

GNAT User's Guide

for OpenVMS Alpha

GNAT, The GNU Ada 95 Compiler

GNAT Version for GCC 3.3.1

Ada Core Technologies, Inc.

Copyright © 1995-2002, Free Software Foundation

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being “GNU Free Documentation License”, with the Front-Cover Texts being “GNAT User’s Guide for OpenVMS Alpha”, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

About This Guide

This guide describes the use of GNAT, a full language compiler for the Ada 95 programming language, implemented on DIGITAL OpenVMS Alpha Systems. It describes the features of the compiler and tools, and details how to use them to build Ada 95 applications.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Getting Started with GNAT], page 5, describes how to get started compiling and running Ada programs with the GNAT Ada programming environment.
- Chapter 2 [The GNAT Compilation Model], page 11, describes the compilation model used by GNAT.
- Chapter 3 [Compiling Using GNAT COMPILE], page 29, describes how to compile Ada programs with `GNAT COMPILE`, the Ada compiler.
- Chapter 4 [Binding Using GNAT BIND], page 59, describes how to perform binding of Ada programs with `GNAT BIND`, the GNAT binding utility.
- Chapter 5 [Linking Using GNAT LINK], page 87, describes `GNAT LINK`, a program that provides for linking using the GNAT run-time library to construct a program. `GNAT LINK` can also incorporate foreign language object units into the executable.
- Chapter 6 [The GNAT Make Program GNAT MAKE], page 91, describes `GNAT MAKE`, a utility that automatically determines the set of sources needed by an Ada compilation unit, and executes the necessary compilations binding and link.
- Chapter 7 [Renaming Files Using GNAT CHOP], page 99, describes `GNAT CHOP`, a utility that allows you to preprocess a file that contains Ada source code, and split it into one or more new files, one for each compilation unit.
- Chapter 8 [Configuration Pragmas], page 103, describes the configuration pragmas handled by GNAT.
- Chapter 9 [Handling Arbitrary File Naming Conventions Using `gnatname`], page 105, shows how to override the default GNAT file naming conventions, either for an individual unit or globally.
- Chapter 10 [GNAT Project Manager], page 109, describes how to use project files to organize large projects.
- Chapter 11 [Elaboration Order Handling in GNAT], page 141, describes how GNAT helps you deal with elaboration order issues.
- Chapter 12 [The Cross-Referencing Tools `GNAT XREF` and `GNAT FIND`], page 167, discusses `GNAT XREF` and `GNAT FIND`, two tools that provide an easy way to navigate through sources.
- Chapter 13 [File Name Krunching Using `GNAT KRUNCH`], page 177, describes the `GNAT KRUNCH` file name krunching utility, used to handle shortened file names on operating systems with a limit on the length of names.
- Chapter 14 [Preprocessing Using `GNAT PREPROCESS`], page 181, describes `GNAT PREPROCESS`, a preprocessor utility that allows a single source file to be used to generate multiple or parameterized source files, by means of macro substitution.

- Chapter 16 [The GNAT Library Browser GNAT LIST], page 187, describes **GNAT LIST**, a utility that displays information about compiled units, including dependences on the corresponding sources files, and consistency of compilations.
- Chapter 17 [Finding Memory Problems with GNAT Debug Pool], page 191, describes how to use the GNAT-specific Debug Pool in order to detect as early as possible the use of incorrect memory references.
- Chapter 18 [Creating Sample Bodies Using GNAT STUB], page 193, discusses **GNAT STUB**, a utility that generates empty but compilable bodies for library units.
- Chapter 19 [Reducing the Size of Ada Executables with GNAT ELIM], page 195, describes **GNAT ELIM**, a tool which detects unused subprograms and helps the compiler to create a smaller executable for the program.
- Chapter 20 [Other Utility Programs], page 199, discusses several other GNAT utilities, including **GNAT STANDARD**.
- Chapter 21 [Running and Debugging Ada Programs], page 205, describes how to run and debug Ada programs.
- Chapter 23 [Inline Assembler], page 243, shows how to use the inline assembly facility in an Ada program.
- Chapter 24 [Performance Considerations], page 267, reviews the trade offs between using defaults or options in program development.
- Chapter 22 [Compatibility with DEC Ada], page 221, details the compatibility of GNAT with DEC Ada 83 for OpenVMS Alpha.

What You Should Know before Reading This Guide

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

Related Information

For further information about related tools, refer to the following documents:

- *GNAT Reference Manual*, which contains all reference material for the GNAT implementation of Ada 95.
- *Ada 95 Language Reference Manual*, which contains all reference material for the Ada 95 programming language.
- *Debugging with GDB*, located in the GNU:[DOCS] directory, contains all details on the use of the GNU source-level debugger.
- *GNU EMACS Manual*, located in the GNU:[DOCS] directory if the EMACS kit is installed, contains full information on the extensible editor and programming environment EMACS.

Conventions

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

- Functions, utility program names, standard names, and classes.
- ‘Option flags’
- ‘File Names’, ‘button names’, and ‘field names’.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text
and then shown this way.

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

1 Getting Started with GNAT

This chapter describes some simple ways of using GNAT to build executable Ada programs.

1.1 Running GNAT

Three steps are needed to create an executable file from an Ada source file:

1. The source file(s) must be compiled.
2. The file(s) must be bound using the GNAT binder.
3. All appropriate object files must be linked to produce an executable.

All three steps are most commonly handled by using the **GNAT MAKE** utility program that, given the name of the main program, automatically performs the necessary compilation, binding and linking steps.

1.2 Running a Simple Ada Program

Any text editor may be used to prepare an Ada program. If **Glide** is used, the optional Ada mode may be helpful in laying out the program. The program text is a normal text file. We will suppose in our initial example that you have used your editor to prepare the following standard format text file:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
    Put_Line ("Hello WORLD!");
end Hello;
```

This file should be named ‘HELLO.ADB’. With the normal default file naming conventions, GNAT requires that each file contain a single compilation unit whose file name is the unit name, with periods replaced by hyphens; the extension is ‘**ads**’ for a spec and ‘**adb**’ for a body. You can override this default file naming convention by use of the special pragma **Source_File_Name** (see Section 2.4 [Using Other File Names], page 15). Alternatively, if you want to rename your files according to this default convention, which is probably more convenient if you will be using GNAT for all your compilations, then the **GNAT CHOP** utility can be used to generate correctly-named source files (see Chapter 7 [Renaming Files Using GNAT CHOP], page 99).

You can compile the program using the following command (\$ is used as the command prompt in the examples in this document):

```
$ GNAT_COMPILE HELLO.ADB
```

GNAT_COMPILE is the command used to run the compiler. This compiler is capable of compiling programs in several languages, including Ada 95 and C. It assumes that you have given it an Ada program if the file extension is either ‘**.ADS**’ or ‘**.ADB**’, and it will then call the GNAT compiler to compile the specified file.

This compile command generates a file `'HELLO.OBJ'`, which is the object file corresponding to your Ada program. It also generates an "Ada Library Information" file `'HELLO.ALI'`, which contains additional information used to check that an Ada program is consistent. To build an executable file, use `GNAT BIND` to bind the program and `GNAT LINK` to link it. The argument to both `GNAT BIND` and `GNAT LINK` is the name of the `'ali'` file, but the default extension of `'ALI'` can be omitted. This means that in the most common case, the argument is simply the name of the main program:

```
$ GNAT BIND hello
$ GNAT LINK hello
```

A simpler method of carrying out these steps is to use `GNAT MAKE`, a master program that invokes all the required compilation, binding and linking tools in the correct order. In particular, `GNAT MAKE` automatically recompiles any sources that have been modified since they were last compiled, or sources that depend on such modified sources, so that "version skew" is avoided.

```
$ GNAT MAKE HELLO.ADB
```

The result is an executable program called `'hello'`, which can be run by entering:

```
$ hello
```

assuming that the current directory is on the search path for executable programs.

and, if all has gone well, you will see

```
Hello WORLD!
```

appear in response to this command.

1.3 Running a Program with Multiple Units

Consider a slightly more complicated example that has three files: a main program, and the spec and body of a package:


```

package Greetings is
  procedure Hello;
  procedure Goodbye;
end Greetings;

with Ada.Text_IO; use Ada.Text_IO;
package body Greetings is
  procedure Hello is
  begin
    Put_Line ("Hello WORLD!");
  end Hello;

  procedure Goodbye is
  begin
    Put_Line ("Goodbye WORLD!");
  end Goodbye;
end Greetings;

with Greetings;
procedure Gmain is
begin
  Greetings.Hello;
  Greetings.Goodbye;
end Gmain;

```

Following the one-unit-per-file rule, place this program in the following three separate files:

```

‘GREETINGS.ADS’
    spec of package Greetings

‘GREETINGS.ADB’
    body of package Greetings

‘GMAIN.ADB’
    body of main program

```

To build an executable version of this program, we could use four separate steps to compile, bind, and link the program, as follows:

```

$ GNAT_COMPILE GMAIN.ADB
$ GNAT_COMPILE GREETINGS.ADB
$ GNAT_BIND gmain
$ GNAT_LINK gmain

```

Note that there is no required order of compilation when using GNAT. In particular it is perfectly fine to compile the main program first. Also, it is not necessary to compile package specs in the case where there is an accompanying body; you only need to compile the body. If you want to submit these files to the compiler for semantic checking and not code generation, then use the ‘/NOLOAD’ qualifier:

```

$ GNAT_COMPILE GREETINGS.ADS /NOLOAD

```

Although the compilation can be done in separate steps as in the above example, in practice it is almost always more convenient to use the **GNAT MAKE** tool. All you need to know in this case is the name of the main program’s source file. The effect of the above four commands can be achieved with a single one:

```

$ GNAT_MAKE GMAIN.ADB

```

In the next section we discuss the advantages of using **GNAT MAKE** in more detail.

1.4 Using the GNAT MAKE Utility

If you work on a program by compiling single components at a time using `GNAT COMPILE`, you typically keep track of the units you modify. In order to build a consistent system, you compile not only these units, but also any units that depend on the units you have modified. For example, in the preceding case, if you edit `'GMAIN.ADB'`, you only need to recompile that file. But if you edit `'GREETINGS.ADS'`, you must recompile both `'GREETINGS.ADB'` and `'GMAIN.ADB'`, because both files contain units that depend on `'GREETINGS.ADS'`.

`GNAT BIND` will warn you if you forget one of these compilation steps, so that it is impossible to generate an inconsistent program as a result of forgetting to do a compilation. Nevertheless it is tedious and error-prone to keep track of dependencies among units. One approach to handle the dependency-bookkeeping is to use a makefile. However, makefiles present maintenance problems of their own: if the dependencies change as you change the program, you must make sure that the makefile is kept up-to-date manually, which is also an error-prone process.

The `GNAT MAKE` utility takes care of these details automatically. Invoke it using either one of the following forms:

```
$ GNAT MAKE GMAIN.ADB
$ GNAT MAKE GMAIN
```

The argument is the name of the file containing the main program; you may omit the extension. `GNAT MAKE` examines the environment, automatically recompiles any files that need recompiling, and binds and links the resulting set of object files, generating the executable file, `'GMAIN.EXE'`. In a large program, it can be extremely helpful to use `GNAT MAKE`, because working out by hand what needs to be recompiled can be difficult.

Note that `GNAT MAKE` takes into account all the Ada 95 rules that establish dependencies among units. These include dependencies that result from inlining subprogram bodies, and from generic instantiation. Unlike some other Ada make tools, `GNAT MAKE` does not rely on the dependencies that were found by the compiler on a previous compilation, which may possibly be wrong when sources change. `GNAT MAKE` determines the exact set of dependencies from scratch each time it is run.

1.5 Editing with EMACS

EMACS is an extensible self-documenting text editor that is available in a separate VMSINSTAL kit.

Invoke EMACS by typing "EMACS" at the command prompt. To get started, click on the EMACS Help menu and run the EMACS Tutorial. In a character cell terminal, EMACS help is invoked with "Ctrl-h" (also written as "C-h"), and the tutorial by "C-h t".

Documentation on EMACS and other tools is available in EMACS under the pull-down menu button: Help - Info. After selecting Info, use the middle mouse button to select a topic (e.g. EMACS).

In a character cell terminal, do "C-h i" to invoke info, and then "m" (stands for menu) followed by the menu item desired, as in "m EMACS", to get to the EMACS manual. Help on EMACS is also available by typing "HELP EMACS" at the DCL command prompt.

The tutorial is highly recommended in order to learn the intricacies of EMACS, which is sufficiently extensible to provide for a complete programming environment and shell for the sophisticated user.

2 The GNAT Compilation Model

This chapter describes the compilation model used by GNAT. Although similar to that used by other languages, such as C and C++, this model is substantially different from the traditional Ada compilation models, which are based on a library. The model is initially described without reference to the library-based model. If you have not previously used an Ada compiler, you need only read the first part of this chapter. The last section describes and discusses the differences between the GNAT model and the traditional Ada compiler models. If you have used other Ada compilers, this section will help you to understand those differences, and the advantages of the GNAT model.

2.1 Source Representation

Ada source programs are represented in standard text files, using Latin-1 coding. Latin-1 is an 8-bit code that includes the familiar 7-bit ASCII set, plus additional characters used for representing foreign languages (see Section 2.2 [Foreign Language Representation], page 11 for support of non-USA character sets). The format effector characters are represented using their standard ASCII encodings, as follows:

VT	Vertical tab, 16#0B#
HT	Horizontal tab, 16#09#
CR	Carriage return, 16#0D#
LF	Line feed, 16#0A#
FF	Form feed, 16#0C#

Source files are in standard text file format. In addition, GNAT will recognize a wide variety of stream formats, in which the end of physical physical lines is marked by any of the following sequences: LF, CR, CR-LF, or LF-CR. This is useful in accommodating files that are imported from other operating systems.

The end of a source file is normally represented by the physical end of file. However, the control character 16#1A# (SUB) is also recognized as signalling the end of the source file. Again, this is provided for compatibility with other operating systems where this code is used to represent the end of file.

Each file contains a single Ada compilation unit, including any pragmas associated with the unit. For example, this means you must place a package declaration (a package *spec*) and the corresponding body in separate files. An Ada *compilation* (which is a sequence of compilation units) is represented using a sequence of files. Similarly, you will place each subunit or child unit in a separate file.

2.2 Foreign Language Representation

GNAT supports the standard character sets defined in Ada 95 as well as several other non-standard character sets for use in localized versions of the compiler (see Section 3.2.11 [Character Set Control], page 51).

2.2.1 Latin-1

The basic character set is Latin-1. This character set is defined by ISO standard 8859, part 1. The lower half (character codes 16#00# ... 16#7F#) is identical to standard ASCII coding, but the upper half is used to represent additional characters. These include extended letters used by European languages, such as French accents, the vowels with umlauts used in German, and the extra letter A-ring used in Swedish.

For a complete list of Latin-1 codes and their encodings, see the source file of library unit `Ada.Characters.Latin_1` in file `'A-CHLAT1.ADS'`. You may use any of these extended characters freely in character or string literals. In addition, the extended characters that represent letters can be used in identifiers.

2.2.2 Other 8-Bit Codes

GNAT also supports several other 8-bit coding schemes:

Latin-2 Latin-2 letters allowed in identifiers, with uppercase and lowercase equivalence.

Latin-3 Latin-3 letters allowed in identifiers, with uppercase and lowercase equivalence.

Latin-4 Latin-4 letters allowed in identifiers, with uppercase and lowercase equivalence.

Latin-5 Latin-4 letters (Cyrillic) allowed in identifiers, with uppercase and lowercase equivalence.

IBM PC (code page 437)

This code page is the normal default for PCs in the U.S. It corresponds to the original IBM PC character set. This set has some, but not all, of the extended Latin-1 letters, but these letters do not have the same encoding as Latin-1. In this mode, these letters are allowed in identifiers with uppercase and lowercase equivalence.

IBM PC (code page 850)

This code page is a modification of 437 extended to include all the Latin-1 letters, but still not with the usual Latin-1 encoding. In this mode, all these letters are allowed in identifiers with uppercase and lowercase equivalence.

Full Upper 8-bit

Any character in the range 80-FF allowed in identifiers, and all are considered distinct. In other words, there are no uppercase and lowercase equivalences in this range. This is useful in conjunction with certain encoding schemes used for some foreign character sets (e.g. the typical method of representing Chinese characters on the PC).

No Upper-Half

No upper-half characters in the range 80-FF are allowed in identifiers. This gives Ada 83 compatibility for identifier names.

For precise data on the encodings permitted, and the uppercase and lowercase equivalences that are recognized, see the file `'CSETS.ADB'` in the GNAT compiler sources. You will need to obtain a full source release of GNAT to obtain this file.

2.2.3 Wide Character Encodings

GNAT allows wide character codes to appear in character and string literals, and also optionally in identifiers, by means of the following possible encoding schemes:

Hex Coding

In this encoding, a wide character is represented by the following five character sequence:

ESC a b c d

Where a, b, c, d are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, ESC A345 is used to represent the wide character with code 16#A345#. This scheme is compatible with use of the full Wide_Character set.

Upper-Half Coding

The wide character with encoding 16#abcd# where the upper bit is on (in other words, "a" is in the range 8-F) is represented as two bytes, 16#ab# and 16#cd#. The second byte cannot be a format control character, but is not required to be in the upper half. This method can be also used for shift-JIS or EUC, where the internal coding matches the external coding.

Shift JIS Coding

A wide character is represented by a two-character sequence, 16#ab# and 16#cd#, with the restrictions described for upper-half encoding as described above. The internal character code is the corresponding JIS character according to the standard algorithm for Shift-JIS conversion. Only characters defined in the JIS code set table can be used with this encoding method.

EUC Coding

A wide character is represented by a two-character sequence 16#ab# and 16#cd#, with both characters being in the upper half. The internal character code is the corresponding JIS character according to the EUC encoding algorithm. Only characters defined in the JIS code set table can be used with this encoding method.

UTF-8 Coding

A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value, the representation is a one, two, or three byte sequence:

```
16#0000#-16#007f#: 2#0xxxxxxx#
16#0080#-16#07ff#: 2#110xxxxx# 2#10xxxxxx#
16#0800#-16#ffff#: 2#1110xxxx# 2#10xxxxxx# 2#10xxxxxx#
```

where the xxx bits correspond to the left-padded bits of the 16-bit character value. Note that all lower half ASCII characters are represented as ASCII bytes and all upper half characters and other wide characters are represented as sequences of upper-half (The full UTF-8 scheme allows for encoding 31-bit characters as 6-byte sequences, but in this implementation, all UTF-8 sequences of four or more bytes length will be treated as illegal).

Brackets Coding

In this encoding, a wide character is represented by the following eight character sequence:

```
[ " a b c d " ]
```

Where **a**, **b**, **c**, **d** are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, ["A345"] is used to represent the wide character with code **16#A345#**. It is also possible (though not required) to use the Brackets coding for upper half characters. For example, the code **16#A3#** can be represented as ["A3"].

This scheme is compatible with use of the full Wide_Character set, and is also the method used for wide character encoding in the standard ACVC (Ada Compiler Validation Capability) test suite distributions.

Note: Some of these coding schemes do not permit the full use of the Ada 95 character set. For example, neither Shift JIS, nor EUC allow the use of the upper half of the Latin-1 set.

2.3 File Naming Rules

The default file name is determined by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit and replacing the separating dots with hyphens and using uppercase for all letters.

An exception arises if the file name generated by the above rules starts with one of the characters A,G,I, or S, and the second character is a minus. In this case, the character dollar sign is used in place of the minus. The reason for this special rule is to avoid clashes with the standard names for child units of the packages System, Ada, Interfaces, and GNAT, which use the prefixes S- A- I- and G- respectively.

The file extension is **‘.ADS’** for a spec and **‘.ADB’** for a body. The following list shows some examples of these rules.

‘MAIN.ADS’

Main (spec)

‘MAIN.ADB’

Main (body)

‘ARITH_FUNCTIONS.ADS’

Arith_Functions (package spec)

‘ARITH_FUNCTIONS.ADB’

Arith_Functions (package body)

‘FUNC-SPEC.ADS’

Func.Spec (child package spec)

‘FUNC-SPEC.ADB’

Func.Spec (child package body)

‘MAIN-SUB.ADB’

Sub (subunit of Main)

`'A$BAD.ADB'`

A.Bad (child package body)

Following these rules can result in excessively long file names if corresponding unit names are long (for example, if child units or subunits are heavily nested). An option is available to shorten such long file names (called file name "krunching"). This may be particularly useful when programs being developed with GNAT are to be used on operating systems with limited file name lengths. See Section 13.2 [Using GNAT KRUNCH], page 177.

Of course, no file shortening algorithm can guarantee uniqueness over all possible unit names; if file name krunching is used, it is your responsibility to ensure no name clashes occur. Alternatively you can specify the exact file names that you want used, as described in the next section. Finally, if your Ada programs are migrating from a compiler with a different naming convention, you can use the GNAT CHOP utility to produce source files that follow the GNAT naming conventions. (For details see Chapter 7 [Renaming Files Using GNAT CHOP], page 99.)

2.4 Using Other File Names

In the previous section, we have described the default rules used by GNAT to determine the file name in which a given unit resides. It is often convenient to follow these default rules, and if you follow them, the compiler knows without being explicitly told where to find all the files it needs.

However, in some cases, particularly when a program is imported from another Ada compiler environment, it may be more convenient for the programmer to specify which file names contain which units. GNAT allows arbitrary file names to be used by means of the `Source_File_Name` pragma. The form of this pragma is as shown in the following examples:

```
pragma Source_File_Name (My_Uilities.Stacks,
  Spec_File_Name => "MYUTILST_A.ADA");
pragma Source_File_name (My_Uilities.Stacks,
  Body_File_Name => "MYUTILST.ADA");
```

As shown in this example, the first argument for the pragma is the unit name (in this example a child unit). The second argument has the form of a named association. The identifier indicates whether the file name is for a spec or a body; the file name itself is given by a string literal.

The source file name pragma is a configuration pragma, which means that normally it will be placed in the `'GNAT.ADC'` file used to hold configuration pragmas that apply to a complete compilation environment. For more details on how the `'GNAT.ADC'` file is created and used see Section 8.1 [Handling of Configuration Pragmas], page 103

GNAT MAKE handles non-standard file names in the usual manner (the non-standard file name for the main program is simply used as the argument to GNAT MAKE). Note that if the extension is also non-standard, then it must be included in the GNAT MAKE command, it may not be omitted.

2.5 Alternative File Naming Schemes

In the previous section, we described the use of the `Source_File_Name` pragma to allow arbitrary names to be assigned to individual source files. However, this approach requires one pragma for each file, and especially in large systems can result in very long 'GNAT.ADC' files, and also create a maintenance problem.

GNAT also provides a facility for specifying systematic file naming schemes other than the standard default naming scheme previously described. An alternative scheme for naming is specified by the use of `Source_File_Name` pragmas having the following format:

```
pragma Source_File_Name (
  Spec_File_Name => FILE_NAME_PATTERN
  [,Casing       => CASING_SPEC]
  [,Dot_Replacement => STRING_LITERAL]);

pragma Source_File_Name (
  Body_File_Name => FILE_NAME_PATTERN
  [,Casing       => CASING_SPEC]
  [,Dot_Replacement => STRING_LITERAL]);

pragma Source_File_Name (
  Subunit_File_Name => FILE_NAME_PATTERN
  [,Casing          => CASING_SPEC]
  [,Dot_Replacement => STRING_LITERAL]);

