;
```

The functions PACK and UNPACK are renamed instantiations of the generic function UNCHECKED_CONVERSION. For portability sake, the functions PACK and UNPACK can be simulated by division on compiler systems not supporting bit level packing.

5.1.5 Reduce the Overhead

The final step in the interface process is to reduce the overhead resulting from frequent subprogram calls to intermediate routines written in Ada. In the C code, this is done with macros masquerading as functions and the effect is to place the code inline. The Ada solution is provided by the predefined `pragma INLINE` (section 6.3.2 in the *Ada Reference Manual*). When used with intermediate routines or routines that call others, this pragma causes the Ada compiler to treat the called Ada routine like a C macro, resulting in reduced stack manipulation and fewer subprogram calls during execution.

This pragma is best applied to Ada programs only after testing and debugging is done, because narrowing down errors during debugging is more difficult with inline code. (The pragmas can be placed appropriately in the code when constructing it and temporarily commented out until the debugging phase is complete.) Depending upon the length of the files being inlined, this pragma has the potential to greatly expand object file size. Exercise caution when disk space is at a premium.

5.2 Program Conversion

Moving projects into Ada can be a lengthy process since the code being replaced can be developed over many years. One modular approach is to write an Ada "wrapper" program that surrounds the subprograms in other languages and gradually replaces them. `pragma INTERFACE` and `pragma INTERFACE_NAME` are useful for this gradual replacement, but because of the interrelation of subprograms that call many other subprograms, it is not normally possible to replace the upper level subprogram without rewriting all its dependents in Ada as well.

SC Ada provides other pragmas, `EXTERNAL` and `EXTERNAL_NAME`, which make this replacement simpler. `pragma EXTERNAL` and `pragma EXTERNAL_NAME` enables subprograms in other languages to call Ada subprograms, exactly the reverse of the `INTERFACE` and `INTERFACE_NAME pragma`s.

To illustrate, imagine that Figure 5-5 is originally written in C:

```
/* C program example that calls an Ada subprogram with a
** parameter declared in C */
#include <stdio.h>
char *gets ();
int atoi ();
int service_number;
extern void ada_put ();      /* Ada: TEXT_IO.INTEGER_IO.PUT of an
                                INTEGER */
test ()
{
  char buf[80];
  printf (“Enter an integer here: ");
  gets (buf);
  service_number = atoi (buf);
  ada_put (service_number);   /* originally used printf */
  putchar (’\n’);
}
```

The `printf` call is replaced with ADA_PUT, an Ada package containing `procedure` ADA_PUT, and interface declarations for the C entities is written.

```
with LANGUAGE; use LANGUAGE;
with TEXT_IO; use TEXT_IO;
package C_INTERFACE is
 SERVICE_NUMBER : INTEGER;
 pragma INTERFACE_NAME (SERVICE_NUMBER, C_PREFIX &
“service_number”);
 procedure ADA_PUT (I : INTEGER);
 pragma EXTERNAL (C, ADA_PUT);
 pragma EXTERNAL_NAME (ADA_PUT, C_SUBP_PREFIX & “ada_put”);
 procedure MAIN;
 pragma INTERFACE (C, MAIN);
 pragma INTERFACE_NAME (MAIN, C_SUBP_PREFIX & “test”);
end C_INTERFACE;
package body C_INTERFACE is
 procedure ADA_PUT (I : INTEGER) is
   package I_IO is new INTEGER_IO (INTEGER);
   use I_IO;
 begin
   PUT (I);
 end ADA_PUT;
end C_INTERFACE;
```

Figure 5-5 Example of Program Conversion

Now a simple Ada “wrapper” to call the original C function `main` is written, so the linker `a.ld` can resolve all the references in the modules and perform its usual elaboration order checks.

```
with C_INTERFACE; use C_INTERFACE;
procedure MN is
begin
   MAIN;
end;
```

Compile both the C and Ada portions by:

- Compiling the C portion
- Using the SC Ada linker to construct the “main” program `MN` and including the C object in the link

The program runs correctly. Its Ada calling sequence is shown graphically in Figure 5-6

.

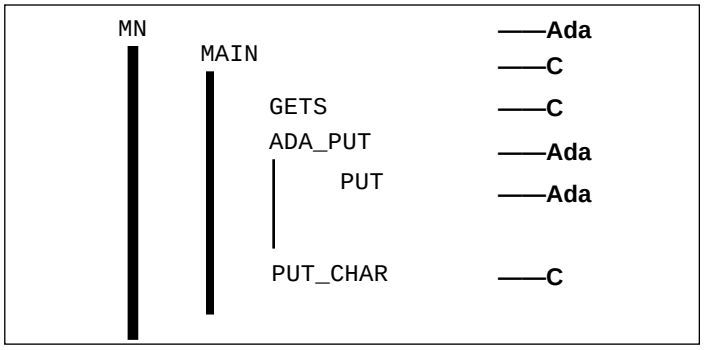

Figure 5-6 Ada Calling Sequence

The real benefit is that new portions of large programs are developed in Ada, but existing, tested, working code need not be replaced wholesale; individual modules or groups of modules can be replaced by newly developed Ada code without undue restrictions on the language of calling or called subprograms. An additional benefit is that once subprogram parameters are defined in Ada, the compiler performs its usual type checking across subprograms.

5.3 Calling Ada From Other Languages

Ada compilations generate objects. Link them with objects generated by other compilers, and call them from programs written in other languages. However, a number of complexities arise. This section discusses them, but we nonetheless recommend that main programs be Ada.

The safest approach to avoiding problems when calling Ada routines from other languages is to ensure that *all* Ada routines called are only in library level packages or be library level subprograms. This guarantees that all references (including those in the called routines) are necessarily correct by Ada visibility rules. Using locally defined subprograms is possible, but is strongly discouraged. The called routine does not have the addressing framework set up that it requires for up-level addressing. Only local and global variables in library level packages can be used correctly in a subprogram that is called from outside Ada and in any routine that is called by that routine.

5.3.1 `pragma EXTERNAL` *and* `pragma EXTERNAL_NAME`

SC Ada provides `pragma EXTERNAL` and `pragma EXTERNAL_NAME`, which enable you to write Ada subprograms that are callable from other languages.

`pragma EXTERNAL` enables Ada programs to generate subprograms callable by other languages.

```
pragma EXTERNAL (language, Ada_proc)
```

The context, allowed languages, and types of subprograms for this pragma are all the same as `pragma INTERFACE`, except that a body must be supplied for the subprogram. This pragma means the subprogram must be callable from the given language, that is, the calling conventions of language are used, and the stack limit register (if any) is restored from memory. If no `pragma EXTERNAL_NAME` is applied to the subprogram, a default external name is defined, using the same rules as for defining the default interface name for `pragma INTERFACE` subprograms. This pragma has an effect only when the calling conventions of the foreign language differ from those of Ada.

`pragma EXTERNAL_NAME` enables a specific link name to be given to an Ada subprogram so that it can, for example, be called from another language.

```
pragma EXTERNAL_NAME (Ada_proc, “ada_proc”)
```

In this example, the external link name is the string "ada_proc." The link name is formed using the same rules as for pragma INTERFACE_NAME.

References

Find pragme INTERFACE_NAME and pragma INTERFACE usage examples in examples directory.

5.3.2 Finding the Right Object

Ada hides the objects in the .objects directory. Often several objects generate during the compilation of a single source file. Each object for a file generates with the filename used as a prefix and two characters at the end, starting with 01, 02, and so forth. For this reason, we recommend compiling each unit in a separate source file. Use the Ada mapping tool, a.map, to determine which object contains the needed entry point.

Ada generates unusual compound names for entities such as subprograms. These names disambiguate overloading and assist in debugging. Usually, the names have the form:

_A_*subname*llXcc.*parent*

subname is the name of the sub-entity. ll is the line number of its definition. X is S if defined in the spec, and B if defined in the body. cc is the character number of its definition, and *parent* is the name of the parent unit, less its prefix of _A_.

Generated names contain line or character numbers and change if the source that contains them changes. Therefore, we recommend that you give explicit names to any entities that you reference from other languages.

References

"pragma EXTERNAL_NAME (subprogram, link_name), in Appendix F, "Implementation-Dependent Characteristics"

5.3.3 Avoiding Elaboration

This section applies to archives of Ada object files.

Ada semantics dictate that units not be used before they are elaborated. SC Ada generates dynamic elaboration checks for many subprograms, to verify this property. If you just link and call an Ada package, the `PROGRAM_ERROR` exception is almost certainly raised. Two approaches can prevent this.

One possibility is to do the elaboration. The simplest technique is to artificially create a main program that calls (directly or indirectly) all the entry points needed from the other language and link it with the verbose option, saving the output in a file. An object having the filename with an `o` appended is created, holding an `ELABORATION_TABLE`, among other things. To elaborate all units, the start-up procedure simply calls each address in that table in the given order — the table is an array of elaboration subprogram addresses.

Another possibility is to avoid elaboration. To do so, use `pragma SUPPRESS (ELABORATION_CHECK)` in every unit linked. Avoid "complex" initialization in units to be linked; complex initialization is any statically allocated object with an initial value that is neither an Ada static value, a value of system address, a record without a representation clause of these items, nor an array of these items.

Any complex initialization generates code in an elaboration procedure. If you expect not to do elaboration, these complex initializations would not get done. The disassembler can help detect and avoid complex initializations. Also, linking only library subprograms ensures that no elaboration is needed. SC Ada supports `pragma NOT_ELABORATED` to assist in creating units that do no elaboration.

References

elaboration of Library Units, section 10.5, and static expressions, section 4.9, in *Ada Reference Manual*

5.3.4 Linking a Non-Ada Main Program

To link, add the names of all appropriate SC Ada object files to the list of files given to the linker. If elaboration is intended, link the file described above under “Avoiding Elaboration,” using an artificial main program. Create a library or archive of these Ada objects for easier use.

5.3.5 Runtime Considerations

Each programming language has its own runtime system. Normally, only use one of these runtime systems in a given execution. That is the runtime initialized by the program start-up entry point (`_ _start` for Ada). The SC Ada linker creates Ada programs that call the SC Ada `_ _start` entry point, initializing the SC Ada Runtime System. That runtime “captures” the machine and, among other things, sets up signal (interrupt) handlers for all Ada-important signals. If the Ada runtime is *not* in control of the machine, do not use tasking, exceptions, exception handling or Ada I/O. Ada I/O requires elaboration.

5

"You play with my world
Like its your little toy."

Bob Dylan

User Library Configuration A

Configuring the user library is optional. Under most circumstances, SC Ada functions properly with no changes to the default configuration. However, if you want to reconfigure the library, the information contained in this appendix enables you to do so.

The first part of this appendix is a set of steps for modifying the configurable components of the user library. That is followed by a discussion of the configurable components.

Caution – Any packages or subprograms written to support configuring SC Ada should be contained within the body of the V_USR_CONF package. The name of the configuration package must remain as V_USR_CONF. Units must not be added to the context clause of this package unless the unit's object file is explicitly included into all the links via a LINK directive or the unit is included in the context clause of a unit which is always referenced by the main programs using this configuration.

A.1 Steps to Configure the User Library

This section provides instructions for configuring the user library on a self-host SC Ada development system. For the purposes of this example, assume that SC Ada is installed in a directory called `/usr2/ada_2.1`.

A.1.1 Build the User Library

The first step is to create an Ada library directory for configuring the user library. See Figure A-1.

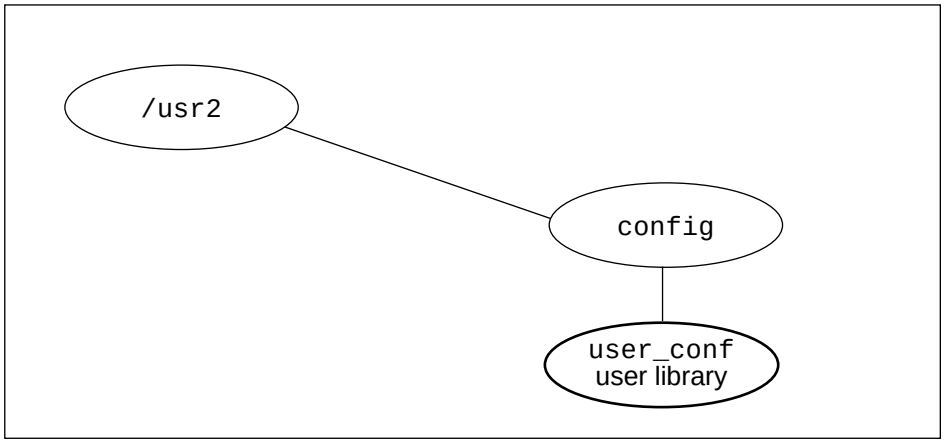

Figure A-1 Directory Structure for Configuring Ada

A.1.2 Create an Ada Library

Create the directory `config` under the existing `/usr2` directory. Enter:

```
mkdir /usr2/config
```

Move to the library that you just created, by entering:

```
cd /usr2/config
```

Use `a.mklib` to create a local directory called `user_conf` and make that directory into an Ada library. We recommend using the `-i` (interactive) option.

```
a.mklib -i user_conf
  1 version of standard is available on this machine:
      Target Name    Version   SC Ada Location
   1  SELF_TARGET    2.1       /usr2/ada_2.1/self
                          host and os name, SPARCompiler Ada version
   1  SELF_TARGET    2.1       /usr2/ada_2.1/self_thr
                          host and os name, SPARCompiler Ada version

   Selection (q to quit):
```

Note – `a.mklib` displays all versions of SC Ada on the host and prompts for the which version to link this SC Ada library. All SC Ada tools check the Ada library structure, so if multiple versions of SC Ada are on the same host, the desired compiler is used. For the VADS Threaded runtime, select 1; for Solaris MT runtime, select 2.

`a.mklib` creates the specified directory, makes it into an SC Ada library, and adds the SC Ada libraries `verdixlib` and `standard` to the new library search list.

A.1.3 Copy the Configuration Files

Move to the `/usr2/config/user_config` library that you just created, by entering:

```
cd user_config
```

Copy the source files in `/usr2/ada_2.1/self/user_conf` (or `usr2/ada_2.1/self_thr/user_conf`) to the new library directory:

For the VADS Threaded Runtime:

```
cp /usr2/ada_2.1/self/user_conf/v_usr_conf*.a  .
```

For Solaris MT Ada:

```
cp /usr2/ada_2.1/self_thr/user_conf/v_usr_conf*.a  .
```

This command copies these files:

```
v_usr_conf.a
v_usr_conf_b.a
```

The only file that may require changes is `v_usr_conf_b.a`.

A.1.4 Edit the User Configuration Package

Most applications require no modifications to the configuration package; the default parameters are sufficient in most instances. If you must modify the configuration package, change the body of the user library configuration package in the file `v_usr_conf_b.a`.

A.1.5 Compile All Ada Files

The following `a.make` command compiles all Ada files from the `usr_conf` directory in the correct order. The output appears below the command.

```
a.make -v -f *.a

finding dependents of: v_usr_conf.a
finding dependents of: v_usr_conf_b.a
finding dependents of: v_usr_conf_i.a
compiling v_usr_conf.a
   spec of v_usr_conf
compiling v_usr_conf_b.a
   body of v_usr_conf
```

A.1.6 Change the `ada.lib` *File*

`a.info` modifies `ada.lib`. We recommend that you use `a.info` in interactive mode, that is, invoked with `a.info -i`, but in our examples we show it used in command mode. Your local user configuration overrides the default configuration at link time.

The following `a.info` command line adds the body of the user configuration file to `ada.lib`:

```
a.info -a WITH1 /usr2/config/user_conf/.objects/v_usr_conf_b01
```

A.1.7 Build a Test Library

Now write an Ada program to test the configuration. For this example, assume you are creating a test directory in `/usr2/config`. Enter these commands:

```
cd /usr2/config
a.mklib test user_conf
cd test
```

`a.mklib` creates the directory named `test` automatically, under the directory `/usr2/config`, converts `test` to an Ada library, and places `/usr2/config/user_conf`, `/usr2/ada_2.1/self/standard`, and `/usr2/ada_2.1/self/verdixlib` on the library search list for `test`.

A.1.8 Edit, Compile, and Link Your Test Program

Use a simple program that prints “Hello, world!” to the terminal as a test program. Here is the source code that must be edited into a source file:

```
with text_io;
procedure hello is
begin
   text_io.put_line(“Hello, world!”);
end;
```

It is assumed that you put this source code into the file `test/hello.a`. By naming the main procedure the same as the file that contains it, compile and link in one step by using the `-M` option:

```
ada -v -M hello.a -o hello.out
```

Alternatively, you can compile and link in two steps:

```
ada -v hello.a
a.ld hello -o hello.out
```

Use `-v` to verify the new `.v_user_conf.b01` is being used.

A.1.9 Run Your Test Program

With the program compiled and linked successfully, run the executable file. Enter:

```
hello.out
```

The executable file displays the output:

```
Hello, world!
```

A.2 User Library Configuration Files

The following files are in the directory `usr_conf`. Copy these files to your local directory if you want to override the default configuration. You do not modify all of the files that you copy. The configuration package specification is intended to be read-only. It is present here so you can refer to it conveniently. The interface package specification `v_usr_conf_i`.a is now located in the `standard` library.

`v_usr_conf.a`	This package specification defines and describes the components that you must provide to configure the SC Ada self-host runtime environment for a user application program. Do not change this file.
`v_usr_conf_b.a`	This is the configuration package for the user library. Changing the body of this package configures the user library for your host.

A.3 V_USR_CONF *Configuration Components*

The configurable components of the user library configuration package are:

```
#c1: FAT_MALLOC's SMALL_BLOCK_SIZES_TABLE structure

#c2: Memory Allocation Parameters specific to
SLIM_MALLOC,
        FAT_MALLOC and DBG_MALLOC

#c3: CONFIGURATION_TABLE structure

#c4: V_GET_HEAP_MEMORY routine

#c5: V_PASSIVE_ISR routine

#c6: V_SIGNAL_ISR routine

#c7: V_CIFO_ISR routine

#c8: V_PENDING_OVERFLOW_CALLOUT routine

#c9: V_KRN_ALLOC_CALLOUT routine

#c10:V_START_PROGRAM and V_START_PROGRAM_CONTINUE
routines
```

Each of these components is discussed in detail. To find any of these components in the source files, `v_usr_conf.a` and `v_usr_conf_b.a`, search for the string, `#c`*N*, where *N* is the number of the desired component. In the source file, the code for each component begins with a line similar to the following:

```
    ---------------- #c3: CONFIGURATION_TABLE structure -----------
-
```

A.3.1 #c1: FAT_MALLOC SMALL_BLOCK_SIZES_TABLE *Structure*

The constant, SMALL_BLOCK_SIZES_TABLE, is declared in the body of this package. This information is used only if the FAT_MALLOC or DBG_MALLOC memory allocation archive is selected. This aggregate lists the small block sizes. If SMALL_BLOCK_SIZES_TABLE is initialized to (8, 16, 128), three small block lists are created that hold objects of 8 bytes, 16 bytes, and 128 bytes respectively. Allocation sizes between these values yield an object of the larger small block size. For example, user objects of size 20 are allocated as 128 byte small blocks.

```
------------ from:  v_usr_conf_i.a

subtype alloc_t is v_i_types.alloc_t;

type small_block_sizes_t is array(positive range <>) of alloc_t;

------------ from:  v_usr_conf_b.a

small_block_sizes_table :
  constant v_usr_conf_i.small_block_sizes_t :=( 8, 16, 24, 32 );
```

The block sizes must be declared in ascending order and each block size must be a multiple of 8.

Assign the address of SMALL_BLOCK_SIZES_TABLE to the parameter SMALL_BLOCK_SIZES_ADDRESS in the configuration table.

References

DBG_MALLOC and FAT_MALLOC, *SPARCompiler Ada Runtime System Guide*

A.3.2 #c2: *Memory Allocation Parameters specific to* SLIM_MALLOC, FAT MALLOC *and* DBG_MALLOC

The memory allocation configuration table contains the memory allocation parameters specific to *SLIM_MALLOC*, *FAT_MALLOC*, and *DBG_MALLOC*.

```
------------ from: v_usr_conf_i.a
  type allocation_strategy_t is (FIRST_FIT, BEST_FIT)
      min_size                 : alloc_t;
      num_small_block_sizes    : integer;
      small_block_sizes_address : address;
      min_list_length          : integer;
      intr_obj_size            : alloc_t;
      initial_intr_heap        : integer;
      allocation_strategy      : allocation_strategy_t
  end record;

------------ from: v_uxr_conf_b.a

mem_alloc_conf_table: constant v_usr_conf_i.mem_alloc_conf_table_t
      := (
MIN_SIZE                  =>  8,
NUM_SMALL_BLOCK_SIZES     =>  small_block_sizes_table'length,
SMALL_BLOCK_SIZES_ADDRESS =>  small_block_sizes_table'address,
MIN_LIST_LENGTH           =>  5,
INTR_OBJ_SIZE             =>  128,
INITIAL_INTR_HEAP         =>  100
ALLOCATION_STRATEGY       =>  FIRST_FIT
     );
```

A.3.2.1 MIN_SIZE

MIN_SIZE defines the minimum-size object to allocated. It determines the size at which an over-large space is broken into a perfect fit and a new free storage block. For example, you ask for 1000 bytes and the next free slot has 1500 bytes. If MIN_SIZE < 500, you receive exactly 1000 bytes of the free space and the remainder (500 bytes less header overhead) is put on the free list. If MIN_SIZE > 500, you receive 1500 bytes. This value should never be larger than the smallest small block. This controls fragment size.

A.3.2.2 NUM_SMALL_BLOCK_SIZES

NUM_SMALL_BLOCK_SIZES gives the number of small object sizes to be handled by the allocator. This is the number of elements in the SMALL_BLOCK_SIZES_TABLE of #c1.

A.3.2.3 SMALL_BLOCK_SIZES_ADDRESS

SMALL_BLOCK_SIZES_ADDRESS indicates the starting address of the small block sizes table defined in the previous section on #c1.

A.3.2.4 MIN_LIST_LENGTH

MIN_LIST_LENGTH specifies the minimum list length of a small blocks list. This keeps the allocator from coalescing blocks off of a list. In addition, when deallocating a block, the allocator decides whether it goes on the small blocks lists or is coalesced with its neighbors and put on the regular free list. The length is often shorter when little or no deallocation is being done.

A.3.2.5 INTR_OBJ_SIZE

INTR_OBJ_SIZE specifies the fixed size of blocks to allocate when a "new" is performed from an interrupt handler. This value is not adjustable at runtime.

A.3.2.6 INITIAL_INTR_HEAP

INITIAL_INTR_HEAP specifies the number of blocks to preallocate for allocation from an interrupt handler. The blocks are pre-allocated when AA_INIT is called. Allocate more blocks by making a call to v_i_alloc.extend_intr_heap, so set this number to 0 if desired.

A.3.2.7 ALLOCATION_STRATEGY

ALLOCATION_STRATEGY specifies the strategy used by AA_GLOBAL_NEW (and AA_ALIGNED_NEW) to choose a block from the free list. Generally, setting it to FIRST_FIT optimizes for speed while setting it to BEST_FIT optimizes for low fragmentation.

A.3.3 #c3: CONFIGURATION_TABLE *Structure*

The configuration table is the mechanism for communicating data parameters from the configuration file to the user library routines using these parameters. This constant, CONFIGURATION_TABLE, must be modified to describe the user program environment to Ada. This record passes to V_BOOT during user program start-up to control its initialization. The declaration of this record type is in package V_USR_CONF_I.A.

```
 ------------ from:  v_usr_conf_i.a
subtype floating_point_control_t is v_i_types.floating_point_control_t;

type configuration_table_t is record

        main_task_stack_size                    : integer;
        default_task_stack_size                 : integer;
        exception_stack_size                    : natural;
        idle_stack_size                         : natural;
        signal_task_stack_size                  : natural;
        fast_rendezvous_enabled                 : boolean;
        wait_stack_size                         : natural;
        floating_point_support                  : integer;
       floating_point_control                : floating_point_control_t;
        heap_max                                : alloc_t;
        heap_extend                             : alloc_t;
        get_heap_memory_callout                 : address;
        mem_alloc_conf_table_address            : address;
        disable_signals_mask                    : integer;
        disable_signals_33_64_mask              : integer;
        numeric_signal_enabled                  : boolean;
        storage_signal_enabled                  : boolean;
        time_slicing_enabled                    : boolean;
                                                -- VADS Threaded Ada only
        time_slice_interval                       : duration;
                                                -- VADS Threaded Ada only
        time_slice_priority                       : priority;
                                                -- VADS Threaded Ada only
        concurrency_level                         : natural;
                                                  -- Solaris MT Ada only
 enable_signals_mask                     : integer;
                                                  -- Solaris MT Ada only
        enable_signals_33_64_mask                 : integer;
                                                  -- Solaris MT Ada only
```

```
        exit_signals_mask                       : integer;
                                                 -- Solaris MT Ada only
        exit_signals_33_64_mask                 : integer;
                                                 -- Solaris MT Ada only
        intr_task_prio                          : priority;
                                                 -- Solaris MT Ada only
        intr_task_stack_size                    : natural;
                                                 -- Solaris MT Ada only
       default_task_attributes              : ada_krn_defs.task_attr_t;
        main_task_attr_address                  : address;
        signal_task_attr_address                : address;
        intr_task_attr_address                  : address;
                                                 -- Solaris MT Ada only
        masters_mutex_attr_address              : address;
        mem_alloc_mutex_attr_address            : address;
        ada_io_mutex_attr_address               : address;
        old_style_max_intr_entry                : address;

        task_storage_size                       : integer;
                                                -- VADS Threaded Ada only
        pending_count                           : integer;
                                                -- VADS Threaded Ada only
end record;
------------ from:  v_usr_conf_b.a

  -- If you don't want to use the default mutex or condition variable
  -- attributes, change the V_INIT_ATTR routine to initialize
  -- these two attribute records and change the mutex_attr_address
  -- or cond_attr_address parameter values in the configuration_table
  -- from NO_ADDR to mutex_attr'address or cond_attr'address.
  --
  mutex_attr :      ada_krn_defs.mutex_attr_t;
  cond_attr :      ada_krn_defs.cond_attr_t;
  configuration_table : constant v_usr_conf_i.configuration_table_t := (

-- Parameter                  Value
------------                  ---------
    --  #c3a: STACK CONFIGURATION PARAMETERS:
MAIN_TASK_STACK_SIZE     =>   16#20_0000#,
DEFAULT_TASK_STACK_SIZE  =>   10_240,
EXCEPTION_STACK_SIZE     =>   5000,
IDLE_STACK_SIZE          =>   10_240,
SIGNAL_TASK_STACK_SIZE   =>   10_240,
FAST_RENDEZVOUS_ENABLED  =>   TRUE,
```

```
     --  #c3b: FLOATING-POINT COPROCESSOR CONFIGURATION PARAMETERS:
FLOATING_POINT_SUPPORT   =>   v_ada_info.FP_NATIVE,

FLOATING_POINT_CONTROL   =>   (
        rd   => v_i_types.to_nearest,
        rp   => v_i_types.extended,
       tem  => (invalid | overflow | zero_div => true, others => false),
        au   => false,
        ftt  => v_i_types.none,
        qne  => false,
        fcc  => v_i_types.equal,
        aexc => (others => false),
        cexc => (others => false)
    ),

    --  #c3c: HEAP MEMORY CALLOUT CONFIGURATION PARAMETERS:

HEAP_MAX                 =>  32*1024*1024,

HEAP_EXTEND              =>   128*1024,
GET_HEAP_MEMORY_CALLOUT  =>   v_get_heap_memory'address,

    --  #c3d: MEMORY ALLOCATION CONFIGURATION TABLE:
MEM_ALLOC_CONF_TABLE_ADDRESS
                         =>   mem_alloc_conf_table'address,


    --  #c3e: UNIX SIGNAL CONFIGURATION PARAMETERS:
DISABLE_SIGNALS_MASK     =>   ada_krn_defs.DISABLE_MASK,
NUMERIC_SIGNAL_ENABLED   =>   TRUE,
STORAGE_SIGNAL_ENABLED   =>   TRUE,

   -- #c3f: TIME SLICE CONFIGURATION PARAMETERS: --VADS Threaded Ada only

  TIME_SLICING_ENABLED    =>   FALSE,
  TIME_SLICE_INTERVAL     =>   1.00,
  TIME_SLICE_PRIORITY
```

```
   --  #c3f: Solaris MT
 Ada CONFIGURATION PARAMETERS:  -- Solaris MT Ada only
CONCURRENCY_LEVEL            => 1,
ENABLE_SIGNALS_MASK          => ada_krn_defs.ENABLE_MASK,
ENABLE_SIGNALS_33_64_MASK    => ada_krn_defs.ENABLE_33_64_MASK,
EXIT_SIGNALS_MASK            => ada_krn_defs.EXIT_MASK,
EXIT_SIGNALS_33_64_MASK      => ada_krn_defs.EXIT_33_64_MASK,
INTR_TASK_PRIO               => priority'last,
INTR_TASK_STACK_SIZE         => 10_240,

      --  #c3g: ATTRIBUTES CONFIGURATION PARAMETERS:
  DEFAULT_TASK_ATTRIBUTES =>   (
                prio             => priority'first,

                flags            => 0,            -- Solaris MT Ada only

               sporadic_attr_address => NO_ADDR, -- not a sporadic task
                                                  --VADS Threaded Ada only
            mutex_attr_address     =>  NO_ADDR,-- use default mutex attr
  --    mutex_attr_address        =>  mutex_attr'address,
             cond_attr_address =>     NO_ADDR),-- use default cond attr
  --    cond_attr_address         =>      cond_attr'address),
  MAIN_TASK_ATTR_ADDRESS      =>  NO_ADDR,-- use DEFAULT_TASK_ATTRIBUTES
  SIGNAL_TASK_ATTR_ADDRESS    =>  NO_ADDR,-- use DEFAULT_TASK_ATTRIBUTES
  INTR_TASK_ATTR_ADDRESS      =>  NO_ADDR,-- use DEFAULT_TASK_ATTRIBUTES
                                          -- Solaris MT Ada only
  MASTERS_MUTEX_ATTR_ADDRESS
                         =>  NO_ADDR,          -- use default mutex attr
  --                      =>  mutex_attr'address,
  MEM_ALLOC_MUTEX_ATTR_ADDRESS
                         =>  NO_ADDR,          -- use default mutex attr
  --                      =>  mutex_attr'address,
  ADA_IO_MUTEX_ATTR_ADDRESS
                         =>  NO_ADDR,          -- use default mutex attr
 --                       =>  mutex_attr'address,

--  #c3h: MISC CONFIGURATION PARAMETERS:
OLD_STYLE_MAX_INTR_ENTRY     => memory_address(16#1FF#),  -- 512 - 1
TASK_STORAGE_SIZE            => 32,        -- VADS Threaded Ada only
PENDING_COUNT                => 20         -- VADS Threaded Ada only
);
```

A.3.4 `#c3a:` *Stack Configuration Parameters*

The following parameters specify different user configurable stack sizes.

A.3.4.1 MAIN_TASK_STACK_SIZE

MAIN_TASK_STACK_SIZE is the size of the user program stack. The stack size includes room at the bottom for exception handling (1k bytes).

A.3.4.2 DEFAULT_TASK_STACK_SIZE

DEFAULT_TASK_STACK_SIZE is the size of each task stack area. Override this value with a T'STORAGE_SIZE length clause.

A.3.4.3 EXCEPTION_STACK_SIZE

EXCEPTION_STACK_SIZE is the space set aside below the bottom of the task stack for exception unwinding.

A.3.4.4 IDLE_STACK_SIZE

IDLE_STACK_SIZE is the size of the stack for the idle task.

A.3.4.5 SIGNAL_TASK_STACK_SIZE

SIGNAL_TASK_STACK_SIZE is the size of the stack for tasks created for doing rendezvous with interrupt entries.

A.3.4.6 FAST_RENDEZVOUS_ENABLED

When FAST_RENDEZVOUS_ENABLED is set to TRUE, if the acceptor task is already waiting, the rendezvous is executed in the context of the caller task. This value can be overridden on a per task basis via the VADS EXEC service V_XTASKING.SET_FAST_RENDEZVOUS_ENABLED.

Caution is recommended when this option is enabled in multithreaded (LWPs) environments. Acceptor tasks can run on LWPs or processors which require special system resources. In this case, rendezvous code cannot be executed by a task running on a different LWP or processor.

References

"WAIT_STACK_SIZE" on page A-16" SET_FAST_RENDEZVOUS_ENABLED,

A.3.4.7 WAIT_STACK_SIZE

For a fast rendezvous, the acceptor task saves its register context, switches to a wait stack and waits. Eventually the caller task restores and uses the acceptor's saved register context. WAIT_STACK_SIZE specifiers how big a stack is needed for when the acceptor switches from its normal task stack to a special stack it can use to call a kernel service to block itself. Note that if FAST_RENDEZVOUS_ENABLED is set to FALSE, WAIT_STACK_SIZE is not used. Setting WAIT_STACK_SIZE to zero also disables the fast rendezvous optimization.

A.3.5 #c3b: FLOATING_POINT *Configuration Parameters*

A.3.5.1 FLOATING_POINT_SUPPORT

If any task executes a floating point instruction, FLOATING_POINT_SUPPORT should be set to V_ADA_INFO.FP_NATIVE. Otherwise, to reduce task context switch times, it should be set to V_ADA_INFO.FP_SOFTWARE. Currently, only FP_NATIVE is supported.

A.3.5.2 FLOATING_POINT_CONTROL

FLOATING_POINT_CONTROL specifies the initial value for the FSR register of the FPU. This structure and its subcomponents are declared in package V_I_TYPES in the standard directory of the release.

The field of the components and their initial values are specified in package V_USR_CONF in the usr_conf directory of the release.

The default value for this structure is as follows: zero-divide and overflow exceptions are unmasked, the running precision is “to nearest,” and the running mode is “extended.”

A.3.6 #c3c: Heap Memory Callout Configuration Parameters

These parameters are used by the default memory allocation strategy. When the user program executes an allocator, the allocated object uses up memory from the heap.

A.3.6.1 HEAP_MAX

HEAP_MAX sets the maximum size of the heap. This value is used to limit the program's heap area by calling the Solaris services getrlimit/setrlimit. Currently, it can be increased to 512MB.

```
HEAP_MAX    =>  64 * 1024 * 1024,
```

If the current stack limit for the process is less than the MAIN_TASK_STACK_SIZE + EXCEPTION_STACK_SIZE, `setrlimit()` is called to extend it.

A.3.6.2 HEAP_EXTEND

HEAP_EXTEND defines the minimum number of storage units requested from GET_HEAP_MEMORY_CALLOUT if the allocator's memory is exhausted.

References

"#c4: V_GET_HEAP_MEMORY Routine" on page A-27

A.3.6.3 GET_HEAP_MEMORY_CALLOUT

GET_HEAP_MEMORY_CALLOUT specifies the address of the routine to be called when more memory is needed for new allocations.

The default value for GET_HEAP_MEMORY_CALLOUT is V_GET_HEAP_MEMORY'address.

A.3.7 #c3d: Memory Allocation Configuration Table

A.3.7.1 MEM_ALLOC_CONF_TABLE_ADDRESS

MEM_ALLOC_CONF_TABLE_ADDRESS points to the memory allocation table containing parameters specific to the particular allocation routines being used. If you write your own allocation routines, you can point this address to your own configuration table.

References

allocator, *Ada Reference Manual*, section 4.8

A.3.8 #c3e: *Solaris Signal Configuration Parameters*

The parameters in this section control signals from the operating system.

A.3.8.1 DISABLE_SIGNALS_MASK

DISABLE_SIGNALS_MASK is the mask used in the kernel when signals 1..32 are disabled. The kernel does not support nested-asynchronous signals. The default value is ADA_KRN_DEFS.DISABLE_MASK. Do not enable additional signals.

A.3.8.2 DISABLE_SIGNALS_33_64_MASK

DISABLE_SIGNALS_33_64_MASK is the mask used in the kernel when signals 33..64 are disabled. The default value is ADA_KRN_DEFS.DISABLE_33_64_MASK. Do not enable additional signals.

A.3.8.3 NUMERIC_SIGNAL_ENABLED

If NUMERIC_SIGNAL_ENABLED is TRUE, the Ada program uses the Solaris 2.1 SIGFPE signal to catch numeric errors. If FALSE, numeric errors are not caught, according to the *Ada Reference Manual*, and the Ada program does not affect SIGFPE. This is useful when calling foreign packages that use or ignore SIGFPE.

References

numeric errors, section 11.1, in *Ada Reference Manual*

A.3.8.4 STORAGE_SIGNAL_ENABLED

If STORAGE_SIGNAL_ENABLED is TRUE, the Ada program uses the Solaris 2.1 SIGSEGV signal to catch storage errors. If it is FALSE, numeric errors are not caught, according to the *Ada Reference Manual*, and the Ada program does not affect SIGSEGV. This is useful when calling foreign packages that use or ignore SIGSEGV.

References

storage errors, section 11.1, in *Ada Reference Manual*

A.3.9 #c3f: Time Slice Configuration Parameters - VADS Threaded Area

Time slice configuration parameters turn time slicing on or off, set the time slice interval, and set the time slicing priority.

Caution – Time slice configuration parameters are only valid using VADS MICRO.

A.3.9.1 TIME_SLICING_ENABLED

`TIME_SLICING_ENABLED` is set to `TRUE` to enable time slicing and to `FALSE` to prevent time slicing.

A.3.9.2 TIME_SLICE_INTERVAL

Each task in the system is started with `TIME_SLICE_INTERVAL` time in its time slice. This is the amount of time it runs before it is preempted to see if any equal priority tasks are ready to run.

The default time slice interval is 1.0 second.

A.3.9.3 TIME_SLICE_PRIORITY

`TIME_SLICE_PRIORITY` specifies the maximum task priority to which time slicing applies. This way, normal tasks can be time-sliced, but high priority tasks execute until they give up the processor (or until an even higher priority task becomes ready).

All tasks whose priority is less than or equal to `TIME_SLICING_PRIORITY` are time-sliced.

A.3.10 #c3f: Solaris MT Ada Configuration Parameters - MT Ada

Caution – The Solaris MT Ada configuration parameters are valid only if using Solaris Threads.

A.3.10.1 CONCURRENCY_LEVEL

CONCURRENCY_LEVEL suggests how many LWPs should be in the multiplexing pool.

A.3.10.2 ENABLE_SIGNALS_MASK *and* ENABLE_SIGNALS_33_64_MASK

These masks are used during startup to initialize the main task's signal mask. All subsequently created threads/tasks inherit this signal mask.

A separate Ada interrupt task is created for each attached signal handler. This task does a `sigwait()` for the attached signal. Therefore, a handler may be attached only for a signal disabled by these enable masks.

By default `ENABLE_SIGNALS_MASK = 16#FEBF_B007#` and `ENABLE_SIGNALS_33_64_MASK = 0`. The default disables all signals except:

```
4 - SIGILL (mapped to an Ada exception)
5 - SIGTRAP
6 - SIGIOT
7 - SIGEMT
8 - SIGFPE (mapped to an Ada exception)
9 - SIGKILL
10 - SIGBUS
11 - SIGSEGV (mapped to an Ada exception)
12 - SIGSYS,
15 - SIGTERM (used by Ada RTS for thread termination)
23 - SIGSTOP
25 - SIGCONT
33 - SIGLWP (used by Solaris Threads)
```

Effectively, only the synchronous exception related signals are enabled.

Since we use `sigwait` to handle asynchronous signals, an asynchronous signal must never be enabled in any thread.

A.3.10.3 EXIT_SIGNALS_MASK *and* EXIT_SIGNALS_33_64_MASK

Since we normally disable all asynchronous signals, we have another set of signal masks that indicate which signals should cause the program to exit if a handler hasn't been attached. An exit signals thread is created that waits for one of these signals. However, attached signals are automatically removed from the exit signal mask.

Exit signals not included in the enable signals masks are ignored.

By default `EXIT_SIGNALS_MASK = 16#0001_8006#` and `EXIT_SIGNALS_33_64_MASK = 0`. The default causes the program to exit if it receives one of the following signals without an attached handler:

```
2 - SIGINT
3 - SIGQUIT
16 - SIGUSR1
17 - SIGUSR2
```

A.3.10.4 INTR_TASK_PRIO

`INTR_TASK_PRIO` is the priority of all the interrupt tasks doing a sigwait() for an attached signal. The default is `PRIORITY'LAST`.

A.3.10.5 INTR_TASK_STACK_SIZE

`INTR_TASK_STACK_SIZE` is the size of the stack for all interrupt tasks. The default is the same as `DEFAULT_TASK_STACK_SIZE`.

A.3.11 #c3g: Attributes Configuration Parameters

When an Ada task is created, attributes for initializing the task are passed to the underlying microkernel. The attributes configuration parameters contain the default task attributes and the attributes.for the main, signal and interrupt tasks.

The Ada tasking, memory allocation and I/O routines use mutexes to protect their data structures. When an Ada task needs to block it waits on a condition variable. The attributes configuration parameters are used to initialize the

mutex and condition variable objects implicitly created by the Ada RTS routines. See Ada Kernel, for more details about a mutex or condition variable object.

In general you should be able to use the default mutex and condition variable attributes. Here's what you get if you stick with the default.

For single processor Ada: a mutex initialized using the default mutex attributes, locks the mutex by executing a test-and-set instruction and does FIFO waiting when the mutex is locked by another task. In the CIFO add-on product, the default changes to priority inheritance waiting when pragma `SET_PRIORITY_INHERITANCE_CRITERIA` appears in the main procedure.

For Solaris MT Ada: a mutex initialized using the default mutex attributes, locks the mutex by executing a test-and-set instruction and does priority waiting when the mutex is locked by another task.

For single processor Ada: a condition variable initialized using the default condition variable attributes, does FIFO waiting. In the CIFO add-on product, the default changes to priority waiting when pragma `SET_PRIORITY_INHERITANCE_CRITERIA` appears in the main procedure.

For Solaris MT Ada: a condition variable initialized using the default mutex attributes does priority waiting.

Now, a few words about each of the attributes configuration parameters.

A.3.11.1 DEFAULT_TASK_ATTRIBUTES

`DEFAULT_TASK_ATTRIBUTES` specifies the default task attributes to be passed to the underlying microkernel at the creation of an Ada task.

The microkernel dependent task attributes record, `TASK_ATTR_T`, is defined in the package `ada_krn_defs.a`, located in the `standard` Ada library. Each field in the `TASK_ATTR_T` record is introduced below:

The `prio` field in the default task attributes isn't used. The default task priority is 0. This can be overridden by using either pragma `PRIORITY()` or pragma `TASK_ATTRIBUTES()` on a per task or task type basis.

For single processor Ada: in the CIFO add-on product, the `sporadic_attr_address` field can be updated with the address of an `ADA_KRN_DEFS.SPORADIC_ATTR_T` record to default all Ada tasks to being

sporadic. However, the main task can't be sporadic. Unless you are doing something weird, this field should remain as NO_ADDR. Why would you want to default all tasks to being sporadic?

For Solaris MT Ada: the flags field contains the value for the "flags" argument passed to the Solaris Threads `THR_CREATE()` service. Two flags attributes might be set: `OS_THREAD.THR_BOUND` and/or `OS_THREAD.THR_NEW_LWP`. All threads created for Ada tasks are started with `THR_SUSPENDED` and `THR_DETACHED` set.

The `mutex_attr_address` field contains the address of the default mutex attributes to be used to initialize the mutex object implicitly created for each task. This mutex is used to protect the task's data structure. For example, the task's mutex is locked when another task attempts to rendezvous with it.

`mutex_attr_address` should be set to `NO_ADDR` to use the default mutex attributes. Otherwise, it should be set to the address of an `ADA_KRN_DEFS.MUTEX_ATTR_T` record initialized in `V_USR_CONF.V_INIT_ATTR`().

If the `mutex_attr_address` field is set to `NO_ADDR` in the `TASK_ATTR_T` record referenced in pragma `TASK_ATTRIBUTES()`, the `DEFAULT_TASK_ATTRIBUTE`'s `mutex_attr_address` is used.

The `cond_attr_address` field contains the address of the default condition variables attributes to be used to initialize the condition variable object implicitly created for each task. When the task blocks, it waits on this condition variable.

`cond_attr_address` should be set to `NO_ADDR` to use the default condition variable attributes. Otherwise, it should be set to the address of an `ADA_KRN_DEFS.COND_ATTR_T` record initialized in `V_USR_CONF.V_INIT_ATTR()`.

If the `cond_attr_address` field is set to `NO_ADDR` in the `TASK_ATTR_T` record referenced in pragma `TASK_ATTRIBUTES()`, the `DEFAULT_TASK_ATTRIBUTE`'s `cond_attr_address` is used.

A.3.11.2 MAIN_TASK_ATTR_ADDRESS

MAIN_TASK_ATTR_ADDRESS points to the task attributes to be used for the main task. Set MAIN_TASK_ATTR_ADDRESS to NO_ADDR to use the above default task attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.TASK_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

The prio field in the main task attributes isn't used. The default priority for the main task is 0. This can be overridden by using pragma PRIORITY() in the main procedure.

For single processor Ada: in the CIFO add-on product, the sporadic_attr_address field in the main task attributes isn't used. The main task can't be sporadic.

A.3.11.3 SIGNAL_TASK_ATTR_ADDRESSS

SIGNAL_TASK_ATTR_ADDRESS points to the task attributes to be used for the tasks created to rendezvous with interrupt entries. Set SIGNAL_TASK_ATTR_ADDRESS to NO_ADDR to use the above default task attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.TASK_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

The prio field in the SIGNAL_TASK attributes is not used. The priority of a SIGNAL_TASK defaults to the priority of the attached task containing the interrupt entry. This priority is overridden by using the new style of an interrupt entry which contains the address of an ADA_KRN_DEFS.INTR_ENTRY_T record. The INTR_ENTRY_T record has the prio field.

A.3.11.4 INTR_TASK_ATTR_ADDRESSS

For Solaris MT Ada: INTR_TASK_ATTR_ADDRESS points to the task attributes to be used for the interrupt tasks created to do an OS_SIGNAL sigwait for attached signals. Set INTR_TASK_ATTR_ADDRESS to NO_ADDR to use the above default task attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.TASK_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

The prio field in the signal task attributes is not used. Instead, the configuration table's INTR_TASK_PRIO is used.

A.3.11.5 MASTERS_MUTEX_ATTR_ADDRESSS

MASTERS_MUTEX_ATTR_ADDRESS points to the mutex attributes to be used to initialize the Ada kernel's master mutex. Set MASTERS_MUTEX_ATTR_ADDRESS to NO_ADDR to use the default mutex attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.MUTEX_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

A.3.11.6 MEM_ALLOC_MUTEX_ATTR_ADDRESSS

MEM_ALLOC_MUTEX_ATTR_ADDRESS points to the mutex attributes to be used to initialize the mutexes used for mutual exclusion during memory allocation. Set MEM_ALLOC_MUTEX_ATTR_ADDRESS to NO_ADDR to use the default mutex attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.MUTEX_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

A.3.11.7 ADA_IO_MUTEX_ATTR_ADDRESSS

ADA_IO_MUTEX_ATTR_ADDRESS points to the mutex attributes to be used to initialize the mutexes used for mutual exclusion during Ada I/O operations. Set ADA_IO_MUTEX_ATTR_ADDRESS to NO_ADDR to use the default mutex attributes. Otherwise, set it to the address of an ADA_KRN_DEFS.MUTEX_ATTR_T record initialized in V_USR_CONF.V_INIT_ATTR().

A.3.12 #c3h: *Miscellaneous Configuration Parameters*

A.3.12.1 OLD_STYLE_MAX_INTR_ENTRY

In the current Ada RTS, the interrupt entry address clause points to an INTR_ENTRY_T record defined in ADA_KRN_DEFS. The INTR_ENTRY_T record contains two fields: the interrupt vector and the task priority for executing the interrupt entry's accept body. In earlier releases, the address clause specified the interrupt vector.

OLD_STYLE_MAX_INTR_ENTRY is provided for backwards compatibility. If the value in the address clause is <= OLD_STYLE_MAX_INTR_ENTRY, it contains the interrupt vector value and not a pointer to the ADA_KRN_DEFS.INTR_ENTRY_T record. Setting OLD_STYLE_MAX_INTR_ENTRY to 0 disables the old way of interpretation.

The default value is MEMORY_ADDRESS(511).

A.3.12.2 TASK_STORAGE_SIZE

TASK_STORAGE_SIZE specifies the size in bytes of the area in the task control block for user storage. The VADS EXEC services, ALLOCATE_TASK_STORAGE, GET_TASK_STORAGE, and GET_TASK_STORAGE2 manage this area in the task control block.

Caution – TASK_STORAGE_SIZE is valid only if using VADS MICRO

A.3.12.3 PENDING_COUNT

PENDING_COUNT specifies the maximum number of kernel service requests from a signal handler (or nested signal handler) held pending until the outermost signal handler completes. When the outermost signal handler completes, the kernel processes the queue of pending requests.

A.3.13 #c4: V_GET_HEAP_MEMORY *Routine*

```
procedure V_GET_HEAP_MEMORY(alloc_size:    in out alloc_t;
                            alloc_address:    out address);
```

V_GET_HEAP_MEMORY is the default routine called to get more memory for new allocations by the allocator (SLIM_MALLOC, FAT_MALLOC or DBG_MALLOC). ALLOC_SIZE indicates the minimum number of storage units needed. It is updated with the actual number of storage units obtained. ALLOC_ADDRESS is updated with the starting address of the heap memory obtained.

This procedure calls the Ada Kernel service, ADA_KRN_I.ALLOC, to get more memory.

The address of this routine should be assigned to the parameter GET_HEAP_MEMORY_CALLOUT in the configuration table.

A.3.14 #c5: V_PASSIVE_ISR *Routine*

```
procedure V_PASSIVE_ISR(i: v_i_pass.isr_header_ref);
       pragma external_name(V_PASSIVE_ISR, “PT_ISR”);
```

When an interrupt entry (section 13.5.1 in the *Ada Reference Manual*) is declared in a passive task, the compiler generates an ISR that looks like this:

```
<<INTERRUPT_ENTRY_ISR>>
       PUSH pas_header’address
       CALL V_PASSIVE_ISR
```

During elaboration of the task specification, the starting address of this generated ISR passes to the kernel using the Ada Kernel's ISR_ATTACH service. When a signal occurs for this vector (specified by the address clause in the interrupt entry), the operating system vectors to the generated signal handler, which calls V_PASSIVE_ISR.

The *pas_header* is a data structure built by the compiler. One *pas_header* exists per passive interrupt entry. The type that describes this header data structure is in the file *ada_location*/self/standard/v_i_pass.a. The name of the type is V_I_PASS.ISR_HEADER.

V_PASSIVE_ISR is called for all passive task interrupt entries. V_PASSIVE_ISR resides in the configuration part of the user library, making it accessible if modification is required for application-specific processing prior to calling the interrupt entry.

A.3.15 #c6: V_SIGNAL_ISR *Routine*

```
procedure V_SIGNAL_ISR(i: v_i_sig.isr_header_ref);
          pragma external_name(V_PASSIVE_ISR, "SIGNAL_ISR");
```

When an interrupt entry (section 13.5.1 in the *Ada Reference Manual*) is declared in a non-passive (interrupt) task, the compiler generates an signal handler that looks like this:

```
<<INTERRUPT_ENTRY_ISR>>
      PUSH sig_header'address
      CALL V_SIGNAL_ISR
```

During elaboration of the task spec, the starting address of this compiler-generated signal handler attaches to the corresponding Solaris 2.1 signal using the kernel ATTACH_ISR signal handler service. When an operating system signal occurs for this vector (specified by the for use at clause in the interrupt entry), Solaris 2.1 vectors to the generated ISR, which calls V_SIGNAL_ISR, as shown.

The *sig_header* is a data structure built by the compiler. One *sig_header* exists per non-passive interrupt entry. The type that describes this header data structure is in the file *ada_location*/self/standard/v_i_sig.a. The name of the type is V_I_SIG.ISR_HEADER.

Also during elaboration, a call to the Ada Kernel's CREATE_SIGNAL creates an additional runtime semaphore and task to associate with this interrupt entry. V_SIGNAL_ISR signal this semaphore to invoke the task interrupt entry.

V_SIGNAL_ISR is called for all non-passive (interrupt) task interrupt entries. V_SIGNAL_ISR resides in the configuration part of the user library, making it accessible if modification is required for application-specific processing prior to calling the interrupt entry.

A.3.15.1 #c7: V_CIFO_ISR Routine

```
procedure V_CIFO_ISR(i: v_i_cifo.cifo_isr_header_ref);
pragma external_name(V_CIFO_ISR, "V_CIFO_ISR");
```

When an interrupt entry (Ada RM, 13.5.1) is declared in a CIFO interrupt task (a task whose specification contains pragma INTERRUPT_TASK(KIND => SIMPLE)), the compiler generates an ISR that looks like this:

```
<<INTERRUPT_ENTRY_ISR>>
PUSH cifo_header'address
CALL V_CIFO_ISR
```

During elaboration of the task specification, the starting address of this generated ISR is passed to the kernel using the Ada Kernel's ISR_ATTACH service. When a signal occurs for this vector (specified by the address clause in the interrupt entry), the operating system vectors to the generated signal handler which calls V_CIFO_ISR.

The CIFO_HEADER is a data structure built by the compiler. There is one CIFO_HEADER per CIFO interrupt entry. (A CIFO interrupt task is restricted to having a single interrupt entry.) The type that describes this header data structure is in the file ada_location/standard/v_i_cifo.a. The name of the type is V_I_CIFO.CIFO_ISR_HEADER.

V_CIFO_ISR is called for all CIFO interrupt entries. V_CIFO_ISR resides in the configuration part of the user library, making it accessible if modification is required for application-specific processing before the interrupt entry's accept body is called as a normal Ada procedure.

Check the CIFO documentation for the restrictions imposed on an interrupt task.

A.3.16 #c8: V_PENDING_OVERFLOW_CALLOUT *Routine*

```
procedure V_PENDING_OVERFLOW_CALLOUT;
      pragma external_name(V_PENDING_OVERFLOW_CALLOUT,
              "V_PENDING_OVERFLOW_CALLOUT");
```

V_PENDING_OVERFLOW_CALLOUT is the routine called when an ISR (signal handler) calls a kernel service and the queue of pending requests is full.

The default action for a pending kernel service request queue overflow is to print a diagnostic message and then exit.

The V_PENDING_OVERFLOW_CALLOUT routine is valid only if using VADS MICRO.

A.3.17 #c9: V_KRN_ALLOC_CALLOUT *Routine*

```
  function V_KRN_ALLOC_CALLOUT(size: integer) return address;
        pragma external_name(V_KRN_ALLOC_CALLOUT,
"V_KRN_ALLOC_CALLOUT");
```

V_KRN_ALLOC_CALLOUT is the routine called when the microkernel needs more memory from the OS.

The default action is to call OS_ALLOC.SBRK(SIZE).

The V_KRN_ALLOC_CALLOUT routine is valid only if using VADS MICRO.

A.3.18 #c*10*: V_START_PROGRAM *and* V_START_PROGRAM_CONTINUE *Routines*

```
procedure V_START_PROGRAM;
pragma external_name(V_START_PROGRAM, “__start”);

procedure v_START_PROGRAM_CONTINUE;
pragma external_name(V_START_PROGRAM_CONTINUE,
     "__start_continue");
```

V_START_PROGRAM is the default entry point into the Ada program. pragma EXTERNAL_NAME associates this routine with the external symbol __start, the default program entry point.

For both single processor and multiprocessor Ada, V_START_PROGRAM calls the initialization routines, V_USR_CONF.V_INIT_ATTR() and V_USR_CONF.V_INIT_USR_DATA(). If you don't use the default mutex or condition variable attributes or the DEFAULT_TASK_ATTRIBUTES aren't used for the main or signal tasks, V_INIT_ATTR() needs to be changed to initialize the MUTEX_ATTR, COND_ATTR or TASK_ATTR records referenced in the configuration table. The easiest way to do this initialization is by calling one of the ADA_KRN_DEFS attribute init routines. The default version of V_INIT_ATTR has commented out examples showing how the attribute init routines can be called.

V_INIT_USR_DATA() initializes the MAIN_PRAGMAS record passed to TS_INITIALIZE. The default version of V_INIT_USR_DATA sets the fields in the MAIN_PRAGMAS record with the values initialized by pragmas in the main procedure. The main procedure can have the following pragmas:

```
pragma PRIORITY(prio: priority);

pragma SET_PRIORITY_INHERITANCE_CRITERIA;

pragma SET_GLOBAL_ENTRY_CRITERIA
        (to: queuing_discipline.discipline);

pragma SET_GLOBAL_SELECT_CRITERIA
        (to: complex_discipline.select_criteria);
```

where, only pragma PRIORITY is valid when the CIFO add-on product is not used. If you want to override the pragmas, V_INIT_USR_DATA must be changed.

If the VADS Threaded runtime is being used, the following actions are completed.

When V_START_PROGRAM completes its preliminary program initialization, it calls TS_INITIALIZE. The address of V_START_PROGRAM_CONTINUE is passed as a parameter to TS_INITIALIZE. When TS_INITIALIZE completes its tasking initialization, it calls V_START_PROGRAM_CONTINUE. V_START_PROGRAM_CONTINUE elaborates the user program's packages and executes the main program. TS_INITIALIZE returns back to V_START_PROGRAM when the program is ready to exit.

If the Solaris MT runtime is being used, the following steps are taken by V_START_PROGRAM and V_START_PROGRAM_CONTINUE.

Here are the steps of V_START_PROGRAM:

1. **Gets arguments and UNIX environment, and saves them for later revival by the user program.**
2. **Initializes Solaris Threads.**
3. **Initializes Solaris data and stack limits.**
4. **Initializes Ada specific tasking structures.**
5. **Jumps to** V_START_PROGRAM_CONTINUE

Here are the steps of V_START_PROGRAM_CONTINUE:

1. **Initializes memory allocator.**
2. **Initializes initial number of LWPs for multiprocessing.**
3. **Elaborates the user program's packages and executes the main subprogram.**
4. **Calls the ADA_EXIT routine which does user program cleanup (such as closing TEXT_IO files) and returns control to kernel.**

Note – Although this routine is included in package V_USR_CONF, you *should not* need to modify it.

"But, soft! what light through yonder window breaks?"

Shakespeare

XView Interface and Runtime System B

XView™ (X Window-System-based Visual/Integrated Environment for Workstations) is a toolkit providing a windowing interface through which you support interactive, graphics-based applications.

Before using this software, be completely familiar with the XView product and documentation. The following documents describe the XView user-interface toolkit:

- Heller, Dan, *XView Programming Manual*, O'Reilly & Associates, Inc., 1990
- *XView Version 2 Reference Manual: Converting SunView Applications*, Sun Microsystems, Inc., 1990 (Part Number: 800-4836-10)
- *Open Window Version 2 User's Guide*, Sun Microsystems, Inc., 1990 (Part Number: 800-4930-10)

In this manual, we provide discussions of how to integrate XView with SC Ada, and descriptions of the Sun extensions to support XView.

The XView interface and runtime system product is installed with SC Ada.

B.1 Product Description

Sun XView software consists of two parts: a runtime system (called The Notifier) to manage input, and building blocks to control output. The XView runtime system controls all events, as well as communication in windows, between windows, and between windows and the operating system.

B

SC Ada XView is a toolkit made up of tools, data structures, and a custom version of the SC Ada Runtime System, which enables you to build and use graphics applications in a windowing environment and manipulate XView objects. To support the use of SC Ada with XView, we provide an interface enabling SC Ada and XView to communicate with each other.

B.2 How To Use SC Ada With XView

Three directories are supplied with this release of SC Ada XView reside in the directory *ada_location*`/self` and *ada_location*`/examples/xview_examples`.

`xview` is an Ada library. When you write Ada programs that interface to XView, use `a.path` to put it on the `ADAPATH` line of your `ada.lib`. Use the source code from the `examples/xview_examples` directory to build sample Ada programs that interface to XView. Look at the corresponding source code to see specific examples of how SC Ada interfaces to XView.

B.2.1 XView Library

The XView library contains Ada package specifications and bodies that parallel the similarly named C header files for XView in the `include` directories `/usr/openwin/include/xview` and `/usr/include`. Follow the documentation for XView, and substitute Ada package specifications for C header files, except for some necessary changes discussed later in this manual.

B.2.2 XView Examples

The `xview_examples` directory contains several example programs in this beta release. Other test programs are included with the XView product release.

Build these programs by following these steps (note that *ada_location* is the directory where you installed the SC Ada XView product, and `xview` is the Ada library you are using, depending on which version of Open Windows you are running under):

```
mkdir examples
cd examples
...
a.mklib . ada_location/self/xview_vN
```

If you are unfamiliar with `a.mklib`, refer to the appropriate entry in the *SPARCompiler Ada Reference Guide.*

```
cp ada_location/examples/xview_examples/* .
```

Refer to *SPARCompiler Ada Reference Guide* for more information about `a.path`.

```
make
```

To run your example programs, first run `xview` and get in a shell or console window.

B.2.3 Compiling and Linking Programs

The only requirement for compiling programs is that the XView Ada library supplied with the release must be the parent library or must be referenced along with the `vads_exec` library on the `ADAPATH`. Use `a.path` to accomplish this.

In the list of files on the `ADAPATH`, `xview` must appear before `standard` because it contains directives that override those in `standard`.

The Ada library `xview` is set up so you do not need any special commands when linking. Therefore, if you have a simple Ada program that uses XView, compile and link it by using these commands:

```
ada test_prog.a
a.ld -o testprog testprog
```

The `ada.lib` file in *ada_location*`/self/xview` has the following directives that play an important role in this release of the XView product (note the use of *ada_location*`/self` to represent the directory where you installed XView). See Figure B-1.