FILE_NAME_PATTERN ::= STRING_LITERAL
CASING_SPEC ::= Lowercase | Uppercase | Mixedcase
```

The `FILE_NAME_PATTERN` string shows how the file name is constructed. It contains a single asterisk character, and the unit name is substituted systematically for this asterisk. The optional parameter `Casing` indicates whether the unit name is to be all upper-case letters, all lower-case letters, or mixed-case. If no `Casing` parameter is used, then the default is all upper-case.

The optional `Dot_Replacement` string is used to replace any periods that occur in subunit or child unit names. If no `Dot_Replacement` argument is used then separating dots appear unchanged in the resulting file name. Although the above syntax indicates that the `Casing` argument must appear before the `Dot_Replacement` argument, but it is also permissible to write these arguments in the opposite order.

As indicated, it is possible to specify different naming schemes for bodies, specs, and subunits. Quite often the rule for subunits is the same as the rule for bodies, in which case, there is no need to give a separate `Subunit_File_Name` rule, and in this case the `Body_File_name` rule is used for subunits as well.

The separate rule for subunits can also be used to implement the rather unusual case of a compilation environment (e.g. a single directory) which contains a subunit and a child unit with the same unit name. Although both units cannot appear in the same partition, the Ada Reference Manual allows (but does not require) the possibility of the two units coexisting in the same environment.

The file name translation works in the following steps:

- If there is a specific `Source_File_Name` pragma for the given unit, then this is always used, and any general pattern rules are ignored.

- If there is a pattern type `Source_File_Name` pragma that applies to the unit, then the resulting file name will be used if the file exists. If more than one pattern matches, the latest one will be tried first, and the first attempt resulting in a reference to a file that exists will be used.
- If no pattern type `Source_File_Name` pragma that applies to the unit for which the corresponding file exists, then the standard GNAT default naming rules are used.

As an example of the use of this mechanism, consider a commonly used scheme in which file names are all lower case, with separating periods copied unchanged to the resulting file name, and specs end with ".1.ADA", and bodies end with ".2.ADA". GNAT will follow this scheme if the following two pragmas appear:

```
pragma Source_File_Name
  (Spec_File_Name => "*.1.ADA");
pragma Source_File_Name
  (Body_File_Name => "*.2.ADA");
```

The default GNAT scheme is actually implemented by providing the following default pragmas internally:

```
pragma Source_File_Name
  (Spec_File_Name => "*.ADS", Dot_Replacement => "-");
pragma Source_File_Name
  (Body_File_Name => "*.ADB", Dot_Replacement => "-");
```

Our final example implements a scheme typically used with one of the Ada 83 compilers, where the separator character for subunits was "_" (two underscores), specs were identified by adding '_.ADA', bodies by adding '.ADA', and subunits by adding '.SEP'. All file names were upper case. Child units were not present of course since this was an Ada 83 compiler, but it seems reasonable to extend this scheme to use the same double underscore separator for child units.

```
pragma Source_File_Name
  (Spec_File_Name => "_.ADA",
   Dot_Replacement => "_",
   Casing = Uppercase);
pragma Source_File_Name
  (Body_File_Name => ".ADA",
   Dot_Replacement => "_",
   Casing = Uppercase);
pragma Source_File_Name
  (Subunit_File_Name => ".SEP",
   Dot_Replacement => "_",
   Casing = Uppercase);
```

2.6 Generating Object Files

An Ada program consists of a set of source files, and the first step in compiling the program is to generate the corresponding object files. These are generated by compiling a subset of these source files. The files you need to compile are the following:

- If a package spec has no body, compile the package spec to produce the object file for the package.
- If a package has both a spec and a body, compile the body to produce the object file for the package. The source file for the package spec need not be compiled in this case

because there is only one object file, which contains the code for both the spec and body of the package.

- For a subprogram, compile the subprogram body to produce the object file for the subprogram. The spec, if one is present, is as usual in a separate file, and need not be compiled.
- In the case of subunits, only compile the parent unit. A single object file is generated for the entire subunit tree, which includes all the subunits.
- Compile child units independently of their parent units (though, of course, the spec of all the ancestor unit must be present in order to compile a child unit).
- Compile generic units in the same manner as any other units. The object files in this case are small dummy files that contain at most the flag used for elaboration checking. This is because GNAT always handles generic instantiation by means of macro expansion. However, it is still necessary to compile generic units, for dependency checking and elaboration purposes.

The preceding rules describe the set of files that must be compiled to generate the object files for a program. Each object file has the same name as the corresponding source file, except that the extension is `‘.OBJ’` as usual.

You may wish to compile other files for the purpose of checking their syntactic and semantic correctness. For example, in the case where a package has a separate spec and body, you would not normally compile the spec. However, it is convenient in practice to compile the spec to make sure it is error-free before compiling clients of this spec, because such compilations will fail if there is an error in the spec.

GNAT provides an option for compiling such files purely for the purposes of checking correctness; such compilations are not required as part of the process of building a program. To compile a file in this checking mode, use the `‘/NOLOAD’` qualifier.

2.7 Source Dependencies

A given object file clearly depends on the source file which is compiled to produce it. Here we are using *depends* in the sense of a typical `make` utility; in other words, an object file depends on a source file if changes to the source file require the object file to be recompiled. In addition to this basic dependency, a given object may depend on additional source files as follows:

- If a file being compiled `with`'s a unit *X*, the object file depends on the file containing the spec of unit *X*. This includes files that are `with`'ed implicitly either because they are parents of `with`'ed child units or they are run-time units required by the language constructs used in a particular unit.
- If a file being compiled instantiates a library level generic unit, the object file depends on both the spec and body files for this generic unit.
- If a file being compiled instantiates a generic unit defined within a package, the object file depends on the body file for the package as well as the spec file.
- If a file being compiled contains a call to a subprogram for which pragma `Inline` applies and inlining is activated with the `‘/INLINE=PRAGMA’` qualifier, the object file depends on the file containing the body of this subprogram as well as on the file containing the

spec. Note that for inlining to actually occur as a result of the use of this qualifier, it is necessary to compile in optimizing mode.

The use of ‘`-gnatN`’ activates a more extensive inlining optimization that is performed by the front end of the compiler. This inlining does not require that the code generation be optimized. Like ‘`/INLINE=PRAGMA`’, the use of this qualifier generates additional dependencies.

- If an object file `O` depends on the proper body of a subunit through inlining or instantiation, it depends on the parent unit of the subunit. This means that any modification of the parent unit or one of its subunits affects the compilation of `O`.
- The object file for a parent unit depends on all its subunit body files.
- The previous two rules meant that for purposes of computing dependencies and recompilation, a body and all its subunits are treated as an indivisible whole.

These rules are applied transitively: if unit `A` `with`’s unit `B`, whose elaboration calls an inlined procedure in package `C`, the object file for unit `A` will depend on the body of `C`, in file ‘`C.ADB`’.

The set of dependent files described by these rules includes all the files on which the unit is semantically dependent, as described in the Ada 95 Language Reference Manual. However, it is a superset of what the ARM describes, because it includes generic, inline, and subunit dependencies.

An object file must be recreated by recompiling the corresponding source file if any of the source files on which it depends are modified. For example, if the `make` utility is used to control compilation, the rule for an Ada object file must mention all the source files on which the object file depends, according to the above definition. The determination of the necessary recompilations is done automatically when one uses `GNAT MAKE`.

2.8 The Ada Library Information Files

Each compilation actually generates two output files. The first of these is the normal object file that has a ‘`.OBJ`’ extension. The second is a text file containing full dependency information. It has the same name as the source file, but an ‘`.ALI`’ extension. This file is known as the Ada Library Information (‘`ali`’) file. The following information is contained in the ‘`ali`’ file.

- Version information (indicates which version of GNAT was used to compile the unit(s) in question)
- Main program information (including priority and time slice settings, as well as the wide character encoding used during compilation).
- List of arguments used in the `GNAT COMPILE` command for the compilation
- Attributes of the unit, including configuration pragmas used, an indication of whether the compilation was successful, exception model used etc.
- A list of relevant restrictions applying to the unit (used for consistency) checking.
- Categorization information (e.g. use of pragma `Pure`).
- Information on all `with`’ed units, including presence of `Elaborate` or `Elaborate_All` pragmas.

- Information from any `Linker_Options` pragmas used in the unit
- Information on the use of `Body_Version` or `Version` attributes in the unit.
- Dependency information. This is a list of files, together with time stamp and checksum information. These are files on which the unit depends in the sense that recompilation is required if any of these units are modified.
- Cross-reference data. Contains information on all entities referenced in the unit. Used by tools like `GNAT XREF` and `GNAT FIND` to provide cross-reference information.

For a full detailed description of the format of the ‘`ali`’ file, see the source of the body of unit `Lib.Writ`, contained in file ‘`LIB-WRIT.ADB`’ in the GNAT compiler sources.

2.9 Binding an Ada Program

When using languages such as C and C++, once the source files have been compiled the only remaining step in building an executable program is linking the object modules together. This means that it is possible to link an inconsistent version of a program, in which two units have included different versions of the same header.

The rules of Ada do not permit such an inconsistent program to be built. For example, if two clients have different versions of the same package, it is illegal to build a program containing these two clients. These rules are enforced by the GNAT binder, which also determines an elaboration order consistent with the Ada rules.

The GNAT binder is run after all the object files for a program have been created. It is given the name of the main program unit, and from this it determines the set of units required by the program, by reading the corresponding ALI files. It generates error messages if the program is inconsistent or if no valid order of elaboration exists.

If no errors are detected, the binder produces a main program, in Ada by default, that contains calls to the elaboration procedures of those compilation unit that require them, followed by a call to the main program. This Ada program is compiled to generate the object file for the main program. The name of the Ada file is ‘`B$xxx.ADB`’ (with the corresponding spec ‘`B$xxx.ADS`’) where `xxx` is the name of the main program unit.

Finally, the linker is used to build the resulting executable program, using the object from the main program from the bind step as well as the object files for the Ada units of the program.

2.10 Mixed Language Programming

2.10.1 Interfacing to C

There are two ways to build a program that contains some Ada files and some other language files depending on whether the main program is in Ada or not. If the main program is in Ada, you should proceed as follows:

1. Compile the other language files to generate object files. For instance:

```
GNAT_COMPILE FILE1.C
GNAT_COMPILE FILE2.C
```

2. Compile the Ada units to produce a set of object files and ALI files. For instance:
`GNAT MAKE /ACTIONS=COMPILE MY_MAIN.ADB`
3. Run the Ada binder on the Ada main program. For instance:
`GNAT BIND MY_MAIN.ALI`
4. Link the Ada main program, the Ada objects and the other language objects. For instance:
`GNAT LINK MY_MAIN.ALI FILE1.OBJ FILE2.OBJ`

The three last steps can be grouped in a single command:

```
GNAT MAKE MY_MAIN.ADB /LINKER_QUALIFIERS FILE1.OBJ FILE2.OBJ
```

If the main program is in some language other than Ada, Then you may have more than one entry point in the Ada subsystem. You must use a special option of the binder to generate callable routines to initialize and finalize the Ada units (see Section 4.7 [Binding with Non-Ada Main Programs], page 82). Calls to the initialization and finalization routines must be inserted in the main program, or some other appropriate point in the code. The call to initialize the Ada units must occur before the first Ada subprogram is called, and the call to finalize the Ada units must occur after the last Ada subprogram returns. You use the same procedure for building the program as described previously. In this case, however, the binder only places the initialization and finalization subprograms into file ‘B\$xxx.ADB’ instead of the main program. So, if the main program is not in Ada, you should proceed as follows:

1. Compile the other language files to generate object files. For instance:
`GNAT COMPILE FILE1.C`
`GNAT COMPILE FILE2.C`
2. Compile the Ada units to produce a set of object files and ALI files. For instance:
`GNAT MAKE /ACTIONS=COMPILE ENTRY_POINT1.ADB`
`GNAT MAKE /ACTIONS=COMPILE ENTRY_POINT2.ADB`
3. Run the Ada binder on the Ada main program. For instance:
`GNAT BIND /NOMAIN ENTRY_POINT1.ALI ENTRY_POINT2.ALI`
4. Link the Ada main program, the Ada objects and the other language objects. You only need to give the last entry point here. For instance:
`GNAT LINK ENTRY_POINT2.ALI FILE1.OBJ FILE2.OBJ`

2.10.2 Calling Conventions

GNAT follows standard calling sequence conventions and will thus interface to any other language that also follows these conventions. The following Convention identifiers are recognized by GNAT:

- Ada. This indicates that the standard Ada calling sequence will be used and all Ada data items may be passed without any limitations in the case where GNAT is used to generate both the caller and callee. It is also possible to mix GNAT generated code and code generated by another Ada compiler. In this case, the data types should be restricted to simple cases, including primitive types. Whether complex data types can be passed depends on the situation. Probably it is safe to pass simple arrays, such as arrays of integers or floats. Records may or may not work, depending on whether both compilers lay them out identically. Complex structures involving variant records, access parameters, tasks, or protected types, are unlikely to be able to be passed.

Note that in the case of GNAT running on a platform that supports DEC Ada 83, a higher degree of compatibility can be guaranteed, and in particular records are laid out in an identical manner in the two compilers. Note also that if output from two different compilers is mixed, the program is responsible for dealing with elaboration issues. Probably the safest approach is to write the main program in the version of Ada other than GNAT, so that it takes care of its own elaboration requirements, and then call the GNAT-generated `adainit` procedure to ensure elaboration of the GNAT components. Consult the documentation of the other Ada compiler for further details on elaboration.

However, it is not possible to mix the tasking run time of GNAT and DEC Ada 83. All the tasking operations must either be entirely within GNAT compiled sections of the program, or entirely within DEC Ada 83 compiled sections of the program.

- **Assembler.** Specifies assembler as the convention. In practice this has the same effect as convention `Ada` (but is not equivalent in the sense of being considered the same convention).
- **Asm.** Equivalent to `Assembler`.
- **Asm.** Equivalent to `Assembly`.
- **COBOL.** Data will be passed according to the conventions described in section B.4 of the Ada 95 Reference Manual.
- **C.** Data will be passed according to the conventions described in section B.3 of the Ada 95 Reference Manual.
- **Default.** Equivalent to `C`.
- **External.** Equivalent to `C`.
- **CPP.** This stands for `C++`. For most purposes this is identical to `C`. See the separate description of the specialized GNAT pragmas relating to `C++` interfacing for further details.
- **Fortran.** Data will be passed according to the conventions described in section B.5 of the Ada 95 Reference Manual.
- **Intrinsic.** This applies to an intrinsic operation, as defined in the Ada 95 Reference Manual. If a pragma `Import (Intrinsic)` applies to a subprogram, this means that the body of the subprogram is provided by the compiler itself, usually by means of an efficient code sequence, and that the user does not supply an explicit body for it. In an application program, the pragma can only be applied to the following two sets of names, which the GNAT compiler recognizes.
 - `Rotate_Left`, `Rotate_Right`, `Shift_Left`, `Shift_Right`, `Shift_Right_` - Arithmetic. The corresponding subprogram declaration must have two formal parameters. The first one must be a signed integer type or a modular type with a binary modulus, and the second parameter must be of type `Natural`. The return type must be the same as the type of the first argument. The size of this type can only be 8, 16, 32, or 64.
 - binary arithmetic operators: `"+"`, `"-"`, `"*"`, `"/"` The corresponding operator declaration must have parameters and result type that have the same root numeric type (for example, all three are `long_float` types). This simplifies the definition of operations that use type checking to perform dimensional checks:

```
type Distance is new Long_Float;
```



```

type Time      is new Long_Float;
type Velocity is new Long_Float;
function "/" (D : Distance; T : Time)
  return Velocity;
pragma Import (Intrinsic, "/");

```

This common idiom is often programmed with a generic definition and an explicit body. The pragma makes it simpler to introduce such declarations. It incurs no overhead in compilation time or code size, because it is implemented as a single machine instruction.

- Stdcall. This is relevant only to NT/Win95 implementations of GNAT, and specifies that the Stdcall calling sequence will be used, as defined by the NT API.
- DLL. This is equivalent to Stdcall.
- Win32. This is equivalent to Stdcall.
- Stubbed. This is a special convention that indicates that the compiler should provide a stub body that raises `Program_Error`.

GNAT additionally provides a useful pragma `Convention_Identifier` that can be used to parametrize conventions and allow additional synonyms to be specified. For example if you have legacy code in which the convention identifier `Fortran77` was used for Fortran, you can use the configuration pragma:

```
pragma Convention_Identifier (Fortran77, Fortran);
```

And from now on the identifier `Fortran77` may be used as a convention identifier (for example in an `Import` pragma) with the same meaning as `Fortran`.

2.11 Building Mixed Ada & C++ Programs

Building a mixed application containing both Ada and C++ code may be a challenge for the unaware programmer. As a matter of fact, this interfacing has not been standardized in the Ada 95 reference manual due to the immaturity and lack of standard of C++ at the time. This section gives a few hints that should make this task easier. In particular the first section addresses the differences with interfacing with C. The second section looks into the delicate problem of linking the complete application from its Ada and C++ parts. The last section give some hints on how the GNAT run time can be adapted in order to allow inter-language dispatching with a new C++ compiler.

2.11.1 Interfacing to C++

GNAT supports interfacing with C++ compilers generating code that is compatible with the standard Application Binary Interface of the given platform.

Interfacing can be done at 3 levels: simple data, subprograms and classes. In the first 2 cases, GNAT offer a specific *Convention CPP* that behaves exactly like *Convention C*. Usually C++ mangle names of subprograms and currently GNAT does not provide any help to solve the demangling problem. This problem can be addressed in 2 ways:

- by modifying the C++ code in order to force a C convention using the *extern "C"* syntax.

- by figuring out the mangled name and use it as the `Link_Name` argument of the pragma `import`.

Interfacing at the class level can be achieved by using the GNAT specific pragmas such as `CPP_Class` and `CPP_Virtual`. See the GNAT Reference Manual for additional information.

2.11.2 Linking a Mixed C++ & Ada Program

Usually the linker of the C++ development system must be used to link mixed applications because most C++ systems will resolve elaboration issues (such as calling constructors on global class instances) transparently during the link phase. GNAT has been adapted to ease the use of a foreign linker for the last phase. Three cases can be considered:

1. Using GNAT and G++ (GNU C++ compiler) from the same GCC installation. The `c++` linker can simply be called by using the `c++` specific driver called `c++`. Note that this setup is not very common because it may request recompiling the whole GCC tree from sources and it does not allow to upgrade easily to a new version of one compiler for one of the two languages without taking the risk of destabilizing the other.

```
$ c++ -c file1.C
$ c++ -c file2.C
$ GNAT MAKE ada_unit /LINKER_QUALIFIERS FILE1.OBJ FILE2.OBJ --LINK=c++
```

2. Using GNAT and G++ from 2 different GCC installations. If both compilers are on the `PATH`, the same method can be used. It is important to be aware that environment variables such as `C_INCLUDE_PATH`, `GCC_EXEC_PREFIX`, `BINUTILS_ROOT` or `GCC_ROOT` will affect both compilers at the same time and thus may make one of the 2 compilers operate improperly if they are set for the other. In particular it is important that the link command has access to the proper GNAT `COMPILE` library `'libgcc.a'`, that is to say the one that is part of the C++ compiler installation. The implicit link command as suggested in the GNAT `MAKE` command from the former example can be replaced by an explicit link command with full verbosity in order to verify which library is used:

```
$ GNAT BIND ada_unit
$ GNAT LINK -v -v ada_unit FILE1.OBJ FILE2.OBJ --LINK=c++
```

If there is a problem due to interfering environment variables, it can be workaround by using an intermediate script. The following example shows the proper script to use when GNAT has not been installed at its default location and `g++` has been installed at its default location:

```
$ GNAT LINK -v -v ada_unit FILE1.OBJ FILE2.OBJ --LINK=./my_script
$ cat ./my_script
#!/bin/sh
unset BINUTILS_ROOT
unset GCC_ROOT
c++ $*
```

3. Using a non GNU C++ compiler. The same set of command as previously described can be used to insure that the `c++` linker is used. Nonetheless, you need to add the path to `libgcc` explicetely, since some libraries needed by GNAT are located in this directory:

```
$ GNAT LINK ada_unit FILE1.OBJ FILE2.OBJ --LINK=./my_script
$ cat ./my_script
#!/bin/sh
CC $* 'GNAT COMPILE -print-libgcc-file-name'
```

Where CC is the name of the non GNU C++ compiler.

2.11.3 A Simple Example

The following example, provided as part of the GNAT examples, show how to achieve procedural interfacing between Ada and C++ in both directions. The C++ class A has 2 methods. The first method is exported to Ada by the means of an extern C wrapper function. The second method calls an Ada subprogram. On the Ada side, The C++ calls is modeled by a limited record with a layout comparable to the C++ class. The Ada subprogram, in turn, calls the c++ method. So from the C++ main program the code goes back and forth between the 2 languages.

Here are the compilation commands for native configurations:

```
$ GNAT MAKE -c simple_cpp_interface
$ c++ -c cpp_main.C
$ c++ -c ex7.C
$ GNAT BIND -n simple_cpp_interface
$ GNAT LINK simple_cpp_interface -o cpp_main --LINK=$(CPLUSPLUS)
-lstdc++ EX7.OBJ CPP_MAIN.OBJ
```

Here are the corresponding sources:

```
//cpp_main.C

#include "ex7.h"

extern "C" {
    void adainit (void);
    void adafinal (void);
    void method1 (A *t);
}

void method1 (A *t)
{
    t->method1 ();
}

int main ()
{
    A obj;
    adainit ();
    obj.method2 (3030);
    adafinal ();
}

//ex7.h

class Origin {
public:
    int o_value;
};
class A : public Origin {
public:
    void method1 (void);
    virtual void method2 (int v);
    A();
    int a_value;
```

```

};

//ex7.C

#include "ex7.h"
#include <stdio.h>

extern "C" { void ada_method2 (A *t, int v);}

void A::method1 (void)
{
    a_value = 2020;
    printf ("in A::method1, a_value = %d \n",a_value);
}

void A::method2 (int v)
{
    ada_method2 (this, v);
    printf ("in A::method2, a_value = %d \n",a_value);
}

A::A(void)
{
    a_value = 1010;
    printf ("in A::A, a_value = %d \n",a_value);
}