```
!ada library
ADAPATH= ada_location/self/vads_exec
ada_location/self/standard  LIBRARY:LINK
ada_location/self/xview/lib/library.a
ada_location/self/xview/lib/svi_struct.a /usr/openwin/lib/libxview.a
   /usr/openwin/lib/libolgx.a /usr/openwin/lib/libX11.a:TASKING:LINK:
ada_location/self/xview/lib/tasking.a
ada_location/self/xview/lib/svi_struct.a /usr/openwin/lib/libxview.a
  /usr/openwin/lib/libolgx.a /usr/openwin/lib/libX11.a:
```

Figure B-1 Example of XView Directives in `ada.lib` File

Note – The example above references the libraries `libxview.a`, `libolgx.a`, and `libX11.a` in the directory `/usr/openwin`. If your libraries are in a different location, change the directory path to point to your libraries.

B.2.4 Interface Limitations

- Support for the macro `DEFINE_ICON_FROM_IMAGE` is not guaranteed for future releases of XView. Therefore, this macro is not supported in the SC Ada XView bindings.
- These bindings represent a direct translation into Ada of the most critical C `include` files for XView. Every effort is made to provide a complete translation. However, during the translation process, references to several function calls and attributes were found in the `include` files for which no documentation exists. In these cases, the interface is provided and the most likely parameters are supported.

  This is not a limitation since these routines are not intended for general use, and they do not appear frequently. In each case, the function is commented or marked with the pattern "`--ND`." If the parameters to these undocumented routines are made public at a later date, they will be updated in the XView bindings.

- Some attribute types contain duplicate attribute/value pair possibilities. For example, the attribute PANEL_VALUE uses many value types. Pass a value for this attribute with the function XVI_AV_FUNCTIONS.CONVERT_VAL() that converts either strings or integers to the appropriate type for PANEL_VALUE.
- Attributes requiring null terminated lists of SVI_STRING, SERVER_IMAGE, EVENT, or INTEGER values are not supported in this release.
- The following attributes are not supported at this time: ATTR_LIST, CANVAS_PAINT, MENU_ITEM, OPENWIN_SPLIT, OPENWIN_VIEW_ATTRS. Support for these attributes will be added in a future product release.

B.2.5 Notifier Limitations

Note – In the following discussion, NOTIFY_TASK refers to the task containing the call to V_NOTIFY_MAIN_LOOP (or V_XV_MAIN_LOOP). Also, V_WINDOW_ENTER/V_WINDOW_LEAVE are interchangeable with V_NOTIFY_ENTER/V_NOTIFY_LEAVE.

Using the XView Notifier in conjunction with Ada tasks has these restrictions:

- Notifier event handlers (excluding asynchronous signal event handlers) can interact with other Ada tasks via rendezvous, task resume, semaphores, mailboxes or any other mechanism for task interaction. However, these event handlers execute in the context of the NOTIFY_TASK, where the NOTIFY_TASK has entered the IN_NOTIFY semaphore already. If an event handler blocks, it blocks all Notifier activity also.

Note – Since enters/leaves of IN_NOTIFY semaphore can be nested from the same task, Notifier event handlers can call V_NOTIFY_ENTER/V_NOTIFY_LEAVE.

- An asynchronous signal event handler has the same restrictions for Ada task interaction as an Interrupt Service Routine (ISR); they cannot call any kernel service that blocks. We strongly recommend that signal events be registered as synchronous.

- Protect all calls to the notify (and window) services via V_NOTIFY_ENTER/V_NOTIFY_LEAVE. However, V_NOTIFY_MAIN_LOOP (or V_XV_MAIN_LOOP) does not need to be protected. It protects all of its calls to notify services.
- The kernel calls notify timer routines for doing delays, servicing timeouts, and task time slicing. Actual handling of these requests is deferred until the Notifier dispatcher is called from V_NOTIFY_MAIN_LOOP in the NOTIFY_TASK. If the NOTIFY_TASK is of lower or equal priority to the current task, these timer requests are deferred until the current task blocks. We strongly recommend that the NOTIFY_TASK execute at a higher priority than any other task. Also, all V_NOTIFY_ENTERs must use the default highest priority value.
- Both the SC Ada kernel and XView make direct calls to the C `malloc(3)` and `free(3)` memory routines. Reentrant access to these C routines can lead to unexpected results. Executing all XView services at the highest priority (as is the default) inhibits their preemption and avoids a reentrant call to the `malloc`/`free` routine from the SC Ada kernel. That is, the kernel only makes such a call on behalf of some tasking request, for example, create/abort another task.
- An Ada program waiting for another event, such as a mouse click, terminates unless a task is ready to run or on the delay queue. For XView applications with Ada tasks, include a dummy task containing: `loop delay 86000.0; end loop;` to inhibit premature program termination. Call the VADS EXEC service, V_XTASKING.SET_EXIT.DISABLED, to inhibit the program from exiting.

B.3 The SC Ada XView Interface

This section provides information about the interface between SC Ada and XView, which includes a description of the SC Ada/XView interface, details of how the interface functions, and interface programming information.

This interface provides a mechanism for using Ada, in an XView environment, to create applications. This mechanism is supplied with a combination of structure mapping, data type translation, and tool interfaces. As much as possible, we kept the Ada programmer interface identical to the C-language programmer interface. By keeping the interfaces for the two languages similar, the programmer can use the XView documentation with little or no additional

instruction. Also, a similar programmer interface simplifies the task of translating existing C-language XView programs to Ada while maintaining their executable integrity.

B.3.1 Interface Package Structure

The XView Ada interface package is organized almost identically to the C `include` files used by XView. Where feasible, the complete `.h` file is translated to an Ada package specification, making all functions, procedures, macros, and structures available. In most instances where an `include` file requires additional `include` files, the additional `include` files are translated. This ensures a complete implementation and keeps the interface structure compatible with the organization provided for the C version of XView.

For example, the `include` file that contains most of the essential definitions for panels, `panel.h`, is translated directly and is contained in `package XVI_PANEL` consisting of the files `xvi_panel.a` and `xvi_panel_b.a`. In this package, `function PANEL_TEXT_NOTIFY` requires a pointer to an object of `type EVENT`. Because the type declaration for an object of `type EVENT` is in the C `include` file `win_input.h`, this file is translated also. The translation is in `package WIN_INPUT`, which is in the files `xvi_win_input.a` and `xvi_win_input_b.a`. `package PANEL` refers to `package WIN_INPUT`, using an Ada `with` statement to access the declaration.

Five exceptions apply to these conventions:

- The C include file `xview.h` includes other `.h` files to provide all the necessary definitions for a XView C program. In this implementation, `xview.a` contains necessary definitions that do not fit well anywhere else. Unlike C, Ada programs must explicitly make all declarations visible using `with` statements.
- The `include` file `window.h` is in two `packages`, `WINDOW` and `WIN_FUNC`. `package WINDOW` is in the file `xvi_window.a`, and `package WIN_FUNC` is in the two files `xvi_win_func.a` and `xvi_win_func_b.a`. This separation is necessary because window functions rely on declarations in other packages. Many of these packages rely on declarations in the window package. The translation of the file `window.h` is separated to avoid circular definitions.

- Many special type declarations, as well as functions and procedure specifications, are required to implement attribute/value (AV) lists. AV lists and the implementation are described in more detail later in this document. AV list structure and data type declarations are contained in `package XVI_AV_LIST` in the file `xvi_av_list.a`. All functions that use AV lists, from all C `include` files translated, are moved to `package XVI_AV_FUNCTIONS` consisting of the files `xvi_av_functions.a` and `xvi_av_functions_b.a`.
- Some special extensions to XView supporting Ada Tasking are added to `xvi_win_func.a`.
- We supply `package XVI_U_ENV` to give XView Ada Programs command line capabilities similar to those available in C. For a simple example of the usage of this package, refer to the demo program, `hello.a`, provided in the `examples` directory of this release.

B.3.2 Data Type Naming Conventions

We attempted to give the same names to the Ada types, functions, and procedures as the corresponding C types, functions, and procedures. Cases exist where this is not possible because of the case insensitivity of Ada. Most conflicts are resolved by extending the type name rather than changing it. For example, given an enumerated type in C declared as `BAR`, a record declared as `BAR`, and a function declared as `BAR` all in the same `include` file, the declarations are changed in the interface package to an enumerated type named `BAR_A`, a record type named `BAR_REC`, and a function called `BAR`.

In addition, many C declarations require pointers to structures. In these cases, an access type is provided with the suffix `_PTR`. This enables pointers to structures to be used while preserving Ada type checking. To illustrate, given a function that returns a pointer to a record of `type FOO`, the Ada interface declares the same function returning `type FOO_PTR`. In the interface package, `type FOO_PTR` is declared as an access type to the record `type FOO`.

B.3.3 Differences In This Implementation

Below is a summary of significant differences between the XView interface and the C interface.

B.3.3.1 Attribute/Value Lists and Functions

XView implements variable length attribute/value (AV) lists. The basic idea is that a relatively small group of functions create or modify XView objects. However, by using AV lists, each function can perform a wide variety of tasks or modify many object characteristics.

The SC Ada interface implements variable length AV lists. Two ways exist to implement this in Ada. First, each function using AV lists can be overloaded for each possible combination of attributes, values, and parameters required. Second, each function requiring AV lists can accept an unconstrained array of variant records. Because of the large number of different AV list combinations possible, the first method is inefficient and impractical. Consequently, the second method is chosen.

In this implementation, each function requiring an AV list accepts an array of variant records specific to the function and type of operation. The XView function is called by pushing the AV list, along with any other required parameters, on the stack and calling the C function using an Ada machine code insertion. The variant record and array declarations specific to AV lists are in Ada `package XVI_AV_LIST`. The declaration of the functions requiring AV lists, as well as the parameter-passing and function-calling routines, are in Ada `package XVI_AV_FUNCTIONS`.

Fortunately, most of these differences are invisible from the application level. Because AV lists are implemented as arrays of variant records, the major difference between the Ada and the C versions is the need for additional parentheses and a different method of list termination in the Ada model. The following two examples illustrate a call to `function XV_CREATE` in C and a call to the same function using the Ada interface package. See Figure B-2.

```
/* C CODE EXAMPLE */
my_frame = xv_create(BASE_WINDOW, FRAME,
               XV_LABEL, “Hello World”, 0);

-- Ada CODE EXAMPLE
my_frame := xv_create(xvi_window.BASE_WINDOW, xvi_xview.FRAME,
                ((XV_LABEL, SVI_STR(“Hello World”)),
                (attr => FRAME_NO_ATTR)
                ));
```

Figure B-2 Example of Attribute/Value Lists and Functions

In the example, the Ada program requires forming an array by using parentheses to group attribute/value pairs and to delineate the beginning and end of the array. The example shows that termination of the AV list requires a named declaration of the last variant record in the array. The special attribute FRAME_NO_ATTR (this name changes between object types, but always has the form FOO_NO_ATTR, where FOO is the LIST type) is provided for each object type to terminate the AV lists. The example illustrates the use of SVI_STRINGS, which is discussed in a later section.

Most attributes require only a single value or no value, but in some cases, a single attribute requires a null terminated list of values. In these cases, a family of utility routines is available. These routines are named FOO_LIST, where FOO is one of STR, INT, SHORT, CADDR_T, PIXRECT or PIXFONT, depending on the type of values contained in the list. This means that function INT_LIST passes a list of integer values, STR_LIST passes a list of SVI_STRING values, and so on.

B.3.3.2 Pointers

As noted earlier, pointers in this interface are declared currently as access types for the appropriate data type. Pointers are given the same name as the type they point to, except the suffix _PTR is appended. Cases exist where an access type is either impractical or very difficult to use. In these cases, pointers are declared as address types. Both work equally well — as long as the right type of data structure is at the address used. However, using address types directly eliminates many of the advantages of type checking provided by Ada and greatly increases the chances of a program error the compiler cannot detect.

B.3.3.3 Strings

This version of XView includes a new design for handling strings. Strings in C are represented by a pointer to a string of characters terminated by a null. We provide package C_STRINGS to help represent C strings in most Ada application programs. In addition, we supply package A_STRINGS, which implements variable length Ada strings. When used together, these packages support most applications. However, the XView interface presents some very special problems.

The biggest problem is that a single C string represented in the old way takes up a great deal of space in memory. When building XView application programs, many C_STRINGS must be allocated and passed to interface routines. Many of these strings are not used again once passed to XView. Further, using the old package C_STRINGS, it is difficult to deallocate strings when they are no longer needed.

To deal with these problems, we provide package SVI_STRINGS with the XView bindings. This package provides routines to create, manipulate, and deallocate fixed-length SVI_STRINGS in Ada programs. A complete set of routines providing for automatic deallocation of SVI_STRINGS is supplied. Details of this implementation are provided in the paragraphs that follow.

String Definition

```
type svi_string is access string;
```

In package SVI_STRINGS, an SVI_STRING is declared as an access type to a string. With this definition, it is necessary to specify the length of the string when space is allocated. To ease this process, SVI_STRINGS provides string creation routines.

String Creation

```
function to_svi(str: string) return svi_string;
function svi_str(str: string) return svi_string;
```

function TO_SVI and function SVI_STR return an SVI_STRING, given a valid Ada string or a quoted group of characters. The difference between these routines is the way in which memory is deallocated. With function TO_SVI the programmer is responsible for explicitly deallocating space when a string is no longer needed. Memory deallocation is accomplished using the free routine described under the heading string deallocation.

In contrast to function TO_SVI, function SVI_STR keeps track of all strings it allocates. After each call to an interface routine using AV_LISTS, all memory allocated by SVI_STR since the last call to a routine that requires AV_LISTS is deallocated. This is most useful when strings are used in AV_LISTS. However, use this method anywhere automatic deallocation is required. Routines for explicitly deallocating space allocated by SVI_STR are provided and are described further under the heading string deallocation.

String Manipulation

```
function svi_strlen(in_str: svi_string) return integer;
function svi_strlen(in_str: string) return integer;
```

function SVI_STRLEN accepts either a string, an SVI_STRING or a quoted group of characters. The returned-integer value indicates the number of characters in the input up to the first null character. If a null character is not found, SVI_STRLEN returns the total number of elements in the string.

```
function svi_strcat(str1, str2: string) return svi_string;
function svi_strcat(char: character; str2: string) return svi_string;
function svi_strcat(str1: string; char: character) return svi_string;
```

function SVI_STRCAT accepts either two strings or one string and one character. The result is a SVI_STRING that points to a concatenated string of the input values. The SVI_STRING returned from SVI_STRCAT must be explicitly deallocated using a call to free when it is no longer needed.

String Deallocation

```
procedure svi_str_free_all;
procedure free is new unchecked_deallocation(string,
svi_string);
```

procedure FREE and procedure SVI_STR_FREE_ALL deallocate SVI_STRINGS created with TO_SVI and SVI_STR respectively. procedure FREE requires a single parameter, the SVI_STRING the programmer wants to deallocate. procedure SVI_STR_FREE_ALL requires no parameters. This procedure deallocates all SVI_STRINGS allocated by calls to SVI_STR, since the last call to a function requiring AV lists. procedure SVI_STR_FREE_ALL is called automatically after every call to a routine that uses AV lists.

String Lists

Some of the attributes used in XView require a pointer to a list of strings terminated by a null string. In this interface, package lists of strings are represented by a list of nodes containing SVI_STRINGS. When AV list parameters are pushed on the stack, this representation converts to the correct C style representation. Therefore ttributes that require lists of strings accept only a parameter of type STR_PTR_LIST. Parameters of this type must be built using special string list creation and deallocation routines.

String List Definition

```
type str_list_type is array(natural range <>) of svi_string;
type str_ptr_list is access str_list_node;
```

The definition of a string list as used in this interface is shown in the two lines above. The declaration of STR_LIST_TYPE describes the input required by string list creation routines. The declaration of STR_PTR_LIST describes the value returned by string creation routines. String lists are deallocated explicitly by the programmer or automatically using XView interface routines.

String List Creation

```
function svi_str_list(strings: str_list_type) return str_ptr_list;
function str_list(strings: str_list_type) return str_ptr_list;
```

function SVI_STR_LIST and function STR_LIST return pointers to lists of nodes containing SVI_STRINGS. The difference between these routines is the method of memory deallocation. function STR_LIST allocates the space required and builds the list. However, memory deallocation becomes the responsibility of the programmer. With function SVI_STR_LIST, all string lists are deallocated automatically after each call to a function that requires AV lists.

String List Deallocation

```
procedure svi_str_list_free_all;
procedure free is new unchecked_deallocation(str_list_node,
    str_ptr_list);
```

procedure SVI_STR_LIST_FREE_ALL deallocates all space allocated by calls to SVI_STR_LIST since the last call to a function that requires AV lists. This procedure is called automatically after each call to a routine that requires AV lists. In most cases, SVI_STR_LIST_FREE_ALL is not called directly by the programmer.

procedure FREE deallocates STR_LIST_NODE objects directly. When using this procedure, you must walk the list of nodes and explicitly deallocate each. This method of deallocating string lists is not recommended.

Pointer Lists

Certain attributes used in XView require lists of integers, short integers, CADDR_T objects, pixrect pointers or pixfont pointers. In the XView interface, these lists are supplied by the programmer as arrays of the appropriate object. These arrays are used by special creation routines to return the pointer expected by the XView interface. Declaration of these objects, as well as declarations and explanations of the allocation and deallocation routines, are provided in the following section.

Pointer List Definitions

```
type int_list_type is array(natural range <>) of integer;
type int_value_list is access int_list_node;

type short_list_type is array(natural range <>) of short_integer;
type short_value_list is new int_value_list;

type caddr_t_list_type is array(positive range <>) of caddr_t;
type caddr_t_value_list is access caddr_t_list_node;

type pixrect_ptr_list_type is array(positive range <>) of pixrect_ptr;
type pixrect_ptr_list is access pixrect_ptr_list_node;

type pixfont_ptr_list_type is array(positive range <>) of pixfont_ptr;
type pixfont_ptr_list is access pixfont_ptr_list_node;

type server_image_list_type is array(positive range <>) of
    server_image;
type server_image_list is access server_image_list_node;
```

The declaration of the arrays supplied to creation routines by the programmer and the actual pointers required by the bindings are outlined above. After an array of objects or values is passed to XView using the pointer list creation routines, lists are deallocated automatically. Procedures that explicitly deallocate these lists are provided in package XVI_POINTERS.

Pointer List Creation

```
function int_list(ints: int_list_type) return int_value_list;
function short_list(shorts: short_list_type)
    return short_value_list;
function caddr_t_list(caddrs: caddr_t_list_type)
    return caddr_t_value_list;
function pixrect_list(pixrect_ptrs: pixrect_ptr_list_type)
    return pixrect_ptr_list;
function pixfont_list(pixfont_ptrs: pixfont_ptr_list_type)
    return pixfont_ptr_list;
function image_list(images: server_image_list_type)
    return server_image_list;
```

Pointer lists are created by passing an array of the required element type to the appropriate function. Functions are provided for creating lists of integers, short integers, `CADDR_T` objects, pixrect pointers, and pixfont pointers. These `function`s are named `FOO_LIST` where `FOO` is int, short, `CADDR_T`, pixrect or pixfont, depending on the type of element the list contains. For example, `function INT_LIST` is for creating lists of integers, `function PIXRECT_LIST` is for creating lists of pixrect pointers, and so on.

Pointer List Deallocation

```
procedure xvi_int_list_free_all;
procedure xvi_ct_list_free_all;
procedure xvi_pr_list_free_all;
procedure xvi_pf_list_free_all;
procedure xvi_si_list_free_all;
```

The four procedures for creating lists of pointers provide automatic deallocation of memory allocated for list creation. Do this in the XView bindings by calling one of the four deallocation procedures shown above after each call to a routine that requires AV lists. Call the deallocation routines directly at any time. In most cases, you never need to explicitly deallocate pointer lists.

B.3.3.4 Return Values

Some functions return values of `type XV_OPAQUE`. The returned value can be many things. The correct way to solve this problem is to overload these functions for each potential correct return value. Presently, only `type XV_OPAQUE` is returned, so use `UNCHECKED_CONVERSION` to get the correct return value type.

B.4 The SC Ada Kernel

This section provides information about the SC ADA kernel for this product. The section includes information about integrating the XView Notifier with Ada tasking and descriptions of the packages and routines added to the runtime system.

B.4.1 Integrating the XView Notifier With Ada Tasking

The XView Notifier assumes that any UNIX process interacting with XView has only a single thread of execution. For almost all C programs on UNIX, this is a valid assumption. However, for Ada processes having tasking, this assumption does not apply.

An SC Ada process can contain multiple tasks. The VADS EXEC runtime system executes entirely in a UNIX process and switches among the Ada tasks according to the semantics of Ada. Time slicing is supported; if multiple Ada tasks have the same highest priority, VADS EXEC uses a timer to switch between them, in a round-robin fashion.

This means that one Ada task can interact with The Notifier when the timer goes off and VADS EXEC switches to another task. If this task tries to interact with The Notifier, this can create problems. To avoid problems, serialize access to The Notifier.

The Notifier places restrictions on the UNIX services used by a UNIX process. Some of these services, such as `sigvec(2)`, `setitimer(3)`, are used normally by VADS EXEC, so it was necessary to build a new version of VADS EXEC that uses the UNIX services as prescribed by The Notifier.

Therefore, in addition to explicit and implicit calls on The Notifier that can be present in the Ada program, VADS EXEC makes calls to The Notifier.

For more information about the XView Notifier, refer to the XView documentation.

B.4.2 Serializing Access to The Notifier

VADS EXEC uses a semaphore called IN_NOTIFIER to ensure that one task at a time is executing Notifier functions. Routines are added for entering and leaving this IN_NOTIFIER semaphore (V_NOTIFY_ENTER and V_NOTIFY_LEAVE).

The routine, V_NOTIFY_MAIN_LOOP, loops and calls the XView Notifier routine NOTIFY_DISPATCH. NOTIFY_DISPATCH dispatches events managed by The Notifier. The call to NOTIFY_DISPATCH by V_NOTIFY_MAIN_LOOP is bracketed by the above-mentioned V_NOTIFY_ENTER and V_NOTIFY_LEAVE routines. At the conclusion of dispatching events via NOTIFY_DISPATCH, V_NOTIFY_MAIN_LOOP task suspends on its DO_DISPATCH semaphore. The task containing V_NOTIFY_MAIN_LOOP is resumed when any asynchronous UNIX signal such as sigalrm or sigint signals its DO_DISPATCH semaphore. Additionally, V_NOTIFY_MAIN_LOOP has a timeout parameter. This places an upper limit on how long it suspends before recalling NOTIFY_DISPATCH. If V_NOTIFY_MAIN_LOOP is called with IN_NOTIFIER semaphore set by V_NOTIFY_ENTER, it calls V_NOTIFY_LEAVE before it loops.

The routine, V_NOTIFY_STOP_LOOP, is added to terminate V_NOTIFY_MAIN_LOOP. This returns V_NOTIFY_MAIN_LOOP to its caller at the conclusion of event dispatching. V_NOTIFY_MAIN_LOOP returns with the IN_NOTIFY semaphore restored to its state upon entry.

The Notifier manages the interval timer and UNIX signals. The Ada DELAY, DELAY_UNTIL, TIMED_CALL, and TIMED_SUSPEND services are changed to use The Notifier routine, NOTIFY_SET_ITIMER_FUNC, instead of making a direct call to the system routine, SET_ITIMER. (Note that the call to NOTIFY_SET_ITIMER_FUNC does not take effect until the next invocation of NOTIFY_DISPATCH. After calling NOTIFY_SET_ITIMER_FUNC, the kernel signals the above DO_DISPATCH semaphore.) Furthermore, the SC Ada UNIX signal handling is changed to call notify_set_signal_func() instead of sigvec().

In order to resume the dispatching of Notifier events, the DO_DISPATCH semaphore must be signaled. Do this by installing a prehandler for each UNIX signal event. This prehandler is registered automatically during start-up initialization for the following signals:

SIGALRM (14)	SIGPROF (27)	SIGUSR2 (31)
SIGCHLD (20)	SIGTERM (15)	SIGVTALRM (26)
SIGCONT (19)	SIGTTIN (21)	SIGWINCH (28)
SIGIO (23)	SIGTTOU (22)	SIGXCPU (24)
SIGLOST (29)	SIGURG (16)	SIGXFSZ (25)
SIGPIPE (13)	SIGUSR1 (30)	

Events for the remaining UNIX signals must be registered via the following added routine, V_NOTIFY_SET_SIGNAL_FUNC() (note that NOTIFY_SET_SIGNAL_FUNC in the notify package specification maps automatically to this routine). V_NOTIFY_SET_SIGNAL_FUNC() simply registers the prehandler before registering the user-specific event handler.

The PRE_HANDLER signals the DO_DISPATCH semaphore, saves the current sigcontext, and updates the sigcontext to transfer control to COMPLETE_PRE_HANDLER at the conclusion of UNIX signal processing. On conclusion of signal handling, instead of transferring control to the interrupted program, control transfers to COMPLETE_PRE_HANDLER. COMPLETE_PRE_HANDLER enters the VADS EXEC kernel to enable preemption of the current task and resumption of V_NOTIFY_MAIN_LOOP.

B.4.3 package XVI_NOTIFY

The following routines are added to enable the coexistence of the XView Notifier with Ada tasking. package XVI_NOTIFY includes the interface to these routines.

```
procedure v_notify_enter(notify_priority: priority := priority'last);
procedure v_notify_leave;
```

These routines guarantee serialized access to The Notifier services by entering/leaving the IN_NOTIFIER semaphore.

You must protect any call to a Notifier service (such as NOTIFY_POST_EVENT()) by preceding or following it with a call to V_NOTIFY_ENTER/V_NOTIFY_LEAVE.

These routines enable nested Notifier enters/leaves from the same task. If the task is in rendezvous with a task in The Notifier, it is granted immediate entry. (Note that all enters must be paired with an equal number of leaves.)

The V_NOTIFY_ENTER routine has one parameter, task priority while doing notify functions. The default is highest priority. Since The Notifier dispatcher must be called to do any timer reprogramming (including task time slicing), we strongly recommend NOTIFY_PRIORITY remains at the default value. The task priority is restored on leaving the IN_NOTIFIER semaphore.

```
procedure v_notify_main_loop(notify_timeout: duration := 0.200;
    notify_priority: priority := priority'last);
```

This routine contains the loop for doing repetitive dispatching of Notifier events. Only one invocation of this routine can be active at a time. The TASKING_ERROR exception is raised for subsequent concurrent invocations.

The NOTIFY_TIMEOUT parameter sets an upper limit on the time between the dispatching of Notifier events. A zero or negative value implies no upper limit.

The NOTIFY_PRIORITY parameter specifies the task priority while doing Notifier dispatching. The default is highest priority. Since The Notifier dispatcher must be called to do any timer reprogramming (including task time slicing), we strongly recommend that the NOTIFY_PRIORITY remain at the default value.

```
procedure v_notify_stop_loop;
```

This routine is called to stop the above V_NOTIFY_MAIN_LOOP.

```
function v_notify_set_signal_func(client: Notify_client_t;
        signal_func: Notify_func;
        signal: integer;
        mode: Notify_signal_mode)
    return Notify_func;
```

This routine is identical to the XView NOTIFY_SET_SIGNAL_FUNC() except that it first registers a PRE_HANDLER. This PRE_HANDLER preempts the current Ada task and resumes execution of the V_NOTIFY_MAIN_LOOP.

Note – The NOTIFY_SET_SIGNAL_FUNC subprogram declaration included in the XVI_NOTIFY package maps to V_NOTIFY_SET_SIGNAL_FUNC.

B.4.3.1 XVI_WIN_FUNC *Package Extensions*

The following routines are added to enable the coexistence of XView windows with Ada tasking. The interface to these routines is included in the specification of package XVI_WIN_FUNC.

```
procedure v_xv_main_loop(base_frame: xvi_frame.Frame;
    notify_timeout: duration := 0.200;
    notify_priority: priority := priority'last);
```

This subprogram makes the frame visible on the screen and calls V_NOTIFY_MAIN_LOOP. Its input parameters pass directly to V_NOTIFY_MAIN_LOOP.

V_XV_MAIN_LOOP interposes in front of the frame destroy event handler. It stops the V_NOTIFY_MAIN_LOOP on receiving a destroy event.

This subprogram has the same restriction as V_NOTIFY_MAIN_LOOP, only one invocation can be active at a time.

Note – The WINDOW_MAIN_LOOP subprogram declaration included in the package WIN_FUNCTIONS maps to V_XV_MAIN_LOOP.

```
procedure v_window_enter(window_priority: priority := priority'last);
procedure v_window_leave;
```

These subprograms provide alternate names to the V_NOTIFY_ENTER/V_NOTIFY_LEAVE subprograms.

Note – Precede/follow any call to a window service by a call to V_WINDOW_ENTER/V_WINDOW_LEAVE (or alternatively V_NOTIFY_ENTER/V_NOTIFY_LEAVE).

"Fast bind, fast find; A proverb never stale
in thrifty mind."

Shakespeare

POSIX Conformance Document C

C.1 Introduction

This POSIX.5-1990 Conformance Document describes those items that must be documented for SPARCompiler Ada (SC Ada) to claim conformance to it, as specified in the POSIX.5.1990 standard as implementation-defined.

This document applies to the IEEE Standard POSIX Ada Language Interfaces, IEEE Standard 1003.5-1990, referred to herein as POSIX.5.

This conformance document has the same structure as the POSIX.5 Standard, with information presented in the equivalently numbered sections, clauses and subclauses. Only those sections, clauses or subclauses in POSIX.5 requiring documentation for undefined or unspecified actions are included in this conformance document.

The SC Ada implementation of POSIX.5.1990 relies on the existence of POSIX.1.1990, or the IEEE Standard Portable Operating System Interface for Computer Environments, on the system that POSIX.5 is to be installed on. Therefore some of the values requiring documentation in POSIX.5 will be determined by POSIX.1.

C.1.1 Release Structure

The SC Ada implementation of POSIX.5 is located in the directory *SCAda_location*`/self/posix` as shown in Figure C-1 on page C-2.

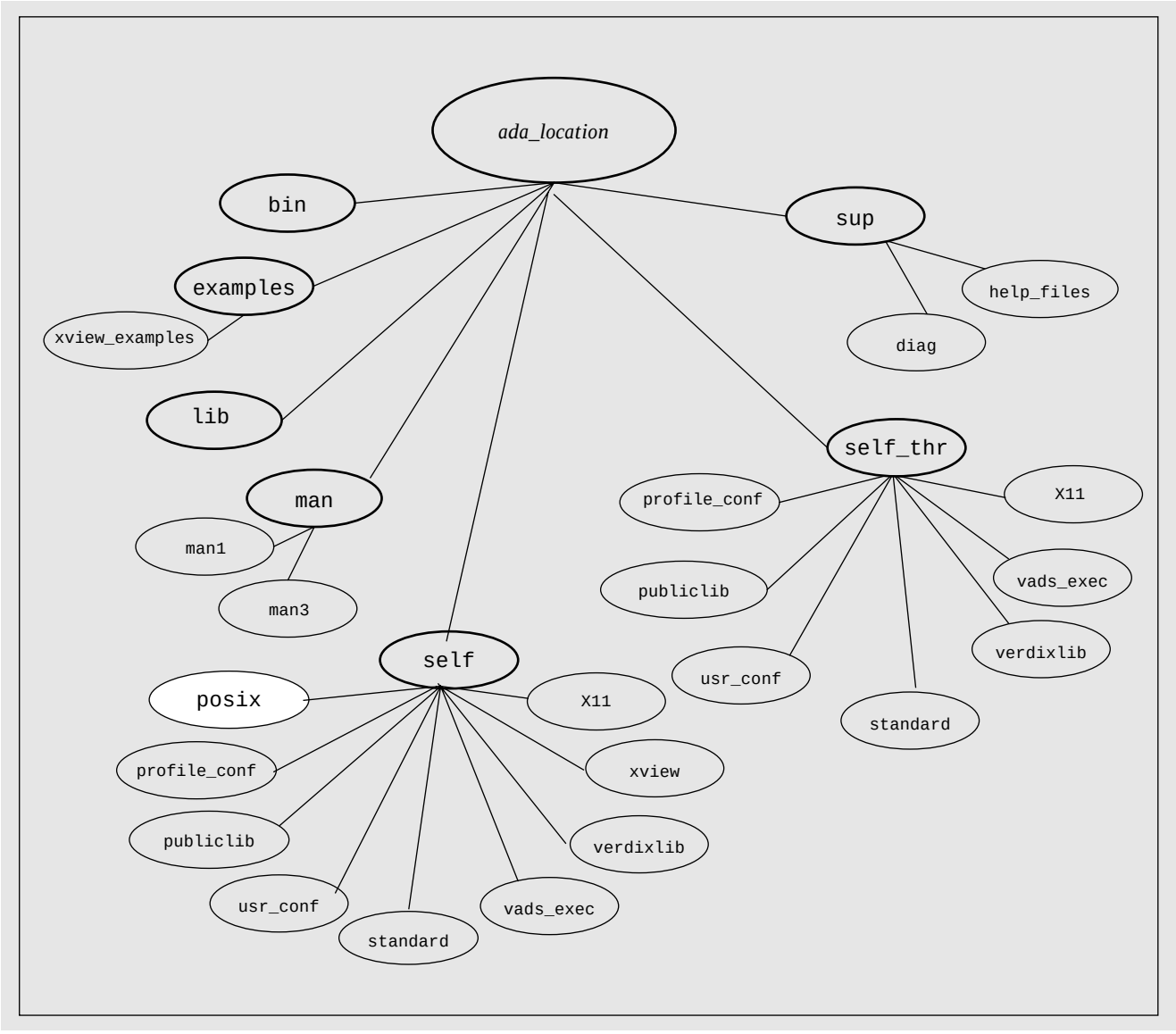

Figure C-1 SPARCompiler Ada Implementation of POSIX.5

Note – NOTE: *SCAda_location*/`self/posix` must be on your adapath to compile and link with the POSIX library.

C.2 Terminology and General Requirements

C.2.1 Definitions

C.2.1.1 General Terms

Appropriate Privilege

A process with an effective user ID of zero (which is known as the super-user's user ID) has all appropriate privileges.

Character Special File

Please refer to POSIX Conformance Document provided by Sun for POSIX-1990.

File

See “package POSIX_FILES” on page C-22 for addition information about links.

File Group Class

No additional members of the file group class of a file are defined other than those defined in POSIX.

Parent Process ID

After the lifetime of the creator has ended, the parent process ID is `POSIX_PROCESS_IDENTIFICATION.SYSTEM_PROCESS_ID`. Refer to “package POSIX_PROCESS_IDENTIFICATION” on page C-19.

Pathname

Multiple consecutive slashes in a pathname are treated as equivalent to a single slash. Multiple consecutive leading slashes in a pathname are treated in the same manner as multiple slashes elsewhere in the pathname.

Read-Only File System

Files and directories on a read-only file system may only be read, not written to or updated.

C.2.2 General Concepts

C.2.2.1 Extended Security Control

See Appropriate Privilege in “General Terms” on page C-3 and File Access Permissions below.

C.2.2.2 File Access Permissions

No additional or alternate file access control mechanisms are provided.

C.2.2.3 File Times Update

No time-related fields are defined other than those defined in POSIX.5. Fields “marked for update” are immediately updated.

C.2.3 package POSIX

```
package POSIX is

  -- Symbolic subtypes and constants

   -- Optional Facilities
 subtype Job_Control_Support is Boolean range
              FALSE .. TRUE;
 subtype Saved_IDs_Support is Boolean range
              FALSE .. TRUE;
 subtype Change_Owner_Restriction is Boolean range
              FALSE .. TRUE;
 subtype Filename_Truncation is Boolean range
              FALSE .. TRUE;

 System_POSIX_Version    : constant := 1990_09;
 POSIX_Ada_Version       : constant := 1992_06;

  -- I/O Count

 type IO_Count is new INTEGER
        range 0--INTEGER'last;
 subtype IO_Count_Maxima is IO_Count range 32767..IO_Count' Last;

   -- System Limits

 Portable_Groups_Maximum            : constant Natural  := 0;
 subtype Groups_Maxima is Natural
         range 0 .. Natural'Last;

 Portable_Argument_List_Maximum     : constant Natural := 4096;
 subtype Argument_List_Maxima is Natural
         range 4_096 .. Natural'Last;

 Portable_Child_Processes_Maximum   : constant Natural := 6;
 subtype Child_Processes_Maxima is Natural
        range 6 .. Natural'Last;

 Portable_Open_Files_Maximum        : constant Natural := 16;
 subtype Open_Files_Maxima is Natural
        range 16 .. Natural'Last;
```

```
(Continued)
 Portable_Stream_Maximum            : constant Natural  := 8;
    subtype Stream_Maxima is Natural
            range 8 .. Natural'last;

 Portable_Time_Zone_String_Maximum  : constant Natural  := 3;
 subtype Time_Zone_String_Maxima is Natural
            range 3 .. Natural'last;

   -- Pathname Variable Values
 Portable_Link_Limit_Maximum        : constant Natural  := 8;
 subtype Link_Limit_Maxima is Natural
            range 8 .. Natural'Last;

 Portable_Input_Line_Limit_Maximum  : constant IO_Count := 255;
 subtype Input_Line_Limit_Maxima is IO_Count
            range 255 .. IO_Count'Last;

 Portable_Input_Queue_Limit_Maximum : constant IO_Count := 255;
 subtype Input_Queue_Limit_Maxima is IO_Count
            range 255 .. IO_Count'Last;

 Portable_Filename_Limit_Maximum    : constant Natural  := 14;
 subtype Filename_Limit_Maxima is Natural
            range 14 .. Natural'Last;

 Portable_Pathname_Limit_Maximum    : constant Natural := 255;
 subtype Pathname_Limit_Maxima is Natural
            range 255 .. Natural’Last;

 Portable_Pipe_Limit_Maximum        : constant IO_Count := 512;
 subtype Pipe_Limit_Maxima is IO_Count
            range 512 .. IO_Count’Last;

   -- Blocking Behavior Values

 type Blocking_Behavior is (Tasks, Program);
 subtype Text_IO_Blocking_Behavior is Blocking_Behavior
            range Program .. Program;

 IO_Blocking_Behavior               : constant Blocking_Behavior
                                    := Program;
```

```
(Continued)
 File_Lock_Blocking_Behavior       : constant Blocking_Behavior
                                   := Program;
 Wait_For_Child_Blocking_Behavior  : constant Blocking_Behavior
                                   := Program;

 -- Signal Masking

 type Signal_Masking is (No_Signals, RTS_Signals, All_Signals);

 -- Characters and Strings

 type POSIX_Character is
 new Standard.Character;   -- really should include hi-bit chars!
    for POSIX_Character'size use 8;
  --      (
  --    -- ' ','0','1','2','3','4','5','6','7','8','9',
  --    -- 'A','B','C','D','E','F','G','H','I','J','K','L','M',
  --    -- 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
  --    -- 'a','b','c','d','e','f','g','h','i','j','k','l','m',
  --    -- 'n','o','p','q','r','s','t','u','v','w','x','y','z',
  --    -- '.','_','-','/','"', '#', '&', ''', '(', ')',
  --    -- '*', '+', ',', ':', ';', '<', '=', '>', '|'
  --       <other characters are implementation defined>);

type POSIX_String is array (Positive range <>) of POSIX_Character;

function To_POSIX_String     (Str : string)
    return POSIX_String;
function To_String           (Str : POSIX_String)
    return string;

subtype Filename is POSIX_String;
subtype Pathname is POSIX_String;

function Is_Filename          (Str : POSIX_String)
    return Boolean;
function Is_Pathname          (Str : POSIX_String)
    return Boolean;

function Is_Portable_Filename (Str : POSIX_String)
    return Boolean;
```

```
  (Continued)
function Is_Portable_Pathname (Str : POSIX_String)
   return Boolean;

-- String Lists

type POSIX_String_List is limited private;

Empty_String_List : constant POSIX_String_List;

procedure Make_Empty (List     : in out POSIX_String_List);

procedure Append (List         : in out POSIX_String_List;
                  Str          : in     POSIX_String);
generic
   with procedure Action
                 (Item         : in POSIX_String;
                  Quit         : in out Boolean);
procedure For_Every_Item (List : in  POSIX_String_List);

function Length (List          : POSIX_String_List)
    return Natural;

function Value
        (List                  : POSIX_String_List;
         Index                 : Positive)
    return POSIX_String;

-- option sets

type Option_Set is private;

function Empty_Set
   return Option_Set;

function "+" (L, R             : Option_Set)
   return Option_Set;
function "-" (L, R             : Option_Set)
   return Option_Set;

-- Exceptions and error codes

POSIX_Error                   : exception;
```

```
  (Continued)
type Error_Code is new Integer;

function Get_Error_Code
   return Error_Code;

procedure Set_Error_Code (Error: in   Error_Code);

function Is_POSIX_Error (Error: Error_Code)
   return Boolean;
function Image (Error: Error_Code)
   return String;

No_Error                    : constant Error_Code
                            :=  0;  -- EOK
Argument_List_Too_Long      : constant Error_Code
                            :=  7;  -- E2BIG
Bad_File_Descriptor         : constant Error_Code
                            :=  9;  -- EBADF
Broken_Pipe                 : constant Error_Code
                            :=  32; -- EPIPE
Directory_Not_Empty         : constant Error_Code
                            :=  93; -- ENOTEMPTY
Exec_Format_Error           : constant Error_Code
                            :=  8;  -- ENOEXEC
File_Exists                 : constant Error_Code
                            :=  17; -- EEXIST
File_Too_Large              : constant Error_Code
                            :=  27; -- EFBIG
Filename_Too_Long           : constant Error_Code
                            :=  78; -- ENAMETOOLONG
Improper_Link               : constant Error_Code
                            :=  18; -- EXDEV
Inappropriate_IO_Control_Operation : constant Error_Code
                            :=  25; -- ENOTTY
Input_Output_Error          : constant Error_Code
                            :=  5;  -- EIO
Interrupted_Operation       : constant Error_Code
                            :=  4;  -- EINTR
Invalid_Argument            : constant Error_Code
                            :=  22; -- EINVAL
Invalid_Seek                : constant Error_Code
                            :=  29; -- ESPIPE
```

```
(Continued)
Is_A_Directory                : constant Error_Code
                              :=  21; -- EISDIR
No_Child_Process              : constant Error_Code
                              :=  10; -- ECHILD
No_Locks_Available            : constant Error_Code
                              :=  46; -- ENOLCK
No_Space_Left_On_Device       : constant Error_Code
                              :=  28; -- ENOSPC
No_Such_Operation_On_Device   : constant Error_Code
                              :=  19;  -- ENODEV
No_Such_Device_Or_Address     : constant Error_Code
                              :=  6; -- ENXIO
No_Such_File_Or_Directory     : constant Error_Code
                              :=  2;  -- ENOENT
No_Such_Process               : constant Error_Code
                              :=  3;  -- ESRCH
Not_A_Directory               : constant Error_Code
                              :=  20; -- ENOTDIR
Not_Enough_Space              : constant Error_Code
                              :=  12; -- ENOMEM
Operation_Not_Implemented     : constant Error_Code
                              :=  89; -- ENOSYS
Operation_Not_Permitted       : constant Error_Code
                              :=  1;  -- EPERM
Permission_Denied             : constant Error_Code
                              :=  13; -- EACCES
Read_Only_File_System         : constant Error_Code
                              :=  30; -- EROFS
Resource_Busy                 : constant Error_Code
                              :=  16; -- EBUSY
Resource_Deadlock_Avoided     : constant Error_Code
                              :=  45; -- EDEADLK
Resource_Temporarily_Unavailable : constant Error_Code
                              :=  11; -- EAGAIN
Too_Many_Links                : constant Error_Code
                              :=  31; -- EMLINK
Too_Many_Open_Files           : constant Error_Code
                              :=  24; -- EMFILE
Too_Many_Open_Files_In_System : constant Error_Code
                              :=  23; -- ENFILE
```

```
(Continued)
 -- System Identification
    function System_Name
        return POSIX_String;

    function Node_Name
        return POSIX_String;

    function Release
        return POSIX_String;

    function Version
        return POSIX_String;

    function Machine
        return POSIX_String;

  private
    type string_ptr is access POSIX_String;
    type list_elem is
      record
        next               : POSIX_String_List;
        current            : string_ptr;
      end record;

    type POSIX_String_List is access list_elem;
    Empty_String_List      : constant POSIX_String_List
                           := null;

    type opt_set_array is array(0..31) of boolean;
    for  opt_set_array'size use integer'size;
    type Option_Set is
    record
       opt_set    : opt_set_array := (others => false);
    end record;

end POSIX;
```

Optional Facilities

The user can call the function in POSIX_CONFIGURABLE_FILE_LIMITS or POSIX_CONFIGURABLE_SYSTEM_LIMITS to determine if the option is supported. (See “package POSIX” on page C-5 for subtype ranges.)

System Limits

The user can call the function in POSIX_CONFIGURABLE_FILE_LIMITS or POSIX_CONFIGURABLE_SYSTEM_LIMITS to determine a limit or the existence of a limit. (See “package POSIX” on page C-5 for ranges and constants.)

Implementation Requirements

This implementation allows programs with

1. At least 0 (PORTABLE_GROUPS_MAXIMUM) simultaneous supplementary group IDs.
2. At least 4096 (PORTABLE_ARGUMENT_LIST_MAXIMUM) POSIX_CHARACTERS as the length of the argument list, environment data and their overhead.
3. At least 6 (PORTABLE_CHILD_PROCESS_MAXIMUM) as the number of simultaneous processes per real user ID.
4. At least 16 (PORTABLE_OPEN_FILES_MAXIMUM) as the number of simultaneously open files per process.
5. At least 8 (PORTABLE_LINK_LIMIT_MAXIMUM) as the value of the link count of a file.
6. At least 255 (PORTABLE_INPUT_LINE_LIMIT_MAXIMUM) POSIX_CHARACTERS as the length of an input line.
7. At least 255 (PORTABLE_INPUT_QUEUE_LIMIT_MAXIMUM) POSIX_CHARACTERS as the length of a terminal input queue.
8. At least 14 (PORTABLE_FILENAME_LIMIT_MAXIMUM) POSIX_CHARACTERS as the length of a filename.
9. At least 255 (PORTABLE_PATHNAME_LIMIT_MAXIMUM) POSIX_CHARACTERS as the length of a pathname.
10. At least 512 (PORTABLE_PIPE_LIMIT_MAXIMUM) POSIX_CHARACTERS as the size of an atomic write to a pipe.
11. At least 3 (PORTABLE_TIME_ZONE_STRING_MAXIMUM) POSIX_CHARACTERS as the size of the TZ environment variable.
12. At least 8 (PORTABLE_STREAM_MAXIMUM) open C-language streams.

C.2.3.1 Error Codes and Exceptions

No additional error codes have been defined by this implementation. (See “package POSIX” on page C-5 for ERROR_CODE constants).

Description

If ERROR is the value of one of the error codes defined by POSIX, the value returned by Image is the identifier of the corresponding constant, in uppercase. Otherwise, ERROR_CODE' image is returned.

C.2.3.2 System Identification

See “package POSIX” on page C-5 for the actual values.

C.3 Process Primitives

C.3.1 package POSIX_PROCESS_PRIMITIVES

C.3.1.1 Process Creation

The search is conducted as if the PATH variable has the value of :.:/bin/:/usr/bin as the default, if the environment variable PATH is not present when START_PROCESS_SEARCH is called.

If START_PROCESS or START_PROCESS_SEARCH fail, but were able to locate the new process image file, refer to the C Conformance Document for expected behavior.

C.3.2 package POSIX_UNSAFE_PROCESS_PRIMITIVES

C.3.2.1 File Execution

If the environment variable for pathname has not been defined, START_PROCESS_SEARCH uses the current working directory, /bin and /usr/bin as the default.

C.3.3 package POSIX_SIGNALS

```
with POSIX,
     POSIX_Process_Identification;
with System;
with os_decl;
package POSIX_Signals is

   -- Signal Type
   type Signal is new integer;

   function Image(Sig  : Signal)
      return String;
   function Value(Str  : String)
      return Signal;

   -- Standard Signals (required)

   Signal_Null, SIGNULL          : constant Signal
                              :=   0;
   Signal_Abort, SIGABRT         : constant Signal
                              :=   6;
   Signal_Alarm, SIGALRM         : constant Signal
                              :=   14;
   Signal_Floating_Point_Error,
   SIGFPE                        : constant Signal
                              :=   8;
   Signal_Hangup, SIGHUP         : constant Signal
                              :=   1;
   Signal_Illegal_Instruction,
   SIGILL                        : constant Signal
                              :=   4;
   Signal_Interrupt, SIGINT      : constant Signal
                              :=   2;
   Signal_Kill, SIGKILL          : constant Signal
                              :=   9;
   Signal_Pipe_Write, SIGPIPE    : constant Signal
                              :=   13;
   Signal_Quit, SIGQUIT          : constant Signal
                              :=   3;
   Signal_Segmentation_Violation,
   SIGSEGV                       : constant Signal
                              :=   11
```

```
(Continued)
   Signal_Terminate, SIGTERM     : constant Signal
                              :=   15;
   Signal_User_1, SIGUSR1        : constant Signal
                              :=   16;
   Signal_User_2, SIGUSR2        : constant Signal
                              :=   17;

   -- Standard Signals (job control)

   Signal_Child, SIGCHLD         : constant Signal
                              :=   18;
   Signal_Continue, SIGCONT      : constant Signal
                              :=   25;
   Signal_Stop, SIGSTOP          : constant Signal
                              :=   23;
   Signal_Terminal_Stop, SIGTSTP : constant Signal
                              :=   24;
   Signal_Terminal_Input,
   SIGTTIN                       : constant Signal
                              :=   26;
   Signal_Terminal_Output,
   SIGTTOU                       : constant Signal
                              :=   27;

   -- Signal Handler References

   Signal_Abort_Ref              : constant System.Address
                              := System.address'ref(SIGABRT);
     -- Signal_Alarm intentionally omitted.
     -- Signal_Floating_Point_Error intentionally omitted.
   Signal_Hangup_Ref             : constant System.Address
                              := System.address'ref(SIGHUP);
     -- Signal_Illegal_Instruction intentionally omitted.
   Signal_Interrupt_Ref          : constant System.Address
                              := System.address'ref(SIGINT);
     -- Signal_Kill intentionally omitted.
   Signal_Pipe_Write_Ref         : constant System.Address
                             :=  System.address'ref(SIGPIPE);
   Signal_Quit_Ref               : constant System.Address
                              := System.address'ref(SIGQUIT);
```

```
   (Continued)
     -- Signal_Segmentation_Violation intentionally omitted.
     Signal_Terminate_Ref        : constant System.Address
                                 := System.address'ref(SIGTERM);
     Signal_User_1_Ref           : constant System.Address
                                 := System.address'ref(SIGUSR1);
     Signal_User_2_Ref           : constant System.Address
                                 := System.address'ref(SIGUSR2);
     Signal_Child_Ref            : constant System.Address
                                 := System.address'ref(SIGCHLD);
     Signal_Continue_Ref         : constant System.Address
                                 := System.address'ref(SIGCONT);
     -- Signal_Stop intentionally omitted.
     Signal_Terminal_Stop_Ref    : constant System.Address
                                 := System.address'ref(SIGTSTP);
     Signal_Terminal_Input_Ref   : constant System.Address
                                 := System.address'ref(SIGTTIN);
     Signal_Terminal_Output_Ref  : constant System.Address
                                 := System.address'ref(SIGTTOU);

     -- Signal Sets

     type Signal_Set is private;

     procedure Add_Signal
            (Set : in out Signal_Set;
             Sig : in     Signal);

     procedure Add_All_Signals (Set : in out Signal_Set);

     procedure Delete_Signal
             (Set : in out Signal_Set;
              Sig : in     Signal);

     procedure Delete_All_Signals (Set : in out Signal_Set);

     function Is_Member
             (Set : Signal_Set;
              Sig : Signal)
          return Boolean;

-- Sending a Signal
```

```
(Continued)
 procedure Send_Signal
      (Process : in POSIX_Process_Identification.Process_ID;
       Sig     : in   Signal);

  procedure Send_Signal
     (Group   : in POSIX_Process_Identification.Process_Group_ID;
       Sig     : in Signal);

  procedure Send_Signal (Sig  : in  Signal);

  -- Blocking and Unblocking Signals

  procedure Set_Blocked_Signals
          (New_Mask  : in   Signal_Set;
          Old_Mask   :   out Signal_Set);

  procedure Block_Signals
          (Mask_to_Add  :   in     Signal_Set;
          Old_Mask      :    out Signal_Set);

  procedure Unblock_Signals
          (Mask_to_Subtract : in     Signal_Set;
           Old_Mask         :    out Signal_Set);

  function Blocked_Signals
     return Signal_Set;

  -- Ignoring Signals

  procedure Ignore_Signal   (Sig : in   Signal;

  procedure Unignore_Signal (Sig : in   Signal);

  function Is_Ignored (Sig :  Signal)
     return Boolean;

  -- Controlling Delivery of Signal_Child Signal

  procedure Set_Stopped_Child Signal
         (Enable : in   Boolean := True);
```

```
  function Stopped_Child_Signal_Enabled
     return Boolean;

  -- Examining Pending Signals

  function Pending_Signals
     return Signal_Set;

private
    type Signal_Set is new os_decl.sigset_rec;
end POSIX_Signals;
```

C.3.3.1 Signal Type

Mapping signals to values of `type SIGNAL` is a 1-to-1 mapping of the POSIX short signal name to the matching signal name in `/usr/include/errno.h`.

No other signals have been defined in this implementation.

If `SIG` is the value of one of the signals defined by POSIX.5, the value returned by `IMAGE` is the identifier of the corresponding long-name constant, in uppercase. Otherwise, the value returned is the `INTEGER'image` value of the signal number.

C.3.3.2 Standard Signals

See “package POSIX_SIGNALS” on page C-14 for signal values.

C.4 Process Environment

C.4.1 package POSIX_PROCESS_IDENTIFICATION

C.4.1.1 Process Identification Operations

Description

`type PROCESS_ID` defines the values for process IDs. `NULL_PROCESS_ID` never represents any process in the system. `SYSTEM_PROCESS_ID` is reserved by the system for system processes.

```
Null_Process_ID   : constant Process_ID := 0;
System_Process_ID : constant Process_ID := 1;
```

`IMAGE` returns `PROCESS_ID'image`.

`VALUE` translates any string into a `PROCESS_ID` as long as only the characters 0..9 of `type STANDARD.CHARACTER` are used. Otherwise a `CONSTRAINT_ERROR` is raised.

Error Handling

No exceptions are raised for `GET_PROCESS_ID` and `GET_PARENT_PROCESS_ID`.

C.4.1.2 Process Group Identification

Description

`IMAGE` returns `PROCESS_GROUP_ID'image`.

`VALUE` translates any string into a `PROCESS_GROUP_ID` as long as only the characters 0..9 of `type STANDARD.CHARACTER` are used. Otherwise a `CONSTRAINT_ERROR` is raised.

Error Handling

No exceptions are raised by `GET_PROCESS_GROUP_ID`.

C.4.1.3 User Identification

Description

IMAGE returns USER_ID'image.

VALUE translates any string into a USER_ID as long as only the characters 0..9 of type STANDARD.CHARACTER are used. Otherwise a CONSTRAINT_ERROR is raised.

Error Handling

No exceptions are raised by GET_REAL_USER_ID, GET_EFFECTIVE_USER_ID, or GET_LOGIN_NAME.

C.4.1.4 User and Group Identification

Description

The effective group ID of the calling process is included in the returned list of supplementary group IDs from a call to GET_GROUPS.

IMAGE returns GROUP_ID'image.

VALUE translates any PROCESS_GROUP_ID that only uses the characters 0..9 of type STANDARD.CHARACTER. Any other string raises a CONSTRAINT_ERROR.

Error Handling

No exceptions are raised by GET_REAL_GROUP_ID, GET_EFFECTIVE_GROUP_ID, or GET_GROUPS.

C.4.2 package POSIX_PROCESS_TIMES

C.4.2.1 Process Time Accounting

```
TICKS_PER_SECOND : Constant := 100;
```

Error Handling

No exceptions are raised for `package POSIX_PROCESS_TIMES`.

C.4.3 package POSIX_PROCESS_ENVIRONMENT

C.4.3.1 Environment Variables

Description

If `POSIX_PROCESS_ENVIRONMENT` is provided with an environment with multiple definitions of the same variable, `DELETE_ENVIRONMENT_VARIABLE` deletes all occurrences of the variable. `SET_ENVIRONMENT_VARIABLE` removes the multiple occurrences before entering the new value.

In a multitasking program, the effect of one task calling an operation that modifies an environment while another task is performing an operation on the same environment is that a semaphore is placed around the environment by the first task to access the structure. The other task may not access the environment until the semaphore is removed.

Error Handling

No exceptions are raised for `COPY_FROM_CURRENT_ENVIRONMENT`, `COPY_TO_CURRENT_ENVIRONMENT`, `COPY_ENVIRONMENT`, `CLEAN_ENVIRONMENT`, `LENGTH`, `FOR_EVERY_ENVIRONMENT_VARIABLE`, or `FOR_EVERY_CURRENT_ENVIRONMENT_VARIABLE`.

C.4.4 package POSIX_CALENDAR

C.4.4.1 Obtaining Time Information from the System

Error Handling

No exceptions are raised by `TO_TIME` or `TO_POSIX_TIME`.

C.5 Files and Directories

C.5.1 package POSIX_PERMISSIONS

C.5.1.1 The Permission Set

Description

No other permissions other than those in `ACCESS_PERMISSION_SET` are defined.

C.5.2 package POSIX_FILES

C.5.2.1 Creating and Removing files

Description

Soft links to files across file systems are supported but hard links are not. Soft links to directories are supported. Hard links to directories can be created only by processes with `USER_ID` of 0, that is, by root.

`CREATE_DIRECTORY` and `CREATE_FIFO` ignore any permissions not in `ACCESS_PERMISSION_SET`.

If the directory indicated in a call to `REMOVE_DIRECTORY` is the root directory for the current process or for the system, `REMOVE_DIRECTORY` raises `POSIX_ERROR` and sets the error code to `IS_A_DIRECTORY`.

Error Handling

If the directory named by the pathname is not empty when using `REMOVE_DIRECTORY`, `POSIX_ERROR` is raised and the error code is set to `DIRECTORY_NOT_EMPTY`.

C.5.2.2 Directory Iteration

FOR_EVERY_DIRECTORY_ENTRY accesses the entries in alphabetical order. If an entry is added to the directory referenced by the pathname during execution of the instance of FOR_EVERY_DIRECTORY_ENTRY, ACTION is not called for that entry. If an entry is removed from the directory referenced by the pathname, ACTION is still called with that entry.

C.5.2.3 Updating File Status Information

Description

On a call to CHANGE_OWNER_AND_GROUP, if the effective ID of the executing process is not a member of the file's group, SET_USER_ID and SET_GROUP_ID of the file mode are cleared, unless the effective user ID of the process is 0 (the super-user).

C.5.3 package POSIX_FILE_STATUS

C.5.3.1 Access Status Information

Description

For non-regular file types, SIZE_OF returns a size of 0.

C.5.4 package POSIX_CONFIGURABLE_FILE_LIMITS

C.5.4.1 File Limits

Description

If FILENAME_LIMIT, PATHNAME_LIMIT or PIPE_LENGTH_LIMIT is called with a file or pathname that is not a directory, the value returned is as if the procedure was called with the directory that file or pathname live in. FILENAME_IS_LIMITED, PATHNAME_IS_LIMITED and PIPE_LENGTH_IS_LIMITED act in the same manner.

C.6 Input and Output Primitives

C.6.1 package POSIX_IO

C.6.1.1 OPEN, OPEN_OR_CREATE, IS_OPEN, CLOSE, DUPLICATE, CREATE_PIPE

Description

The option Exclusive specifies that `OPEN_OR_CREATE` fails if the file named by the `NAME` parameter exists. This option has no effect on `OPEN`.

The option Truncate denotes whether the file is truncated when opened. The effect of opening a file with `Mode=>Read_Only` and the Truncate option set is that the file is truncated to length zero and the mode and owner remain unchanged.

C.6.1.2 Read, Write

Description

The effect of instantiating `GENERIC_READ` or `GENERIC_WRITE` if the external file is a pipe or FIFO and the size of the element is greater than `POSIX_CONFIGURABLE_FILE_LIMITS.PIPE_LENGTH` is that `ITEM'size` elements are read / written.

C.6.1.3 Seek

Description

`SEEK`, `FILE_POSITION`, and `FILE_SIZE` operations on devices that are incapable of seeking have no effect.

C.6.1.4 File Control

Description

No values other than `APPEND` and `NON-BLOCKING` are returned in the Options parameter of `GET_FILE_CONTROL`.

C.7 Device- and Class-Specific Functions

C.7.1 General Terminal Interface

Package not implemented.

C.7.2 package POSIX_TERMINAL_FUNCTIONS

Package not implemented.

C.8 Language Specific Services for the C Programming Language

C.8.1 Interoperable Ada I/O Services

Package not implemented.

C.8.2 package POSIX_SUPPLEMENT_TO_ADA_IO

Package not implemented.

C.9 System Databases

C.9.1 package POSIX_USER_DATABASE

Package not implemented.

C.9.2 package POSIX_GROUP_DATABASE

Package not implemented

C

"It has long been an axiom of mine that the little things are infinitely the most important."

Conan Doyle

Implementation-Dependent Characteristics F

This document summarizes the tools specific to this implementation of Ada. General information applying to all Ada implementations is presented first, followed by information specific to this implementation. The material required by Appendix F of the *Ada Reference Manual* is covered.

SPARCompiler Ada provides the full Ada language as specified in the *Ada Reference Manual*. In the *Ada Reference Manual*, a number of sections contain the annotation *implementation dependent*, meaning that the interpretation is left to the compiler implementor.

The Ada compiler provides these features:

- Shared-generic bodies
- All-Ada runtime system
- Representation clauses to the bit level and `pragma PACK` (section 13.1 in the *Ada Reference Manual*)
- Length clauses and unsigned types (8- and 16-bit) (section 13.2 in the *Ada Reference Manual*)
- Enumeration representation clauses (section 13.3 in the *Ada Reference Manual*)
- Record representation clauses (section 13.4 in the *Ada Reference Manual*)
- Interrupt entries (section 13.5.1 in the *Ada Reference Manual*)
- Representation attributes (section 13.7.2 in the *Ada Reference Manual*)

- Machine code insertions and `pragma IMPLICIT_CODE` (section 13.8 in the *Ada Reference Manual*)
- Interface programming features, including `pragma INTERFACE`, `pragma EXTERNAL_NAME`, `pragma INTERFACE_NAME`, `WITH`*n* directives, `a.info` and external dependencies capabilities (section 13.9 in the *Ada Reference Manual*)
- Unchecked deallocations (section 13.10.1 in the *Ada Reference Manual*)
- Unchecked conversions (section 13.10.2 in the *Ada Reference Manual*)
- Pool-based memory allocation option

F.1 Pragmas and Their Effects

Each pragma of this implementation is described briefly; more information on some of them is in the discussions of particular language constructs.

`pragma BIT_PACK`
Indicates to the compiler that packing down to the bit level is desired. `pragma BIT_PACK` can be used interchangeably with `pragma PACK` and `pragma BYTE_PACK`.

`pragma BUILT_IN`
Used in some parts of the code for `TEXT_IO`, `MACHINE_CODE`, `UNCHECKED_CONVERSION`, `UNCHECKED_DEALLOCATION`, and lower-level support packages in `standard`. It is reserved and cannot be accessed directly.

`pragma BYTE_PACK`
Indicates to the compiler that packing down to the byte level is desired. Components at least as large as, or larger than, a byte, are packed at byte boundaries. `pragma BYTE_PACK` can be used interchangeably with `pragma PACK` and `pragma BIT_PACK`.

`pragma CONTROLLED`
Recognized by the implementation but has no effect in the current release.

`pragma ELABORATE`
Implemented as described in Appendix B of the *Ada Reference Manual*.

`pragma EXTERNAL(`*language, subprogram*)
Supports calling Ada subprograms from foreign languages. The compiler generates code for the subprogram that is compatible with the calling conventions of the foreign language. Call the subprogram from Ada normally. The supported languages and restrictions on parameter and result types are the same as `pragma INTERFACE`. This pragma has an effect only if the calling conventions of the foreign language differ from those of Ada.

References

“pragma EXTERNAL and pragma EXTERNAL_NAME” on page 5-22
Section F.9, "Parameter Passing," on page F-29

`pragma EXTERNAL_NAME` (*subprogram, link_name*)
Allows you to specify a *link* for an Ada variable or subprogram so the object can be referenced from other languages. This pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

Objects must be variables defined in a package specification; subprograms are either library level or in a package specification.

References

“pragma EXTERNAL and pragma EXTERNAL_NAME” on page 5-22

`pragma IMPLICIT_CODE`
Specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF). Only use this pragma in the declarative part of a machine code procedure. Implicit code includes preamble and postamble code, for example, code that moves parameters to and from the stack. Use of `pragma IMPLICIT_CODE` does not eliminate code generated for runtime checks, nor call/return instructions (Eliminate them with `pragma SUPPRESS` and `pragma INLINE`, respectively). A warning issues if OFF is used and implicit code must be generated. Use this pragma with caution.

`pragma INITIALIZE(STATIC|DYNAMIC)`
When placed in a library-level package, spec, or body, initializes all objects in the package as indicated, statically or dynamically. Only library-level objects are subject to static initialization. By definition, all objects in procedures are dynamic.

If pragma INITIALIZE(STATIC) is used and an object cannot be initialized statically, code is generated to initialize the object, and a warning message generates.

pragma INLINE
Implemented as described in Appendix B of the *Ada Reference Manual*, with the addition that recursive calls can be expanded with the pragma up to the maximum depth of 4. Warnings are produced for bodies that are not available for inline expansion. pragma INLINE is ignored and a warning issues when it is applied to subprograms that declare tasks, packages, exceptions, types or nested subprograms.

pragma INLINE_ONLY
When used in the same way as pragma INLINE, indicates to the compiler that the subprogram must *always* be inlined (very important for some code procedures). This pragma suppresses the generation of a callable version of the routine, which saves code space. If you erroneously make an INLINE_ONLY subprogram recursive, a warning message emits and a PROGRAM_ERROR raises at runtime.

pragma INTERFACE (*language, subprogram*)
Supports calls to Ada, C, PASCAL, FORTRAN, and UNCHECKED language functions. The Ada specifications are either functions or procedures. Use pragma INTERFACE to call code written in unspecified languages using UNCHECKED for the language name.

For Ada, the compiler generates the call as if it is to an Ada procedure but does not expect a matching procedure body.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS. Pass record and array objects by reference using the 'ADDRESS attribute. All parameters must have mode IN.

For PASCAL, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS. Pass record and array objects by reference using the ADDRESS attribute.

For FORTRAN, all parameters are passed by reference; the parameter types must have type SYSTEM.ADDRESS. The result type for a FORTRAN function must be a scalar type.

Use UNCHECKED to interface to an unspecified language, such as assembler. The compiler generates the call as if it is to an Ada procedure, but it does not expect a matching Ada procedure body.

References

Section F.9, "Parameter Passing" on page F-29

pragma INTERFACE_NAME (*Ada_name*, *link_name*)

Allows variables or subprograms defined in another language to be referenced directly in Ada. It replaces all occurrences of *Ada_name* with an external reference to *link_name* in the object file.

If *Ada_name* denotes an object, the pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. Declare the object as a scalar or an access type. The object *cannot* be any of the following.

- Loop variable
- Constant
- Initialized variable
- Array
- Record

If *Ada_name* denotes a subprogram, a pragma INTERFACE must be specified already for the subprogram.

Construct the *link_name* as expected by the linker. For example, some C compilers and linkers preface the C variable name with an underscore. Such conventions are defined in package LANGUAGE. The following example makes the C global variable errno available in an Ada program:

```
with LANGUAGE;
package PACKAGE_NAME is
   ...
   ERRNO:INTEGER;
   pragma INTERFACE_NAME (ERRNO,LANGUAGE.C_PREFIX & “errno”);
   ...
end PACKAGE_NAME;
```

`pragma LINK_WITH` (*constant string expression*)

Passes arguments to the target linker. It can appear in any declarative part and accepts one argument, a constant string expression. This argument passes to the target linker if a link includes the unit containing the pragma.

For example, the following package puts the `-lm` option on the command line for the linker when the linked program includes `MATH`:

```
package MATH is
   pragma LINK_WITH("-lm");
end;
```

Or the following package links with the named object file `sin.o`:

```
package MATH is
-------------------------------------------------------------
--      SIN is a routine written in C or assembly: the object
--      for the routine is in the object file sin.o
-------------------------------------------------------------
   function SIN (X:FLOAT)          return FLOAT;
   pragma interface (C, SIN);
   pragma LINK_WITH("sin.o");
end MATH;
```

If the constant string expression begins with "`-`," the string is left untouched. However, if the string begins with neither "`-`" nor "`/`," the string is prefixed with "`./`."

`pragma LIST`

Implemented as described in Appendix B of the *Ada Reference Manual*.

`pragma MEMORY_SIZE`

Recognized by the implementation but has no effect in the current release. This implementation does not allow `package SYSTEM` to be modified by means of pragmas; it must be recompiled.

`pragma NO_IMAGE`
Suppresses the generation of the image array used for the `IMAGE` attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. An attempt to use the `IMAGE` attribute on a type whose image array is suppressed results in a compilation warning and `PROGRAM_ERROR` is raised at runtime.

`pragma NON_REENTRANT`
Takes one argument that is the name of a library subprogram or a subprogram declared immediately in a library package specification or body. This pragma indicates to the compiler that the subprogram is not called recursively, allowing the compiler to perform specific optimizations. Apply the pragma to a subprogram or a set of overloaded subprograms in a package specification or package body.

`pragma NOT_ELABORATED`
Suppresses the generation of elaboration code and issues warnings if elaboration code is required. It indicates that the package is not elaborated because it is either part of the RTS, a configuration package or an Ada package referenced from a language other than Ada. It can appear only in a library package specification.

`pragma OPTIMIZE`
Recognized by the implementation, but has no effect in the current release.

`pragma OPTIMIZE_CODE(OFF|ON)`
Specifies whether the code is optimized (ON) by the compiler or not (OFF). Use it in any subprogram. When OFF is specified, the compiler generates unoptimized code. The default is ON. If ON, the `-Ox` option to `ada` or `a.make` controls the level of optimization.

Suppress optimization selectively with this pragma at the subprogram level. Inline subprograms optimize if they have `pragma OPTIMIZE_CODE(OFF)` unless the caller has `pragma OPTIMIZE_CODE(OFF)`.

References

code optimization levels (`ada`), *SPARCompiler Ada Reference Guide*

`pragma PACK`
Causes the compiler to minimize gaps between components in the representation of composite types. Objects larger than a single `STORAGE_UNIT` are packed to the nearest `STORAGE_UNIT`. This pragma can be used interchangeably with `pragma BIT_PACK` and `pragma BYTE_PACK`.

`pragma PAGE`
Implemented as described in Appendix B of the *Ada Reference Manual*. The source code formatting tool, `a.pr`, recognizes it.

`pragma PASSIVE` has five forms:

```
pragma PASSIVE
pragma PASSIVE(ABORT_SAFE);
pragma PASSIVE(ABORT_UNSAFE);
pragma PASSIVE(ABORT_SAFE,  mutex attr'address);
pragma PASSIVE(ABORT_UNSAFE,  mutex attr'address);
```

Apply this pragma to a task or task type declared immediately in a library package specification or body. It directs the compiler to optimize certain tasking operations. Statements in the task body can prevent the intended optimization; in these cases a warning generates at compile time and `TASKING_ERROR` raises at runtime.

References

Passive Tasks, *SPARCompiler Ada Runtime System Guide*

`pragma PRIORITY`
Implemented as described in Appendix B of the *Ada Reference Manual*. The allowable range for `pragma PRIORITY` is 0 .. 99.

`pragma RTS_INTERFACE`(*RTS_routine, user_routine*)
Allows for the replacement of the default calls made implicitly at runtime to the underlying RTS routines. Causes the compiler to generate calls to any routine of your choosing as long as its parameters and `RETURN` value match the original. Use this pragma with caution.

`pragma SHARE_CODE`(*generic unit / instantiation, boolean*)
Provides for the sharing of object code between multiple instantiations of the same generic subprogram or package body. A "parent" instantiation is created and subsequent instantiations of the same types share the parent

object code, reducing program size and compilation times. In the runtime, pragma SHARE_CODE is used for the generic packages INTETGER_IO, FLOAT_IO, and ENUMERATION_IO.

pragma SHARE_CODE takes the name of a generic instantiation or a generic unit as the first argument and either one of the identifiers TRUE or FALSE as a second argument. When the first argument is a generic unit, the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation, the pragma applies only to the specified instantiation or overloaded instantiations.

If the second argument is TRUE, the compiler tries to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE, each instantiation gets a unique copy of the generated code.

The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit. It is only allowed immediately at the place of a declarative item in a declarative part or package specification or after a library unit in a compilation but before any subsequent compilation unit.

Use pragma SHARE_BODY instead of SHARE_CODE with the same effect.

pragma SHARED
Recognized by the implementation but has no effect in the current release.

pragma STORAGE_UNIT
Recognized by the implementation but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, achieve the same effect by recompiling package SYSTEM with altered values.

pragma SUPPRESS
Implemented as described in Appendix B of the *Ada Reference Manual*, except that DIVISION_CHECK and, in some cases, OVERFLOW_CHECK, cannot be suppressed.

The use of pragma SUPPRESS(ALL_CHECKS) is equivalent to writing at the same point in the program a suppress pragma for each of the checks listed in section 11.7 of the *Ada Reference Manual*.

pragma SUPPRESS(EXCEPTION_TABLES) informs the code generator that the tables normally generated for exception regions are not generated for the enclosing compilation unit. This reduces the size of the static data for a unit but disables exception handling in that unit.

pragma SYSTEM_NAME
Recognized by the implementation but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, copy the file system.a from the STANDARD library to a local Ada library and recompile it there with new values.

pragma TASK_ATTRIBUTES has two forms:

```
pragma TASK_ATTRIBUTES(task_attr'address);
pragma TASK_ATTRIBUTES(task_object, task_attr'address);
```

The first form is only allowed within the specification of a task unit. It specifies the task attributes of the task or tasks of the task type. The second form is applicable to any task object. It takes precedence over the task attributes specified for the task's type.

The address of an ADA_KRN_DEFS.TASK_ATTR_T record is the first or second argument of the pragma and is passed to the underlying microkernel at task creation.

The task attributes are microkernel dependent. See ada_krn_defs.a in standard for the type definition of TASK_ATTR_T and the different options supported. When there isn't a TASK_ATTRIBUTES pragma for a task, the DEFAULT_TASK_ATTRIBUTES found in v_usr_conf_b.a's configuration table are used.

All variations of the TASK_ATTR_T record contain at least the prio, mutex_attr_address and cond_attr_address fields. prio specifies the priority of the task. If the task also has a pragma PRIORITY(PRIO), the prio specified in the TASK_ATTR_T record takes precedence.

The mutex_attr_address field contains the address of the attributes to be used to initialize the mutex object implicitly created for the task. This mutex is used to protect the task's data structure. For example, the task's mutex is locked when another task attempts to rendezvous with it.

If mutex_attr_address is set to NO_ADDR, the mutex_attr_address value specified by the V_USR_CONF.CONFIGURATION_TABLE parameter, DEFAULT_TASK_ATTRIBUTES, is used. Otherwise, mutex_attr_address must

be set to the address of an ADA_KRN_DEFS.MUTEX_ATTR_T record. The MUTEX_ATTR_T record should be initialized using one of the ADA_KRN_DEFS mutex attribute init subprograms.

References

Mutex Support Subprograms, *SPARCompiler Ada Runtime System Guide*

The cond_attr_address field contains the address of the attributes to be used to initialize the condition variable object implicitly created for the task. When the task blocks, it waits on this condition variable. If cond_attr_address is set to NO_ADDR, then, the cond_attr_address value specified by the V_USR_CONF.CONFIGURATION_TABLE parameter, DEFAULT_TASK_ATTRIBUTES is used. Otherwise, cond_attr_address must be set to the address of a ADA_KRN_DEFS.COND_ATTR_T record. The COND_ATTR_T record should be initialized using one of the ADA_KRN_DEFS condition variable attribute init routines.

References

Condition Variable Support Subprograms, *SPARCompiler Ada Runtime System Guide*

ada_krn_defs.a has overloaded versions of the following subprogram for initializing the task attributes:

```
function task_attr_init(
         prio            : priority;
         .
         .  OS dependent fields
         .
         mutex_attr  : a_mutex_attr_t := null;
         cond_attr : a_cond_attr_t := null
) return address;
```

The first argument in the second form is the name of a task object. This allows task objects of the same task type to have different task attributes (including different task priorities).

References

Ada Kernel, *SPARCompiler Ada Runtime System Guide*

`pragma VOLATILE(`*object*`)`
Guarantees that loads and stores to the named object are performed as expected after optimization.

The object declaration and the pragma must both occur (in this order) immediately in the same declarative part or package specification.

```
pragma warnings (off);
    statement(s) that generate warnings;
pragma warnings (on);
```

F.2 Predefined Packages and Generics

The following predefined Ada packages given in Appendix C(22) of the *Ada Reference Manual* are provided in the `standard` and `cross_io` libraries. See Figure F-1.

- `generic function UNCHECKED_CONVERSION`
- `generic package DIRECT_IO`
- `generic package SEQUENTIAL_IO`
- `generic procedure UNCHECKED_DEALLOCATION`
- `package CALENDAR`
- `package IO_EXCEPTIONS`
- `package LOW_LEVEL_IO`
- `package MACHINE_CODE`
- `package STANDARD`
- `package SYSTEM`
- `package TEXT_IO`

F.2.1 Specification of package SYSTEM

```
with UNSIGNED_TYPES;
package SYSTEM is
   pragma LINK_WITH(“-Bstatic”);
   pragma SUPPRESS(ALL_CHECKS);
   pragma SUPPRESS(EXCEPTION_TABLES);
   pragma NOT_ELABORATED;
   type NAME is ( sun4_unix );
   SYSTEM_NAME     : constant NAME := sun4_unix;
   STORAGE_UNIT    : constant := 8;
   MEMORY_SIZE     : constant := 16_777_216

   -- System-Dependent Named Numbers

   MIN_INT         : constant := -2_147_483_648;
   MAX_INT         : constant := 2_147_483_647;
   MAX_DIGITS      : constant := 15;
   MAX_MANTISSA    : constant := 31;
   FINE_DELTA      : constant := 2.0**(-31);
   TICK            : constant := 0.01;
   -- Other System-dependent Declarations
   subtype PRIORITY is INTEGER range 0 .. 99;
   MAX_REC_SIZE : integer := 64*1024;
   type ADDRESS is private;
   function “>” (A: ADDRESS; B: ADDRESS) return BOOLEAN;
   function “<“ (A: ADDRESS; B: ADDRESS) return BOOLEAN;
   function “>=”(A: ADDRESS; B: ADDRESS) return BOOLEAN;
   function “<=”(A: ADDRESS; B: ADDRESS) return BOOLEAN;
   function “-” (A: ADDRESS; B: ADDRESS) return INTEGER;
   function “+” (A: ADDRESS; I: INTEGER) return ADDRESS;
   function “-” (A: ADDRESS; I: INTEGER) return ADDRESS;
   function “+” (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS;
   function MEMORY_ADDRESS
      (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS renames “+”;
   NO_ADDR : constant ADDRESS;
   type TASK_ID is private;
   NO_TASK_ID : constant TASK_ID;
   subtype SIG_STATUS_T is INTEGER;
   SIG_STATUS_SIZE: constant  :=4
   type PROGRAM_ID is private;
   NO_PROGRAM_ID : constant PROGRAM_ID;
```

(Continued)

```
        type LONG_ADDRESS is private;
        NO_LONG_ADDR : constant LONG_ADDRESS;
        function "+" (A: LONG_ADDRESS; I: INTEGER) return LONG_ADDRESS;
        function "-" (A: LONG_ADDRESS; I: INTEGER) return LONG_ADDRESS;
        function MAKE_LONG_ADDRESS (A: ADDRESS) return LONG_ADDRESS;
        function LOCALIZE (A: LONG_ADDRESS ; BYTE_SIZE : INTEGER) return ADDRESS;
        function STATION_OF(A: LONG_ADDRESS) return INTEGER;

        -- Constants describing the configuration of the CIFO add-on
        -- product Only valid for single processor Ada
        SUPPORTS_INVOCATION_BY_ADDRESS  : constant BOOLEAN := TRUE;
        SUPPORTS_PREELABORATION         : constant BOOLEAN := TRUE;
        MAKE_ACCESS_SUPPORTED           : constant BOOLEAN := TRUE;
private
   type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
   NO_ADDR : constant ADDRESS := 0;
   pragma BUILT_IN(“>”);
   pragma BUILT_IN(“<“);
   pragma BUILT_IN(“>=”);
   pragma BUILT_IN(“<=”);
   pragma BUILT_IN(“-”);
   pragma BUILT_IN(“+”);
   type TASK_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
   NO_TASK_ID : constant TASK_ID := 0;
   type PROGRAM_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
   NO_PROGRAM_ID : constant PROGRAM_ID := 0;
end SYSTEM;
```

Figure F-1 Example Specification of package SYSTEM

F.2.2 package CALENDAR

CALENDAR clock function (in package CALENDAR.LOCAL_TIME located in the file calendar_s.a) uses the Solaris 2.1 service routines GETTIMEOFDAY and LOCALTIME for getting the current time.

F.2.3 package MACHINE_CODE

package MACHINE_CODE provides an assembly-language interface for the target machine, including the record types needed in the code statement, an enumeration type containing all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. It supplies pragma IMPLICIT_CODE and the 'REF (only use in units that WITH MACHINE_CODE).

Machine code statements take operands of type OPERAND, a private type that forms the basis of all machine code address formats for the target.

The general syntax of a machine code statement is:

```
CODE_n'(opcode, operand [, operand]));
```

where *n* indicates the number of operands in the aggregate.

When a variable number of operands exist, list them in a subaggregate using this syntax:

```
CODE_n'(opcode, (operand [, operand]);
```

In the following example, code_2 is a record "format" whose first argument is an enumeration value of type OPCODE followed by two operands of type OPERAND:

```
CODE_2'(add, a'ref, b'ref);
```

For those opcodes requiring no operands, named notation must be used (see section 4.3(4) in the *Ada Reference Manual*).

```
CODE_0'(op => opcode);
```

The *opcode* must be an enumeration literal, that is, it cannot be an object, attribute or a rename. An *operand* is only an entity defined in MACHINE_CODE or 'REF.

'REF denotes the effective address of the first of the storage units allocated to the object. 'REF is not supported for a package, task unit, or entry.

Arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE.

As an example of machine code insertions, `procedure WRITE_Y_REGISTER` is defined in Figure F-2. It writes its argument in `%y` register of the SPARC processor.

```
procedure write_y_register(x: integer) is
begin
   code_3'(wr, g0, x'ref, y);
end write_y_register;
pragma inline(write_y_register);
procedure example is
   before_call, after_call: integer;
   value: integer := 2;
begin
   before_call := 1;  -- instruction before call
   write_y_register(value);
   after_call := 1;   -- instruction after call
end example;
```

Figure F-2 Example of Machine Code Insertions

Note that the machine code procedure is inline. Figure F-3, an excerpted `a.das` output, shows a procedure that calls `WRITE_Y_REGISTER` and the code generated for the call:

```
11       before_call := 1;  -- instruction before call;
      0010:  or      %g0, +01, %g2
7           code_3'(wr, g0, x'ref, y);
      0014:  wry     %g0, %g1, %y
13       after_call := 1;   -- instruction after call
      0018:  or      %g0, +01, %g3
```

Figure F-3 Example of Machine Code Insertions - Disassembled Output

References

Section F.41, "'REF" on page -20
opcode named notation, section 4.3(4) in *Ada Reference Manual*

F.2.4 package SEQUENTIAL_IO

Currently, sequential I/O is implemented for variant records with the restriction that the maximum size for the record is always written. This is true of direct I/O. For unconstrained records and arrays, set the constant, SYSTEM.MAX_REC_SIZE, prior to the elaboration of the generic instantiation of SEQUENTIAL_IO or DIRECT_IO. For example, if unconstrained strings are written, SYSTEM.MAX_REC_SIZE effectively restricts the maximum size of string that can be written. If you know the maximum size of such strings, set the SYSTEM.MAX_REC_SIZE prior to instantiating SEQUENTIAL_IO for the string type. Reset this variable after the instantiation with no effect.

F.2.5 package UNSIGNED_TYPES

package UNSIGNED_TYPES is supplied to illustrate the definition of and services for the unsigned types supplied in this version of Ada. We do not given any warranty, expressed or implied, for the effectiveness or legality of this package. Use it at your own risk.

We intend to withdraw this implementation if and when the AJPO and the Ada community reach agreement on a practical unsigned types specification. We can then standardize on that accepted version at a practical date thereafter.

The package is supplied in comment form because the actual package cannot be expressed in normal Ada — the types are not symmetric about 0 as required by the *Ada Reference Manual*. This package is supplied and is accessible through the Ada WITH statement as though it is present in source form.

Example:

```
with unsigned_types;
procedure foo( xxx: unsigned_types.unsigned_integer) is ...
```

Caution – Use package UNSIGNED_TYPES at your own risk.

F.2.6 Specification of package UNSIGNED_TYPES

See Figure F-4.

```
-- package unsigned_types is
--
-- type unsigned_integer is range 0 .. (2**32 - 1);    -- 0..4294967295
--    function “=” (a, b: unsigned_integer) return boolean;
--    function “/=”(a, b: unsigned_integer) return boolean;
--    function “<” (a, b: unsigned_integer) return boolean;
--    function “<=”(a, b: unsigned_integer) return boolean;
--    function “>” (a, b: unsigned_integer) return boolean;
--    function “>=”(a, b: unsigned_integer) return boolean;
--    function “+” (a, b: unsigned_integer) return unsigned_integer;
--    function “-” (a, b: unsigned_integer) return unsigned_integer;
--    function “+” (a   : unsigned_integer) return unsigned_integer;
--    function “-” (a   : unsigned_integer) return unsigned_integer;
--    function “*” (a, b: unsigned_integer) return unsigned_integer;
--    function “/” (a, b: unsigned_integer) return unsigned_integer;
--    function “mod”(a, b: unsigned_integer) return unsigned_integer;
--    function “rem”(a, b: unsigned_integer) return unsigned_integer;
--    function “**” (a, b: unsigned_integer) return unsigned_integer;
--    function “abs”(a, b: unsigned_integer) return unsigned_integer;
--
-- type unsigned_short_integer is range 0 ..(2**16 - 1);   -- 0..65535
--    function “=” (a, b: unsigned_short_integer) return boolean;
--    function “/=”(a, b: unsigned_short_integer) return boolean;
--    function “<” (a, b: unsigned_short_integer) return boolean;
--    function “<=”(a, b: unsigned_short_integer) return boolean;
--    function “>” (a, b: unsigned_short_integer) return boolean;
--    function “>=”(a, b: unsigned_short_integer) return boolean;
--    function “+” (a, b: unsigned_short_integer)
--                  return unsigned_short_integer;
--    function “-” (a, b: unsigned_short_integer)
--                  return unsigned_short_integer;
--    function “+” (a   : unsigned_short_integer)
--                  return unsigned_short_integer;
--    function “-” (a   : unsigned_short_integer)
--                  return unsigned_short_integer;
--    function “*” (a, b: unsigned_short_integer)
--                  return unsigned_short_integer;
--    function “/” (a, b: unsigned_short_integer)
--                  return unsigned_short_integer;
```

(Continued)

```
--    function “mod”(a, b: unsigned_short_integer)
--                  return  unsigned_short_integer;
--    function “rem”(a, b: unsigned_short_integer)
--                   return  unsigned_short_integer;
--    function “**” (a, b: unsigned_short_integer)
--                   return  unsigned_short_integer;
--    function “abs”(a, b: unsigned_short_integer)
--                   return  unsigned_short_integer;
--
-- type unsigned_tiny_integer is range 0 .. (2**8 - 1);    -- 0..255
--      function "=" (a, b: unsigned_tiny_integer) return boolean;
--      function "/="(a, b: unsigned_tiny_integer) return boolean;
--      function "<" (a, b: unsigned_tiny_integer) return boolean;
--      function "<="(a, b: unsigned_tiny_integer) return boolean;
--      function ">" (a, b: unsigned_tiny_integer) return boolean;
--      function ">="(a, b: unsigned_tiny_integer) return boolean;
--      function "+" (a, b: unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "-" (a, b: unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "+" (a   : unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "-" (a   : unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "*" (a, b: unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "/" (a, b: unsigned_tiny_integer)
--                                          return unsigned_tiny_integer;
--      function "mod"(a, b: unsigned_tiny_integer)
--                                          return  unsigned_tiny_integer;
--      function "rem"(a, b: unsigned_tiny_integer)
--                                          return  unsigned_tiny_integer;
--      function "**" (a, b: unsigned_tiny_integer)
--                                          return  unsigned_tiny_integer;
--      function "abs"(a, b: unsigned_tiny_integer)
--                                          return  unsigned_tiny_integer;
-- end unsigned_types;
```

Figure F-4 Example Specification of `package UNSIGNED_TYPES`

F

F.3 Slices

A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value. The syntax is:

```
prefix(discrete_range)
```

The prefix of a slice must be appropriate for a one-dimensional array type. The type of the slice is the base type of this array type. The bounds of the discrete range define those of the slice and must be of the type of the index. The slice is a *null slice* denoting a null array if the discrete range is a null range.

For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated in some order not defined by the language. The exception `CONSTRAINT_ERROR` is raised by the evaluation of a slice, other than a null slice, if any of the bounds of the discrete range do not belong to the index range of the prefixing array. (The bounds of a null slice need not belong to the subtype of the index.)

References

slices, section 4.1.2 in *Ada Reference Manual*

F.4 Implementation-defined Attributes

F.4.1 'REF

`'REF` denotes the effective address of the first of the storage units allocated to the object. `'REF` is not supported for a package, task unit or entry. The two forms for this attribute are `X'REF` and `SYSTEM.ADDRESS'REF`(N). Only use `X'REF` in machine code procedures. Use `SYSTEM.ADDRESS'REF`(N) anywhere to convert an integer expression to an address.

F.4.2 X'REF

The attribute generates a reference to the entity to which it is applied.

In `X'REF`, X must be either a constant, variable, procedure, function, or label. The attribute returns a value of the `type MACHINE_CODE.OPERAND`. Only use it to designate an operand in a code-statement.

Precede the instruction generated by the code-statement in which the attribute occurs by additional instructions needed to facilitate the reference, such as loading a base register. If the declarative section of the procedure contains `pragma IMPLICIT_CODE (OFF)`, a warning generates if additional code is required.

References

Chapter 4, "Machine Code Insertions"

F.4.3 SYSTEM.ADDRESS'REF(N)

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. However, use `SYSTEM.ADDRESS'REF(N)` instead in the following listed circumstances. In these circumstances, N must be static.

In `SYSTEM.ADDRESS'REF(N)`, `SYSTEM.ADDRESS` must be `type SYSTEM.ADDRESS`. N must be an expression of `type UNIVERSAL_INTEGER`. The attribute returns a value of `type SYSTEM.ADDRESS`, which represents the address designated by N.

- With any of the runtime configuration packages:

  Use of unchecked conversion in an address clause requires the generation of elaboration code, but the configuration packages are not elaborated.

- In any instance where N is greater than `INTEGER'LAST`:

  Such values are required in address clauses that reference the upper portion of memory. To use unchecked conversion in these instances requires the expression be given as a negative integer.

- To place an object at an address, use `'REF`.

In the following example, the *integer_value* converts to an address for use in the address representation clause. The form avoids `UNCHECKED_CONVERSION` and is useful for 32-bit unsigned addresses.

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)

--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY : SYSTEM.ADDRESS := SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

F.4.4 X'TASK_ID

For a non-passive task object or a value, X, `X'TASK_ID` yields the unique task ID associated with the task. The value of this attribute is of the `type SYSTEM.TASK_ID`. If the task objector value X denotes a passive task, the result is the passive task header record object associated with the passive task. The result type is then of `type SYSTEM.PASSIVE_TASK_ID`

F.5 Restrictions on Main Programs

Ada requires that a "main" program must be a non-generic subprogram that is either a procedure or a function returning an Ada `STANDARD.INTEGER` (the predefined type). A "main" program cannot be a generic subprogram or an instantiation of a generic subprogram.

F.6 Generic Declarations

Ada does not require that a generic declaration and the corresponding body be part of the same compilation, and they need not exist in the same Ada library. An error generates if a compilation contains two versions of the same unit.

F.7 Shared Object Code for Generic Subprograms

The Ada compiler generates code for a generic instantiation that can be shared by other instantiations of the same generic, thus reducing the size of the generated code and increasing compilation speed. Overhead is associated with the use of shared-code instantiations because the generic parameters must be accessed indirectly, and for a generic package instantiation, declarations in the

package are accessed indirectly. Greater optimization is possible for unshared instantiations because exact actual parameters are known. It is the responsibility of the programmer to decide whether space or time is most critical in a specific application.

pragma SHARE_CODE controls if an instantiation generates unique code or shares code with other similar instantiations. Apply this pragma to a generic declaration or to individual instantiations.

It is not practical to share the code for instantiations of all generics. If the generic has a formal private type parameter, the generated code to accommodate an instantiation with an arbitrary actual type is extremely inefficient.

The Ada compiler does not share code by default. Specify the INFO directive, SHARE_BODY, in an Ada library, so the compiler always shares generic code bodies. Apply pragma SHARE_CODE to generic units or generic instances to control whether specific instances are shared.

To override the default, pragma SHARE_CODE(name, FALSE) must be used. If formal subprogram parameters exist, instantiations are not shared unless an explicit pragma SHARE_CODE(name, TRUE) is used.

pragma SHARE_CODE indicates whether to share or not share an instantiation. The pragma references either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing for all instantiations of that generic unless specific SHARE_CODE pragmas for individual instantiations override it. If it references an instantiated unit, sharing is set only for that unit.

pragma SHARE_CODE is only allowed immediately in a declarative part, immediately in a package specification or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is:

```
pragma SHARE_CODE (generic, boolean_literal)
```

Note that a parent instantiation (the instantiation that creates the shareable body) is independent of any individual instantiation. Reinstantiation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by other units that share the parent instantiation.

Sharing generics has a slight execution time penalty, because all type attributes are referenced indirectly (as if an extra calling argument is added). However, it reduces program size and in most cases, compilation time substantially.

We compiled a unit, SHARED_IO, in the standard library that instantiates all Ada generic I/O packages for the most commonly-used base types. Thus, any instantiation of an Ada I/O generic package shares one of the parent instantiation generic bodies unless the following pragma is used:

```
pragma SHARE_CODE ( generic, FALSE );
```

F.8 Representation Clauses

Representation Clauses — Ada supports bit-level representation clauses.

pragma PACK — Ada does not define any additional representation pragmas.

Length Clauses — Ada supports all representation clauses.

Enumeration Representation Clauses — Enumeration representation clauses are supported.

Record Representation Clauses — Representation clauses are based on the target machine word-, byte-, and bit-order numbering, so Ada is consistent with machine architecture manuals for both "big-endian" and "little-endian" machines. Bits in a STORAGE_UNIT are numbered according to the target machine manuals. You need not understand the default layout for records and other aggregates since fine control over the layout is obtained from record representation clauses. Align record fields with structures and other aggregates from other languages by specifying the location of each element explicitly. Use 'FIRST_BIT and 'LAST_BIT to construct bit-manipulation code for different bit-numbered systems.

Figure F-5 illustrates SPARC addressing and bit numbering[1].

1. From "Machine Architecture," *Porting Software to SPARC Systems*, p.1, Mountain View, CA: Sun Microsystems, Inc., May 1988, Part No: 800-1796-10

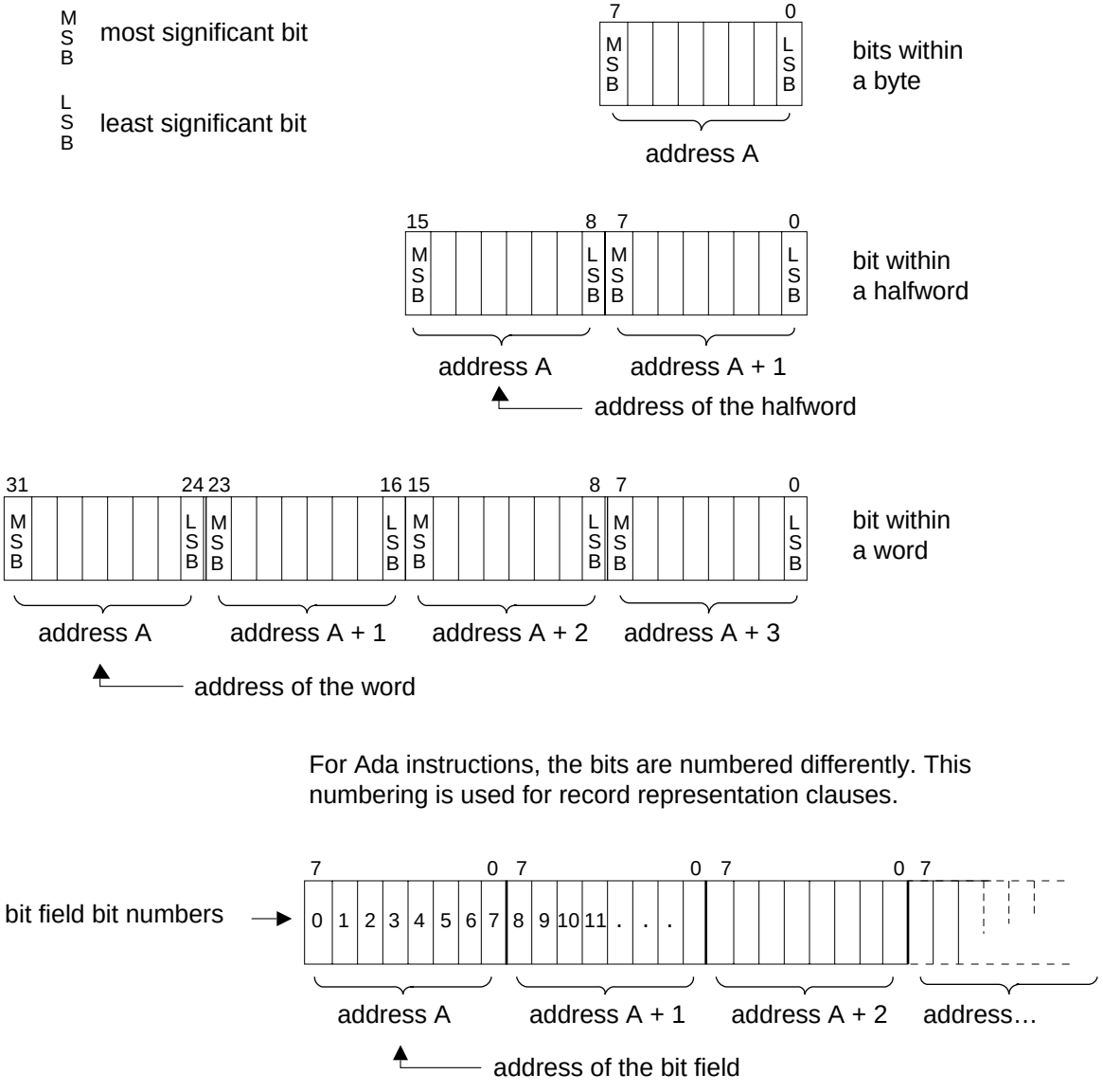

Figure F-5 SPARC Addressing and Bit-numbering Scheme

The only restriction on record representation clauses is that if a component does not start and end on a storage unit boundary, it must be possible to get it in a register with one move instruction. On the Intel 80386, for example, this means such a field must fit in 4 bytes.

The size of object modules is aligned. It is assumed that “mod2” is a worst case restriction assuming that even the C compiler algins things to a 2 byte boundary.

Note that the alignment clause portion of a record representation must be a power of 2. The alignment is obeyed for all allocations of the record type with the following exceptions

- objects declared within a procedure
- objects created by an allocator

For these two exceptions, the maximum alignment obeyed is the default stack and heap alignment.

Address Clauses — Address clauses are supported for the following entities:

- Objects
- Entries

Some Considerations When Using Address Clauses

A common use of Ada address clauses is to locate an Ada object at a specific location in memory, for example:

```
x: some_type;
for x use at some_address;
```

Be aware that some Ada types require default initialization; the compiler generates code to initialize the memory locations where the object is located. A future release of the compiler will give a warning when an object is given an address clause and the object requires default initialization.

Data types that require default initialization include:

- Access types (access types initialize to 0)
- Records with user-defined default initialization
- Records with dynamic-sized components, for example, an array whose bounds are not known at compile time

- Records with gaps between components (the record initializes to zero in order to support record comparisons)
- Records or arrays whose component type is any of the above

If you use an address clause to locate an object at a memory location, but you do not want any writes to this memory except those that your Ada program provides explicitly, define your types so default initialization is not required.

For example, for access types, one approach is to put a 32-bit integer at the memory location and assign the access value using an `UNCHECKED_CONVERSION`.

Caution – Use code that references memory-mapped devices using a `for use at` clause to locate an object at the I/O address with caution. The default optimization of the compiler eliminates redundant moves to and from memory.

If this is a problem, compile with `pragma OPTIMIZE_CODE(OFF)`.

References

Section F.43, "SYSTEM.ADDRESS'REF(N)," on page F-20

Interrupt Entries on Solaris 2.1 — Ada allows task entries to be associated with Solaris 2.1 signals. Solaris 2.1 handles all interrupts and faults initially and returns control to the user program as a signal.

The available signals are described in *Solaris Developer Documentation*, `SIGNAL(3)`. Some signals cannot be caught because of restrictions in Solaris 2.1. The attempt to assign an entry to these signals results in a kernel error "replace vector."

```
#define    SIGKILL     9     /* kill */
```

The Ada runtime discourages attempts to catch the timer-related signals, since these are used with time slicing.

Figure F-6 shows how to attach to the interrupt-from-keyboard signal or Control–c:

```
with system;    use system;
with text_io;

task interrupt is
  entry SIGINT;
     for SIGINT use at address'ref(2);  -- interrupt
end;
task body interrupt is
begin
   loop
     accept SIGINT do
        text_io.put_line("SIGINT");
     end;
   end loop;
end;
```

Figure F-6 Example of Attach interrupt-from-keyboard Signal

Signal handlers are set up for the following signals by the Ada Runtime system:

Table F-1 Signals

SIGILL	**4**	**illegal instruction**
SIGFPE	8	floating point exception
SIGSEGV	11	segmentation violation
SIGTERM	15	Solaris MT Ada (asynchronous abort of another task)
SIGWAITING	32	Solaris Threads
SIGLWP	33	Solaris Threads

If a task entry is attached to SIGFPE or SIGILL, NUMERIC_ERROR exceptions are not raised correctly. If a task entry is attached to SIGSEGV, STORAGE_ERROR exceptions may not be raised correctly.

Use of signal handlers is complicated when non-Ada routines are involved.

References

Section 5.3, “Calling Ada From Other Languages,” on page 5-22

Change of Representation — Change of representation is supported.

`package SYSTEM` — The specification of `package SYSTEM` is available both earlier in this chapter and on line in the file `system.a` in the `standard` release library. The implementation recognizes `pragma`s `SYSTEM_NAME`, `STORAGE_UNIT`, and `MEMORY_SIZE`, but they have no effect. The implementation does not allow `SYSTEM` to be modified by means of pragmas. However, achieve the same effect by recompiling `package SYSTEM` with altered values. Note that such a compilation causes other units in the `standard` library to be out-of-date. Consequently, recompile in a library other than `standard`.

References

Figure F-1 on page F-14

System-dependent Named Numbers — The specification of `package SYSTEM` is listed earlier in this chapter. This specification is available on line in the file `system.a` in the `standard` release library.

Representation Attributes — `'ADDRESS` is supported for the following entities.

- Variables
- Constants
- Procedures
- Functions

If the prefix of an address attribute is an object that is not aligned on a storage unit boundary, the attribute yields the address of the storage unit with the first bit of the object. This is consistent with the `FIRST_BIT` attribute definition.

All other Ada representation attributes are supported fully.

Representation Attributes of Real Types — These attributes are supported.

References

Section F.2, "Predefined Packages and Generics," on page F-11

Machine Code Insertions — Machine code insertions are supported.

References

Chapter 4, "Machine Code Insertions"

Interface to Other Languages — The Ada interface to other languages is discussed in the Interface Programming chapter and the section Pragmas and Their Effects.

References

Chapter 5, "Interface Programming"
Section F.1, "Pragmas and Their Effects," on page F-2

Unchecked Programming — Both `UNCHECKED_DEALLOCATION` and `UNCHECKED_CONVERSION` are provided.

Unchecked Storage Deallocations — Deallocate any allocated object. No checks are performed currently on released objects. However, when an object is deallocated, its access variable is set to null. Subsequent deallocations using the null access variable are ignored.

Unchecked Type Conversions — The predefined `generic function UNCHECKED_CONVERSION` cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

F.9 Parameter Passing

Parameters are passed in registers and by pushing values (or addresses) on the stack. Extra information is passed for records (`'CONSTRAINED`) and for arrays (dope-vector address).

Small results return in registers; large results with known targets are passed by reference. Large results of anonymous target and known size are passed by reference to temporary space created on the caller stack. Large results of anonymous target and unknown size are returned by copying the value from temporary space in the callee, so that this space is reclaimed.

The compiler assumes the following calling conventions:

1. Caller passes scalar arguments in `%o0 - %o5` and floating point arguments in the floating point registers; other arguments are passed on the stack.
2. Caller calls callee.
3. Callee gets a new register window and allocates space for locals with the `SAVE` instruction.
4. Callee executes.
5. Callee leaves function results in `%o0` or `%f0` and `%f1`.
6. Callee restores the previous register window with the `RESTORE` instruction.
7. Callee returns to caller.
8. Caller deallocates space used for arguments on the stack.

Use machine code insertions to explicitly build a call interface when compiler conventions are not compatible or when interfacing to assembly language.

For example, suppose an interface to a C function `pass_int` is desired, where the C compiler generated code such that the callee deallocates space for the parameters.

```
int pass_int(x)
   int x;
   {...
   }
```

Figure F-7 provides a wrapper to call this function, while the C function handles the deallocation.

```
with MACHINE_CODE;
        function PASS_INT(X: INTEGER) return INTEGER is
                RETURN_VAL: INTEGER;
                procedure WRAPPER is
                           use MACHINE_CODE;
                begin;
                         code_2'(sethi, hi("_pass_int"), g1);
                         code_2'(ld, x'ref,o0);
                         code_3'(sub, sp, + 32, sp);
                         code_2'(mov, g4, l0);
                        code_2'(jmpl, g1 + lo("_pass_int"), o7);
                         code_0'(op => nop);
                         code_2'(mov, l0, G4);
                         code_3(add, sp, + 32, sp);
                  end WRAPPER;
                begin
                  WRAPPER;
                  return RETURN_VAL;
                end PASS_INT;
```

Figure F-7 Example of Interface to C Function

References

Chapter 4, "Machine Code Insertions"

F.10 Conversion and Deallocation

The predefined `generic function UNCHECKED_CONVERSION`, cannot be instantiated with a target type that is an unconstrained-array type or an unconstrained-record type with discriminants.

No restrictions exist on the types with which `generic function UNCHECKED_DEALLOCATION` can be instantiated. No checks are performed on released objects.

F.11 Process Stack Size

Occasionally, the compiler and other large dynamic compiled programs give problems due to the shell stack limit. Altering the stack size and recompiling or re-executing is sometimes necessary. A process inherits its stack limit from the invoking process, usually the shell.

The C shell allows the default stack size to be reset, usually up to the limit of the process size. To change the stacksize for the C shell, execute the following command or include it in the `.login` file.

```
limit stacksize number
```

Most Bourne shell implementations do not permit the stack size to be altered.

F.12 Types, Ranges, and Attributes

The maximum `ARRAY`, `RECORD`, and `TYPE` limits increase to 256_000_000 bits.

Numeric Literals — Ada uses unlimited precision arithmetic for computations with numeric literals.

Enumeration Types — Ada allows an unlimited number of literals in an enumeration type.

Attributes of Discrete Types — Ada defines the image of a character that is not a graphic character as the corresponding 2- or 3-character identifier from `package ASCII` of Annex C-4 of the *Ada Reference Manual*. The identifier is in uppercase without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence `CR` (`ASCII.CR`).

`type STRING` — Except for memory size, Ada places no specific limit on the length of the predefined `type STRING`. Any type derived from the `type STRING` is similarly unlimited. See Table F-2.

Table F-2 `type STRING` Attribute Values

Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
SIZE	32	16	8
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

Operation of Floating-point Types — Ada floating-point types have the attributes in Table F-3:

Table F-3 Floating-point Type Attribute Values

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST LAST	-1.79769313486232E+308 1.79769313486232E+308	-3.40282E+38 3.40282E+38
DIGITS MANTISSA	15 52	6 23
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL LARGE	1.94469227433161E-62 2.57110087081438E+61	2.58493941422821E-26 1.93428038904620E+25
SAFE_EMAX SAFE_SMALL SAFE_LARGE	1021 2.22507385850720E-308 2.24711641857789E+307	125 1.17549435082229E-38 4.25352755827077E+37
MACHINE_RADIX MACHINE_MANTISSA MACHINE_EMAX MACHINE_EMIN	2 53 1024 -1021	2 24 128 -125
MACHINE_ROUNDS MACHINE_OVERFLOWS	TRUE TRUE	TRUE TRUE

Fixed-point Types — Ada provides fixed-point types mapped to the supported integer sizes.

Operation of Fixed-point Types — Ada fixed-point `type DURATION` has the attributes in Table F-4.

Table F-4 Fixed-point Type Attribute Values

Name of Attribute	Attribute Value for `DURATION`
`SIZE`	`32`
FIRST LAST	`-214748.3648` `214748.3647`
DELTA	`1.00000000000000E-04`
MANTISSA	`31`
SMALL LARGE	`1.00000000000000E-04` `2.14748364700000E+05`
FORE AFT	`7` `4`
SAFE_SMALL SAFE_LARGE	`1.00000000000000E-04` `2.14748364700000E+05`
MACHINE_ROUNDS MACHINE_OVERFLOWS	`TRUE` `TRUE`

F.13 Input/Output

The Ada I/O system is implemented using Solaris 2.1 I/O. Both formatted and binary I/O are available. No restrictions are on the types with which `DIRECT_IO` and `SEQUENTIAL_IO` can be instantiated, except that the element size must be less than a maximum given by the variable `SYSTEM.MAX_REC_SIZE`. Set this variable to any value prior to the generic instantiation; thus, use any element size. Instantiate `DIRECT_IO` with unconstrained types, but each element is padded to the maximum for that type or to `SYSTEM.MAX_REC_SIZE`, whichever is smaller. No checking — other than normal static Ada type checking — is done to ensure that values from files are read into correctly-sized and -typed objects.

In most cases, Ada file and terminal input-output are identical and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal. `procedure FILE_SUPPORT.ALWAYS_FLUSH` (*file_ptr*) flushes the buffer associated with *file_ptr* after all output requests. Refer to the source code

for file_spprt_b.a in the standard library. Note that the limited private type, file_type defined in TEXT_IO, is derived from the type *file_ptr*. Currently, you must convert between them using UNCHECKED_CONVERSION, because the derivation happens in the private part of the specification of TEXT_IO.

For example, Figure F-8 stops buffering for standard output:

```
with text_io;
with file_support;
with unchecked_conversion;
procedure dont_buffer(file: text_io.file_type) is
   function cvt is new unchecked_conversion(
      source => text_io.file_type,
      target => file_support.file_ptr);
begin
   file_support.always_flush(cvt(file));
end;
```

Figure F-8 Example of Stop Buffering for Standard Output

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * STORAGE_UNIT bits. DIRECT_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as STRING where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X