-- Ada sources
package body Simple_Cpp_Interface is

    procedure Ada_Method2 (This : in out A; V : Integer) is
    begin
        Method1 (This);
        This.A_Value := V;
    end Ada_Method2;

end Simple_Cpp_Interface;

package Simple_Cpp_Interface is
    type A is limited
        record
            O_Value : Integer;
            A_Value : Integer;
        end record;
    pragma Convention (C, A);

    procedure Method1 (This : in out A);
    pragma Import (C, Method1);

    procedure Ada_Method2 (This : in out A; V : Integer);
    pragma Export (C, Ada_Method2);

end Simple_Cpp_Interface;

```

2.11.4 Adapting the Run Time to a New C++ Compiler

GNAT offers the capability to derive Ada 95 tagged types directly from preexisting C++ classes and . See "Interfacing with C++" in the GNAT reference manual. The mechanism used by GNAT for achieving such a goal has been made user configurable through a GNAT library unit `Interfaces.CPP`. The default version of this file is adapted to the GNU c++ compiler. Internal knowledge of the virtual table layout used by the new C++ compiler is needed to configure properly this unit. The Interface of this unit is known by the compiler and cannot be changed except for the value of the constants defining the characteristics of the virtual table: `CPP_DT_Prologue_Size`, `CPP_DT_Entry_Size`, `CPP_TSD_Prologue_Size`, `CPP_TSD_Entry_Size`. Read comments in the source of this unit for more details.

2.12 Comparison between GNAT and C/C++ Compilation Models

The GNAT model of compilation is close to the C and C++ models. You can think of Ada specs as corresponding to header files in C. As in C, you don't need to compile specs; they are compiled when they are used. The Ada `with` is similar in effect to the `#include` of a C header.

One notable difference is that, in Ada, you may compile specs separately to check them for semantic and syntactic accuracy. This is not always possible with C headers because they are fragments of programs that have less specific syntactic or semantic rules.

The other major difference is the requirement for running the binder, which performs two important functions. First, it checks for consistency. In C or C++, the only defense against assembling inconsistent programs lies outside the compiler, in a makefile, for example. The binder satisfies the Ada requirement that it be impossible to construct an inconsistent program when the compiler is used in normal mode.

The other important function of the binder is to deal with elaboration issues. There are also elaboration issues in C++ that are handled automatically. This automatic handling has the advantage of being simpler to use, but the C++ programmer has no control over elaboration. Where GNAT BIND might complain there was no valid order of elaboration, a C++ compiler would simply construct a program that malfunctioned at run time.

2.13 Comparison between GNAT and Conventional Ada Library Models

This section is intended to be useful to Ada programmers who have previously used an Ada compiler implementing the traditional Ada library model, as described in the Ada 95 Language Reference Manual. If you have not used such a system, please go on to the next section.

In GNAT, there is no *library* in the normal sense. Instead, the set of source files themselves acts as the library. Compiling Ada programs does not generate any centralized information, but rather an object file and a ALI file, which are of interest only to the binder and linker. In a traditional system, the compiler reads information not only from the source

file being compiled, but also from the centralized library. This means that the effect of a compilation depends on what has been previously compiled. In particular:

- When a unit is `with`'ed, the unit seen by the compiler corresponds to the version of the unit most recently compiled into the library.
- Inlining is effective only if the necessary body has already been compiled into the library.
- Compiling a unit may obsolete other units in the library.

In GNAT, compiling one unit never affects the compilation of any other units because the compiler reads only source files. Only changes to source files can affect the results of a compilation. In particular:

- When a unit is `with`'ed, the unit seen by the compiler corresponds to the source version of the unit that is currently accessible to the compiler.
- Inlining requires the appropriate source files for the package or subprogram bodies to be available to the compiler. Inlining is always effective, independent of the order in which units are compiled.
- Compiling a unit never affects any other compilations. The editing of sources may cause previous compilations to be out of date if they depended on the source file being modified.

The most important result of these differences is that order of compilation is never significant in GNAT. There is no situation in which one is required to do one compilation before another. What shows up as order of compilation requirements in the traditional Ada library becomes, in GNAT, simple source dependencies; in other words, there is only a set of rules saying what source files must be present when a file is compiled.

3 Compiling Using GNAT COMPILE

This chapter discusses how to compile Ada programs using the `GNAT COMPILE` command. It also describes the set of qualifiers that can be used to control the behavior of the compiler.

3.1 Compiling Programs

The first step in creating an executable program is to compile the units of the program using the `GNAT COMPILE` command. You must compile the following files:

- the body file (`‘.ADB’`) for a library level subprogram or generic subprogram
- the spec file (`‘.ADS’`) for a library level package or generic package that has no body
- the body file (`‘.ADB’`) for a library level package or generic package that has a body

You need *not* compile the following files

- the spec of a library unit which has a body
- subunits

because they are compiled as part of compiling related units. GNAT package specs when the corresponding body is compiled, and subunits when the parent is compiled. If you attempt to compile any of these files, you will get one of the following error messages (where *fff* is the name of the file you compiled):

```
No code generated for file fff (package spec)
No code generated for file fff (subunit)
```

The basic command for compiling a file containing an Ada unit is

```
$ GNAT COMPILE [qualifiers] ‘file name’
```

where *file name* is the name of the Ada file (usually having an extension `‘.ADS’` for a spec or `‘.ADB’` for a body). The result of a successful compilation is an object file, which has the same name as the source file but an extension of `‘.OBJ’` and an Ada Library Information (ALI) file, which also has the same name as the source file, but with `‘.ALI’` as the extension. GNAT creates these two output files in the current directory, but you may specify a source file in any directory using an absolute or relative path specification containing the directory information.

`GNAT COMPILE` is actually a driver program that looks at the extensions of the file arguments and loads the appropriate compiler. For example, the GNU C compiler is `‘CC1’`, and the Ada compiler is `‘GNAT1’`. These programs are in directories known to the driver program (in some configurations via environment variables you set), but need not be in your path. The `GNAT COMPILE` driver also calls the assembler and any other utilities needed to complete the generation of the required object files.

It is possible to supply several file names on the same `GNAT COMPILE` command. This causes `GNAT COMPILE` to call the appropriate compiler for each file. For example, the following command lists three separate files to be compiled:

```
$ GNAT COMPILE X.ADB Y.ADB Z.C
```

calls `GNAT1` (the Ada compiler) twice to compile `‘X.ADB’` and `‘Y.ADB’`, and `CC1` (the C compiler) once to compile `‘Z.C’`. The compiler generates three object files `‘X.OBJ’`, `‘Y.OBJ’` and `‘Z.OBJ’` and the two ALI files `‘X.ALI’` and `‘Y.ALI’` from the Ada compilations. Any qualifiers apply to all the files listed.

3.2 Qualifiers for GNAT COMPILE

The GNAT COMPILE command accepts qualifiers that control the compilation process. These qualifiers are fully described in this section. First we briefly list all the qualifiers, in alphabetical order, then we describe the qualifiers in more detail in functionally grouped sections.

/DEBUG Generate debugging information. This information is stored in the object file and copied from there to the final executable file by the linker, where it can be read by the debugger. You must use the **/DEBUG** qualifier if you plan on using the debugger.

/SEARCH=*dir*
 Direct GNAT to search the *dir* directory for source files needed by the current compilation (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56).

/NOCURRENT_DIRECTORY
 Except for the source file named in the command line, do not look for source files in the directory containing the source file named in the command line (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56).

/NOOPTIMIZE (default)

/OPTIMIZE[=(keyword[,...])]

 Selects the level of optimization for your program. The supported keywords are as follows:

ALL (default)

 Perform most optimizations, including those that be expensive.

NONE

 Do not do any optimizations. Same as **/NOOPTIMIZE**.

SOME

 Perform some optimizations, but omit ones that are costly.

DEVELOPMENT

 Same as **SOME**.

INLINING

 Full optimization, and also attempt automatic inlining of small sub-programs within a unit (see Section 24.4 [Inlining of Subprograms], page 269).

UNROLL_LOOPS

 Try to unroll loops. This keyword may be specified together with any keyword above other than **NONE**. Loop unrolling usually, but not always, improves the performance of programs.

/RUNTIME_SYSTEM=*rts-path*

 Specifies the default location of the runtime library. Same meaning as the equivalent GNAT MAKE flag (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).

/ASM

 Used to cause the assembler source file to be generated, using **‘.S’** as the extension, instead of the object file. This may be useful if you need to examine the generated assembly code.

/VERBOSE Show commands generated by the GNAT COMPILE driver. Normally used only for debugging purposes or if you need to be sure what version of the compiler you are executing.

/CHECKS=ASSERTIONS
Assertions enabled. `Pragma Assert` and `pragma Debug` to be activated.

-gnatA Avoid processing 'GNAT.ADC'. If a GNAT.ADC file is present, it will be ignored.

/WARNINGS=BRIEF
Generate brief messages to 'SYS\$ERROR' even if verbose mode set.

/NOLOAD Check syntax and semantics only (no code generation attempted).

/COMPRESS_NAMES
Compress debug information and external symbol name table entries.

/XDEBUG Output expanded source files for source level debugging. This qualifier also suppress generation of cross-reference information (see `/XREF=SUPPRESS`).

-gnatecpath
Specify a configuration pragma file. (see Section 8.2 [The Configuration Pragma Files], page 104)

-gnatempath
Specify a mapping file. (see Section 3.2.16 [Units to Sources Mapping Files], page 56)

/CHECKS=ELABORATION
Full dynamic elaboration checks.

/REPORT_ERRORS=FULL
Full errors. Multiple errors per line, all undefined references.

/UPPERCASE_EXTERNALS
Externals names are folded to all uppercase.

/STYLE=GNAT
Internal GNAT implementation mode. This should not be used for applications programs, it is intended only for use by the compiler and its run-time library. For documentation, see the GNAT sources.

/EXPAND_SOURCE
List generated expanded code in source form.

/IDENTIFIER_CHARACTER_SET=c
Identifier character set For details of the possible selections for *c*, see Section 3.2.11 [Character Set Control], page 51.

/HELP Output usage information. The output is written to 'SYS\$OUTPUT'.

/FILE_NAME_MAX_LENGTH=n
Limit file names to *n* (1-999) characters .

/LIST Output full source listing with embedded error messages.

/ERROR_LIMIT=n
Limit number of detected errors to *n* (1-999).

`/INLINE=PRAGMA`
Activate inlining across unit boundaries for subprograms for which pragma `inline` is specified.

`-gnatN` Activate front end inlining.

`/INLINE=SUPPRESS`
Suppresses all inlining, even if other optimization or inlining qualifiers are set.

`/CHECKS=OVERFLOW`
Enable numeric overflow checking (which is not normally enabled by default).
Not that division by zero is a separate check that is not controlled by this qualifier (division by zero checking is on by default).

`/CHECKS=SUPPRESS_ALL`
Suppress all checks.

`/TRY_SEMANTICS`
Don't quit; try semantics, even if parse errors.

`/FORCE_ALI`
Don't quit; generate 'ali' and tree files even if illegalities.

`/POLLING_ENABLE`
Enable polling. This is required on some systems (notably Windows NT) to obtain asynchronous abort and asynchronous transfer of control capability. See the description of pragma Polling in the GNAT Reference Manual for full details.

`/REPRESENTATION_INFO[0/1/2/3] [s]`
Output representation information for declared types and objects.

`/SYNTAX_ONLY`
Syntax check only.

`/TREE_OUTPUT`
Tree output file to be generated.

`-gnatT nnn`
Set time slice to specified number of microseconds

`/UNITS_LIST`
List units for this compilation.

`/UNIQUE_ERROR_TAG`
Tag all error messages with the unique string "error:"

`/REPORT_ERRORS=VERBOSE`
Verbose mode. Full error output with source lines to 'SYS\$OUTPUT'.

`/VALIDITY_CHECKING`
Control level of validity checking. See separate section describing this feature.

`/WARNINGS=xxx`
Warning mode where xxx is a string of options describing the exact warnings that are enabled or disabled. See separate section on warning control.

`/WIDE_CHARACTER_ENCODING=e`
 Wide character encoding method (e =BRACKETS, NONE, HEX, UPPER, SHIFT_JIS, EUC, UTF8)

`/XREF=SUPPRESS`
 Suppress generation of cross-reference information.

`/STYLE_CHECKS=(option,option..)`
 Enable built-in style checks. See separate section describing this feature.

`/DISTRIBUTION_STUBS=m`
 Distribution stub generation and compilation (m =RECEIVER or CALLER to specify the type of stubs to be generated and compiled).

`/83`
 Enforce Ada 83 restrictions.

The following restrictions apply to the combination of qualifiers in this manner:

- The qualifier `'/NOLOAD'` if combined with other qualifiers must come first in the string.
- The qualifier `'/SYNTAX_ONLY'` if combined with other qualifiers must come first in the string.
- Once a "y" appears in the string (that is a use of the `'/STYLE='` qualifier), then all further characters in the qualifier are interpreted as style modifiers (see description of `'/STYLE='`).
- Once a "d" appears in the string (that is a use of the `'-gnatd'` qualifier), then all further characters in the qualifier are interpreted as debug flags (see description of `'-gnatd'`).
- Once a "w" appears in the string (that is a use of the `'-gnatw'` qualifier), then all further characters in the qualifier are interpreted as warning mode modifiers (see description of `'-gnatw'`).
- Once a "V" appears in the string (that is a use of the `'/VALIDITY_CHECKING'` qualifier), then all further characters in the qualifier are interpreted as validity checking options (see description of `'/VALIDITY_CHECKING'`).

3.2.1 Output and Error Message Control

The standard default format for error messages is called "brief format." Brief format messages are written to `'SYS$ERROR'` (the standard error file) and have the following form:

```
E.ADB:3:04: Incorrect spelling of keyword "function"
E.ADB:4:20: ";" should be "is"
```

The first integer after the file name is the line number in the file, and the second integer is the column number within the line. `glide` can parse the error messages and point to the referenced character. The following qualifiers provide control over the error message format:

`/REPORT_ERRORS=VERBOSE`
 The effect of this setting is to write long-format error messages to `'SYS$OUTPUT'` (the standard output file. The same program compiled with the `'/REPORT_ERRORS=VERBOSE'` qualifier would generate:

```

3. funcion X (Q : Integer)
   |
>>> Incorrect spelling of keyword "function"
4. return Integer;
   |
>>> ";" should be "is"

```

The vertical bar indicates the location of the error, and the '>>>' prefix can be used to search for error messages. When this qualifier is used the only source lines output are those with errors.

/LIST This qualifier causes a full listing of the file to be generated. The output might look as follows:

```

1. procedure E is
2.   V : Integer;
3.   funcion X (Q : Integer)
   |
   >>> Incorrect spelling of keyword "function"
4.   return Integer;
   |
   >>> ";" should be "is"
5.   begin
6.     return Q + Q;
7.   end;
8. begin
9.   V := X + X;
10.end E;

```

When you specify the `/REPORT_ERRORS=VERBOSE` or `/LIST` qualifiers and standard output is redirected, a brief summary is written to `'SYS$ERROR'` (standard error) giving the number of error messages and warning messages generated.

/UNIQUE_ERROR_TAG

This qualifier forces all error messages to be preceded by the unique string "error:". This means that error messages take a few more characters in space, but allows easy searching for and identification of error messages.

/WARNINGS=BRIEF

This qualifier causes GNAT to generate the brief format error messages to `'SYS$ERROR'` (the standard error file) as well as the verbose format message or full listing (which as usual is written to `'SYS$OUTPUT'` (the standard output file).

/ERROR_LIMIT=n

n is a decimal integer in the range of 1 to 999 and limits the number of error messages to be generated. For example, using `/ERROR_LIMIT=2` might yield

```

E.ADB:3:04: Incorrect spelling of keyword "function"
E.ADB:5:35: missing ".."
fatal error: maximum errors reached
compilation abandoned

```

`/REPORT_ERRORS=FULL`

Normally, the compiler suppresses error messages that are likely to be redundant. This qualifier causes all error messages to be generated. In particular, in the case of references to undefined variables. If a given variable is referenced several times, the normal format of messages is

```
E.ADB:7:07: "V" is undefined (more references follow)
```

where the parenthetical comment warns that there are additional references to the variable `V`. Compiling the same program with the `'/REPORT_ERRORS=FULL'` qualifier yields

```

E.ADB:7:07: "V" is undefined
E.ADB:8:07: "V" is undefined
E.ADB:8:12: "V" is undefined
E.ADB:8:16: "V" is undefined
E.ADB:9:07: "V" is undefined
E.ADB:9:12: "V" is undefined

```

`/TRY_SEMANTICS`

In normal operation mode, the compiler first parses the program and determines if there are any syntax errors. If there are, appropriate error messages are generated and compilation is immediately terminated. This qualifier tells GNAT to continue with semantic analysis even if syntax errors have been found. This may enable the detection of more errors in a single run. On the other hand, the semantic analyzer is more likely to encounter some internal fatal error when given a syntactically invalid tree.

`/FORCE_ALI`

In normal operation mode, the `'ali'` file is not generated if any illegalities are detected in the program. The use of `'/FORCE_ALI'` forces generation of the `'ali'` file. This file is marked as being in error, so it cannot be used for binding purposes, but it does contain reasonably complete cross-reference information, and thus may be useful for use by tools (e.g. semantic browsing tools or integrated development environments) that are driven from the `'ali'` file.

In addition, if `'/TREE_OUTPUT'` is also specified, then the tree file is generated even if there are illegalities. It may be useful in this case to also specify `'/TRY_SEMANTICS'` to ensure that full semantic processing occurs. The resulting tree file can be processed by ASIS, for the purpose of providing partial information about illegal units, but if the error causes the tree to be badly malformed, then ASIS may crash during the analysis.

In addition to error messages, which correspond to illegalities as defined in the Ada 95 Reference Manual, the compiler detects two kinds of warning situations.

First, the compiler considers some constructs suspicious and generates a warning message to alert you to a possible error. Second, if the compiler detects a situation that is sure to raise an exception at run time, it generates a warning message. The following shows an example of warning messages:

```
E.ADB:4:24: warning: creation of object may raise Storage_Error
E.ADB:10:17: warning: static value out of range
E.ADB:10:17: warning: "Constraint_Error" will be raised at run time
```

GNAT considers a large number of situations as appropriate for the generation of warning messages. As always, warnings are not definite indications of errors. For example, if you do an out-of-range assignment with the deliberate intention of raising a `Constraint_Error` exception, then the warning that may be issued does not indicate an error. Some of the situations for which GNAT issues warnings (at least some of the time) are given in the following list, which is not necessarily complete.

- Possible infinitely recursive calls
- Out-of-range values being assigned
- Possible order of elaboration problems
- Unreachable code
- Fixed-point type declarations with a null range
- Variables that are never assigned a value
- Variables that are referenced before being initialized
- Task entries with no corresponding accept statement
- Duplicate accepts for the same task entry in a select
- Objects that take too much storage
- Unchecked conversion between types of differing sizes
- Missing return statements along some execution paths in a function
- Incorrect (unrecognized) pragmas
- Incorrect external names
- Allocation from empty storage pool
- Potentially blocking operations in protected types
- Suspicious parenthesization of expressions
- Mismatching bounds in an aggregate
- Attempt to return local value by reference
- Unrecognized pragmas
- Premature instantiation of a generic body
- Attempt to pack aliased components
- Out of bounds array subscripts
- Wrong length on string assignment
- Violations of style rules if style checking is enabled
- Unused with clauses
- `Bit_Order` usage that does not have any effect
- Compile time biased rounding of floating-point constant
- `Standard.Duration` used to resolve universal fixed expression

- Dereference of possibly null value
- Declaration that is likely to cause storage error
- Internal GNAT unit with'd by application unit
- Values known to be out of range at compile time
- Unreferenced labels and variables
- Address overlays that could clobber memory
- Unexpected initialization when address clause present
- Bad alignment for address clause
- Useless type conversions
- Redundant assignment statements
- Accidental hiding of name by child unit
- Unreachable code
- Access before elaboration detected at compile time
- A range in a **for** loop that is known to be null or might be null

The following qualifiers are available to control the handling of warning messages:

/WARNINGS=OPTIONAL (activate all optional errors)

This qualifier activates most optional warning messages, see remaining list in this section for details on optional warning messages that can be individually controlled. The warnings that are not turned on by this qualifier are `'/WARNINGS=BIASED_ROUNDING'` (biased rounding), `'/WARNINGS=IMPLICIT_DEREFERENCE'` (implicit dereferencing), and `'/WARNINGS=HIDING'` (hiding). All other optional warnings are turned on.

/WARNINGS=NOOPTIONAL (suppress all optional errors)

This qualifier suppresses all optional warning messages, see remaining list in this section for details on optional warning messages that can be individually controlled.

/WARNINGS=BIASED_ROUNDING (activate warnings on biased rounding)

If a static floating-point expression has a value that is exactly half way between two adjacent machine numbers, then the rules of Ada (Ada Reference Manual, section 4.9(38)) require that this rounding be done away from zero, even if the normal unbiased rounding rules at run time would require rounding towards zero. This warning message alerts you to such instances where compile-time rounding and run-time rounding are not equivalent. If it is important to get proper run-time rounding, then you can force this by making one of the operands into a variable. The default is that such warnings are not generated. Note that `'/WARNINGS=OPTIONAL'` does not affect the setting of this warning option.

/WARNINGS=NOBIASED_ROUNDING (suppress warnings on biased rounding)

This qualifier disables warnings on biased rounding.

/WARNINGS=CONDITIONALS (activate warnings on conditionals)

This qualifier activates warnings for conditional expressions used in tests that are known to be True or False at compile time. The default is that

such warnings are not generated. This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

`/WARNINGS=NOCONDITIONALS` (suppress warnings on conditionals)

This qualifier suppresses warnings for conditional expressions used in tests that are known to be True or False at compile time.

`/WARNINGS=IMPLICIT_DEREFERENCE` (activate warnings on implicit dereferencing)

If this qualifier is set, then the use of a prefix of an access type in an indexed component, slice, or selected component without an explicit `.all` will generate a warning. With this warning enabled, access checks occur only at points where an explicit `.all` appears in the source code (assuming no warnings are generated as a result of this qualifier). The default is that such warnings are not generated. Note that `'/WARNINGS=OPTIONAL'` does not affect the setting of this warning option.

`/WARNINGS=NOIMPLICIT_DEREFERENCE` (suppress warnings on implicit dereferencing)

This qualifier suppresses warnings for implicit dereferences in indexed components, slices, and selected components.

`/WARNINGS=ERROR` (treat warnings as errors)

This qualifier causes warning messages to be treated as errors. The warning string still appears, but the warning messages are counted as errors, and prevent the generation of an object file.

`/WARNINGS=UNREFERENCED_FORMALS` (activate warnings on unreferenced formals)

This qualifier causes a warning to be generated if a formal parameter is not referenced in the body of the subprogram. This warning can also be turned on using `'/WARNINGS=OPTIONAL'` or `'/WARNINGS=UNUSED'`.

`/WARNINGS=NOUNREFERENCED_FORMALS` (suppress warnings on unreferenced formals)

This qualifier suppresses warnings for unreferenced formal parameters. Note that the combination `'/WARNINGS=UNUSED'` followed by `'/WARNINGS=NOUNREFERENCED_FORMALS'` has the effect of warning on unreferenced entities other than subprogram formals.

`/WARNINGS=HIDING` (activate warnings on hiding)

This qualifier activates warnings on hiding declarations. A declaration is considered hiding if it is for a non-overloadable entity, and it declares an entity with the same name as some other entity that is directly or use-visible. The default is that such warnings are not generated. Note that `'/WARNINGS=OPTIONAL'` does not affect the setting of this warning option.

`/WARNINGS=NOHIDING` (suppress warnings on hiding)

This qualifier suppresses warnings on hiding declarations.

`/WARNINGS=IMPLEMENTATION` (activate warnings on implementation units).

This qualifier activates warnings for a `with` of an internal GNAT implementation unit, defined as any unit from the `Ada`, `Interfaces`, `GNAT`, `DEC`, or `System` hierarchies that is not documented in either the Ada Reference Manual or the

GNAT Programmer's Reference Manual. Such units are intended only for internal implementation purposes and should not be `with`'ed by user programs. The default is that such warnings are generated. This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

`/WARNINGS=NOIMPLEMENTATION` (disable warnings on implementation units).

This qualifier disables warnings for a `with` of an internal GNAT implementation unit.

`/WARNINGS=ELABORATION` (activate warnings on elaboration pragmas)

This qualifier activates warnings on missing pragma `Elaborate_All` statements. See the section in this guide on elaboration checking for details on when such pragma should be used. The default is that such warnings are not generated. This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

`/WARNINGS=NOELABORATION` (suppress warnings on elaboration pragmas)

This qualifier suppresses warnings on missing pragma `Elaborate_All` statements. See the section in this guide on elaboration checking for details on when such pragma should be used.

`/WARNINGS=OVERLAYS` (activate warnings on address clause overlays)

This qualifier activates warnings for possibly unintended initialization effects of defining address clauses that cause one variable to overlap another. The default is that such warnings are generated. This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

`/WARNINGS=NOOVERLAYS` (suppress warnings on address clause overlays)

This qualifier suppresses warnings on possibly unintended initialization effects of defining address clauses that cause one variable to overlap another.

`-gnatwp` (activate warnings on ineffective pragma `Inlines`)

This qualifier activates warnings for failure of front end inlining (activated by `'-gnatN'`) to inline a particular call. There are many reasons for not being able to inline a call, including most commonly that the call is too complex to inline. This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

`-gnatwP` (suppress warnings on ineffective pragma `Inlines`)

This qualifier suppresses warnings on ineffective pragma `Inlines`. If the inlining mechanism cannot inline a call, it will simply ignore the request silently.

`/WARNINGS=REDUNDANT` (activate warnings on redundant constructs)

This qualifier activates warnings for redundant constructs. The following is the current list of constructs regarded as redundant: This warning can also be turned on using `'/WARNINGS=OPTIONAL'`.

- Assignment of an item to itself.
- Type conversion that converts an expression to its own type.
- Use of the attribute `Base` where `typ'Base` is the same as `typ`.
- Use of pragma `Pack` when all components are placed by a record representation clause.

`/WARNINGS=NOREDUNDANT` (suppress warnings on redundant constructs)

This qualifier suppresses warnings for redundant constructs.

/WARNINGS=SUPPRESS (suppress all warnings)

This qualifier completely suppresses the output of all warning messages from the GNAT front end. Note that it does not suppress warnings from the GNAT COMPILE back end. To suppress these back end warnings as well, use the qualifier `-w` in addition to `/WARNINGS=SUPPRESS`.

/WARNINGS=UNUSED (activate warnings on unused entities)

This qualifier activates warnings to be generated for entities that are defined but not referenced, and for units that are `with`'ed and not referenced. In the case of packages, a warning is also generated if no entities in the package are referenced. This means that if the package is referenced but the only references are in `use` clauses or `renames` declarations, a warning is still generated. A warning is also generated for a generic package that is `with`'ed but never instantiated. In the case where a package or subprogram body is compiled, and there is a `with` on the corresponding spec that is only referenced in the body, a warning is also generated, noting that the `with` can be moved to the body. The default is that such warnings are not generated. This qualifier also activates warnings on unreferenced formals (it includes the effect of `/WARNINGS=UNREFERENCED_FORMALS`). This warning can also be turned on using `/WARNINGS=OPTIONAL`.

/WARNINGS=NOUNUSED (suppress warnings on unused entities)

This qualifier suppresses warnings for unused entities and packages. It also turns off warnings on unreferenced formals (and thus includes the effect of `/WARNINGS=NOUNREFERENCED_FORMALS`).

A string of warning parameters can be used in the same parameter. For example:

```
-gnatwaLe
```

Would turn on all optional warnings except for elaboration pragma warnings, and also specify that warnings should be treated as errors.

-w

This qualifier suppresses warnings from the GNAT COMPILE backend. It may be used in conjunction with `/WARNINGS=SUPPRESS` to ensure that all warnings are suppressed during the entire compilation process.

3.2.2 Debugging and Assertion Control

/CHECKS=ASSERTIONS

The pragmas `Assert` and `Debug` normally have no effect and are ignored. This qualifier, where `'a'` stands for assert, causes `Assert` and `Debug` pragmas to be activated.

The pragmas have the form:

```
pragma Assert (Boolean-expression [,
               static-string-expression])
pragma Debug (procedure call)
```

The **Assert** pragma causes *Boolean-expression* to be tested. If the result is **True**, the pragma has no effect (other than possible side effects from evaluating the expression). If the result is **False**, the exception **Assert_Failure** declared in the package **System.Assertions** is raised (passing *static-string-expression*, if present, as the message associated with the exception). If no string expression is given the default is a string giving the file name and line number of the pragma.

The **Debug** pragma causes *procedure* to be called. Note that **pragma Debug** may appear within a declaration sequence, allowing debugging procedures to be called between declarations.

/DEBUG[=debug-level]

/NODEBUG Specifies how much debugging information is to be included in the resulting object file where 'debug-level' is one of the following:

TRACEBACK (default)

Include both debugger symbol records and traceback the object file.

ALL

Include both debugger symbol records and traceback in object file.

NONE

Excludes both debugger symbol records and traceback the object file. Same as **/NODEBUG**.

SYMBOLS

Includes only debugger symbol records in the object file. Note that this doesn't include traceback information.

3.2.3 Validity Checking

The Ada 95 Reference Manual has specific requirements for checking for invalid values. In particular, RM 13.9.1 requires that the evaluation of invalid values (for example from unchecked conversions), not result in erroneous execution. In GNAT, the result of such an evaluation in normal default mode is to either use the value unmodified, or to raise **Constraint_Error** in those cases where use of the unmodified value would cause erroneous execution. The cases where unmodified values might lead to erroneous execution are case statements (where a wild jump might result from an invalid value), and subscripts on the left hand side (where memory corruption could occur as a result of an invalid value).

The **'-gnatVx'** qualifier allows more control over the validity checking mode. The **x** argument here is a string of letters which control which validity checks are performed in addition to the default checks described above.

- **'-gnatVc'** Validity checks for copies

The right hand side of assignments, and the initializing values of object declarations are validity checked.

- **'/VALIDITY_CHECKING=RM'** Default (RM) validity checks

Some validity checks are done by default following normal Ada semantics (RM 13.9.1 (9-11)). A check is done in case statements that the expression is within the range of the subtype. If it is not, **Constraint_Error** is raised. For assignments to array components, a check is done that the expression used as index is within the range. If it is not, **Constraint_Error** is raised. Both these validity checks may be turned off using qualifier

'-gnatVD'. They are turned on by default. If '-gnatVD' is specified, a subsequent qualifier '/VALIDITY_CHECKING=RM' will leave the checks turned on. Qualifier '-gnatVD' should be used only if you are sure that all such expressions have valid values. If you use this qualifier and invalid values are present, then the program is erroneous, and wild jumps or memory overwriting may occur.

- '-gnatVi' Validity checks for **in** mode parameters

Arguments for parameters of mode **in** are validity checked in function and procedure calls at the point of call.

- '-gnatVm' Validity checks for **in out** mode parameters

Arguments for parameters of mode **in out** are validity checked in procedure calls at the point of call. The 'm' here stands for modify, since this concerns parameters that can be modified by the call. Note that there is no specific option to test **out** parameters, but any reference within the subprogram will be tested in the usual manner, and if an invalid value is copied back, any reference to it will be subject to validity checking.

- '-gnatVo' Validity checks for operator and attribute operands

Arguments for predefined operators and attributes are validity checked. This includes all operators in package **Standard**, the shift operators defined as intrinsic in package **Interfaces** and operands for attributes such as **Pos**.

- '-gnatVr' Validity checks for function returns

The expression in **return** statements in functions is validity checked.

- '-gnatVs' Validity checks for subscripts

All subscripts expressions are checked for validity, whether they appear on the right side or left side (in default mode only left side subscripts are validity checked).

- '-gnatVt' Validity checks for tests

Expressions used as conditions in **if**, **while** or **exit** statements are checked, as well as guard expressions in entry calls.

- '/VALIDITY_CHECKING=FULL' Validity checks for floating-point values

In the absence of this qualifier, validity checking occurs only for discrete values. If '/VALIDITY_CHECKING=FULL' is specified, then validity checking also applies for floating-point values, and NaN's and infinities are considered invalid, as well as out of range values for constrained types. Note that this means that standard IEEE infinity mode is not allowed. The exact contexts in which floating-point values are checked depends on the setting of other options. For example '-gnatVif' or '-gnatVfi' (the order does not matter) specifies that floating-point parameters of mode **in** should be validity checked.

- '-gnatVa' All validity checks

All the above validity checks are turned on. That is '-gnatVa' is equivalent to **gnatVcdfimorst**.

- '-gnatVn' No validity checks

This qualifier turns off all validity checking, including the default checking for case statements and left hand side subscripts. Note that the use of the qualifier '/CHECKS=SUPPRESS_ALL' suppresses all run-time checks, including validity checks, and thus implies '-gnatVn'.

The `/VALIDITY_CHECKING` qualifier may be followed by a string of letters to turn on a series of validity checking options. For example, `-gnatVcr` specifies that in addition to the default validity checking, copies and function return expressions be validity checked. In order to make it easier to specify a set of options, the upper case letters CDFIMORST may be used to turn off the corresponding lower case option, so for example `-gnatVaM` turns on all validity checking options except for checking of `in out` procedure arguments.

The specification of additional validity checking generates extra code (and in the case of `-gnatva` the code expansion can be substantial. However, these additional checks can be very useful in smoking out cases of uninitialized variables, incorrect use of unchecked conversion, and other errors leading to invalid values. The use of pragma `Initialize_Scalars` is useful in conjunction with the extra validity checking, since this ensures that wherever possible uninitialized variables have invalid values.

See also the pragma `Validity_Checks` which allows modification of the validity checking mode at the program source level, and also allows for temporary disabling of validity checks.

3.2.4 Style Checking

The `/STYLE=(option,option,...)` qualifier causes the compiler to enforce specified style rules.

A limited set of style rules has been used in writing the GNAT sources themselves. This qualifier allows user programs to activate all or some of these checks. If the source program fails a specified style check, an appropriate warning message is given, preceded by the character sequence `"(style)".` (OPTION,OPTION,...) is a sequence of keywords indicating the particular style checks to be performed. The following checks are defined:

1-9 (specify indentation level)

If a digit from 1-9 appears in the string after `/STYLE=` then proper indentation is checked, with the digit indicating the indentation level required. The general style of required indentation is as specified by the examples in the Ada Reference Manual. Full line comments must be aligned with the `--` starting on a column that is a multiple of the alignment level.

ATTRIBUTE (check attribute casing)

If the word `ATTRIBUTE` appears in the string after `/STYLE=` then attribute names, including the case of keywords such as `digits` used as attributes names, must be written in mixed case, that is, the initial letter and any letter following an underscore must be uppercase. All other letters must be lowercase.

BLANKS (blanks not allowed at statement end)

If the word `BLANKS` appears in the string after `/STYLE=` then trailing blanks are not allowed at the end of statements. The purpose of this rule, together with `h` (no horizontal tabs), is to enforce a canonical format for the use of blanks to separate source tokens.

COMMENTS (check comments)

If the word `COMMENTS` appears in the string after `/STYLE=` then comments must meet the following set of rules:

- The `"--"` that starts the column must either start in column one, or else at least one blank must precede this sequence.

- Comments that follow other tokens on a line must have at least one blank following the "-" at the start of the comment.
- Full line comments must have two blanks following the "-" that starts the comment, with the following exceptions.
- A line consisting only of the "-" characters, possibly preceded by blanks is permitted.
- A comment starting with "-x" where x is a special character is permitted. This allows proper processing of the output generated by specialized tools including GNAT PREPROCESS (where -! is used) and the SPARK annotation language (where -# is used). For the purposes of this rule, a special character is defined as being in one of the ASCII ranges 16#21#..16#2F# or 16#3A#..16#3F#.
- A line consisting entirely of minus signs, possibly preceded by blanks, is permitted. This allows the construction of box comments where lines of minus signs are used to form the top and bottom of the box.
- If a comment starts and ends with "-" is permitted as long as at least one blank follows the initial "-". Together with the preceding rule, this allows the construction of box comments, as shown in the following example:

```
-----
-- This is a box comment --
-- with two text lines.  --
-----
```

END (check end/exit labels)

If the word END appears in the string after '/STYLE=' then optional labels on **end** statements ending subprograms and on **exit** statements exiting named loops, are required to be present.

VTABS (no form feeds or vertical tabs)

If the word VTABS appears in the string after '/STYLE=' then neither form feeds nor vertical tab characters are not permitted in the source text.

HTABS (no horizontal tabs)

If the word HTABS appears in the string after '/STYLE=' then horizontal tab characters are not permitted in the source text. Together with the b (no blanks at end of line) check, this enforces a canonical form for the use of blanks to separate source tokens.

IF_THEN (check if-then layout)

If the word IF_THEN appears in the string after '/STYLE=', then the keyword **then** must appear either on the same line as corresponding **if**, or on a line on its own, lined up under the **if** with at least one non-blank line in between containing all or part of the condition to be tested.

KEYWORD (check keyword casing)

If the word KEYWORD appears in the string after '/STYLE=' then all keywords must be in lower case (with the exception of keywords such as **digits** used as attribute names to which this check does not apply).

LAYOUT (check layout)

If the word **LAYOUT** appears in the string after **/STYLE=** then layout of statement and declaration constructs must follow the recommendations in the Ada Reference Manual, as indicated by the form of the syntax rules. For example an **else** keyword must be lined up with the corresponding **if** keyword.

There are two respects in which the style rule enforced by this check option are more liberal than those in the Ada Reference Manual. First in the case of record declarations, it is permissible to put the **record** keyword on the same line as the **type** keyword, and then the **end** in **end record** must line up under **type**. For example, either of the following two layouts is acceptable:

```
type q is record
  a : integer;
  b : integer;
end record;

type q is
  record
    a : integer;
    b : integer;
  end record;
```

Second, in the case of a block statement, a permitted alternative is to put the block label on the same line as the **declare** or **begin** keyword, and then line the **end** keyword up under the block label. For example both the following are permitted:

```
Block : declare
  A : Integer := 3;
begin
  Proc (A, A);
end Block;

Block :
  declare
    A : Integer := 3;
  begin
    Proc (A, A);
  end Block;
```

The same alternative format is allowed for loops. For example, both of the following are permitted:

```

Clear : while J < 10 loop
  A (J) := 0;
end loop Clear;

Clear :
  while J < 10 loop
    A (J) := 0;
  end loop Clear;

```

LINE_LENGTH (check maximum line length)

If the word `LINE_LENGTH` appears in the string after `‘/STYLE=’` then the length of source lines must not exceed 79 characters, including any trailing blanks. The value of 79 allows convenient display on an 80 character wide device or window, allowing for possible special treatment of 80 character lines.

MAX_LENGTH=nnn (set maximum line length)

If the sequence `MAX_LENGTH=nnn`, where `nnn` is a decimal number, appears in the string after `‘/STYLE=’` then the length of lines must not exceed the given value.

STANDARD_CASING (check casing of entities in Standard)

If the word `STANDARD_CASING` appears in the string after `‘/STYLE=’` then any identifier from Standard must be cased to match the presentation in the Ada Reference Manual (for example, `Integer` and `ASCII.NUL`).

ORDERED_SUBPROGRAMS (check order of subprogram bodies)

If the word `ORDERED_SUBPROGRAMS` appears in the string after `‘/STYLE=’` then all subprogram bodies in a given scope (e.g. a package body) must be in alphabetical order. The ordering rule uses normal Ada rules for comparing strings, ignoring casing of letters, except that if there is a trailing numeric suffix, then the value of this suffix is used in the ordering (e.g. `Junk2` comes before `Junk10`).

PRAGMA (check pragma casing)

If the word `PRAGMA` appears in the string after `‘/STYLE=’` then pragma names must be written in mixed case, that is, the initial letter and any letter following an underscore must be uppercase. All other letters must be lowercase.

REFERENCES (check references)

If the word `REFERENCES` appears in the string after `‘/STYLE=’` then all identifier references must be cased in the same way as the corresponding declaration. No specific casing style is imposed on identifiers. The only requirement is for consistency of references with declarations.

SPECS (check separate specs)

If the word `SPECS` appears in the string after `‘/STYLE=’` then separate declarations ("specs") are required for subprograms (a body is not allowed to serve as its own declaration). The only exception is that parameterless library level pro-

cedures are not required to have a separate declaration. This exception covers the most frequent form of main program procedures.

TOKEN (check token spacing)

If the word TOKEN appears in the string after `/STYLE=` then the following token spacing rules are enforced:

- The keywords `abs` and `not` must be followed by a space.
- The token `=>` must be surrounded by spaces.
- The token `<>` must be preceded by a space or a left parenthesis.
- Binary operators other than `**` must be surrounded by spaces. There is no restriction on the layout of the `**` binary operator.
- Colon must be surrounded by spaces.
- Colon-equal (assignment) must be surrounded by spaces.
- Comma must be the first non-blank character on the line, or be immediately preceded by a non-blank character, and must be followed by a space.
- If the token preceding a left paren ends with a letter or digit, then a space must separate the two tokens.
- A right parenthesis must either be the first non-blank character on a line, or it must be preceded by a non-blank character.
- A semicolon must not be preceded by a space, and must not be followed by a non-blank character.
- A unary plus or minus may not be followed by a space.
- A vertical bar must be surrounded by spaces.

In the above rules, appearing in column one is always permitted, that is, counts as meeting either a requirement for a required preceding space, or as meeting a requirement for no preceding space.

Appearing at the end of a line is also always permitted, that is, counts as meeting either a requirement for a following space, or as meeting a requirement for no following space.

If any of these style rules is violated, a message is generated giving details on the violation. The initial characters of such messages are always `"(style)"`. Note that these messages are treated as warning messages, so they normally do not prevent the generation of an object file. The `/WARNINGS=ERROR` qualifier can be used to treat warning messages, including style messages, as fatal errors.

The qualifier `/STYLE_CHECKS=ALL_BUILTIN` is equivalent to all checking options enabled with the exception of `ORDERED.SUBPROGRAMS`, with an indentation level of 3. This is the standard checking option that is used for the GNAT sources.

3.2.5 Run-Time Checks

If you compile with the default options, GNAT will insert many run-time checks into the compiled code, including code that performs range checking against constraints, but not arithmetic overflow checking for integer operations (including division by zero) or checks

for access before elaboration on subprogram calls. All other run-time checks, as required by the Ada 95 Reference Manual, are generated by default. The following GNAT COMPILE qualifiers refine this default behavior:

`/CHECKS=SUPPRESS_ALL`

Suppress all run-time checks as though `pragma Suppress (all_checks)` had been present in the source. Validity checks are also suppressed (in other words `'/CHECKS=SUPPRESS_ALL'` also implies `'-gnatVn'`). Use this qualifier to improve the performance of the code at the expense of safety in the presence of invalid data or program bugs.

`/CHECKS=OVERFLOW`

Enables overflow checking for integer operations. This causes GNAT to generate slower and larger executable programs by adding code to check for overflow (resulting in raising `Constraint_Error` as required by standard Ada semantics). These overflow checks correspond to situations in which the true value of the result of an operation may be outside the base range of the result type. The following example shows the distinction:

```
X1 : Integer := Integer'Last;
X2 : Integer range 1 .. 5 := 5;
...
X1 := X1 + 1;  -- '/CHECKS=OVERFLOW' required to catch the Constraint_Error
X2 := X2 + 1;  -- range check, '/CHECKS=OVERFLOW' has no effect here
```

Here the first addition results in a value that is outside the base range of `Integer`, and hence requires an overflow check for detection of the constraint error. The second increment operation results in a violation of the explicit range constraint, and such range checks are always performed. Basically the compiler can assume that in the absence of the `'/CHECKS=OVERFLOW'` qualifier that any value of type `xxx` is in range of the base type of `xxx`.

Note that the `'/CHECKS=OVERFLOW'` qualifier does not affect the code generated for any floating-point operations; it applies only to integer semantics). For floating-point, GNAT has the `Machine_Overflows` attribute set to `False` and the normal mode of operation is to generate IEEE NaN and infinite values on overflow or invalid operations (such as dividing 0.0 by 0.0).

The reason that we distinguish overflow checking from other kinds of range constraint checking is that a failure of an overflow check can generate an incorrect value, but cannot cause erroneous behavior. This is unlike the situation with a constraint check on an array subscript, where failure to perform the check can result in random memory description, or the range check on a case statement, where failure to perform the check can cause a wild jump.

Note again that `'/CHECKS=OVERFLOW'` is off by default, so overflow checking is not performed in default mode. This means that out of the box, with the default settings, GNAT does not do all the checks expected from the language description in the Ada Reference Manual. If you want all constraint checks to be performed, as described in this Manual, then you must explicitly use the `/CHECKS=OVERFLOW` qualifier either on the GNAT MAKE or GNAT COMPILE command.

/CHECKS=ELABORATION

Enables dynamic checks for access-before-elaboration on subprogram calls and generic instantiations. For full details of the effect and use of this qualifier, See Chapter 3 [Compiling Using GNAT COMPILE], page 29.

The setting of these qualifiers only controls the default setting of the checks. You may modify them using either **Suppress** (to remove checks) or **Unsuppress** (to add back suppressed checks) pragmas in the program source.

3.2.6 Stack Overflow Checking

For most operating systems, GNAT COMPILE does not perform stack overflow checking by default. This means that if the main environment task or some other task exceeds the available stack space, then unpredictable behavior will occur.

To activate stack checking, compile all units with the GNAT COMPILE option **-fstack-check**. For example:

```
GNAT COMPILE -fstack-check PACKAGE1.ADB
```

Units compiled with this option will generate extra instructions to check that any use of the stack (for procedure calls or for declaring local variables in declare blocks) do not exceed the available stack space. If the space is exceeded, then a **Storage_Error** exception is raised.

For declared tasks, the stack size is always controlled by the size given in an applicable **Storage_Size** pragma (or is set to the default size if no pragma is used).

For the environment task, the stack size depends on system defaults and is unknown to the compiler. The stack may even dynamically grow on some systems, precluding the normal Ada semantics for stack overflow. In the worst case, unbounded stack usage, causes unbounded stack expansion resulting in the system running out of virtual memory.

The stack checking may still work correctly if a fixed size stack is allocated, but this cannot be guaranteed. To ensure that a clean exception is signalled for stack overflow, set the environment variable **GNAT_STACK_LIMIT** to indicate the maximum stack area that can be used, as in:

```
SET GNAT_STACK_LIMIT 1600
```

The limit is given in kilobytes, so the above declaration would set the stack limit of the environment task to 1.6 megabytes. Note that the only purpose of this usage is to limit the amount of stack used by the environment task. If it is necessary to increase the amount of stack for the environment task, then this is an operating systems issue, and must be addressed with the appropriate operating systems commands.

3.2.7 Run-Time Control

-gnatT nnn

The **gnatT** qualifier can be used to specify the time-slicing value to be used for task switching between equal priority tasks. The value **nnn** is given in microseconds as a decimal integer.

Setting the time-slicing value is only effective if the underlying thread control system can accommodate time slicing. Check the documentation of your operating system for details. Note that the time-slicing value can also be set by use

of pragma `Time_Slice` or by use of the `t` qualifier in the GNAT BIND step. The pragma overrides a command line argument if both are present, and the `t` qualifier for GNAT BIND overrides both the pragma and the GNAT COMPILE command line qualifier.

3.2.8 Using GNAT COMPILE for Syntax Checking

/SYNTAX_ONLY

Run GNAT in syntax checking only mode. For example, the command

```
$ GNAT COMPILE /SYNTAX_ONLY X.ADB
```

compiles file 'X.ADB' in syntax-check-only mode. You can check a series of files in a single command .

You may use other qualifiers in conjunction with '/SYNTAX_ONLY'. In particular, '/LIST' and '/REPORT_ERRORS=VERBOSE' are useful to control the format of any generated error messages.

The output is simply the error messages, if any. No object file or ALI file is generated by a syntax-only compilation. Also, no units other than the one specified are accessed. For example, if a unit `X` with's a unit `Y`, compiling unit `X` in syntax check only mode does not access the source file containing unit `Y`.

Normally, GNAT allows only a single unit in a source file. However, this restriction does not apply in syntax-check-only mode, and it is possible to check a file containing multiple compilation units concatenated together. This is primarily used by the GNAT CHOP utility (see Chapter 7 [Renaming Files Using GNAT CHOP], page 99).

3.2.9 Using GNAT COMPILE for Semantic Checking

/NOLOAD

Causes the compiler to operate in semantic check mode, with full checking for all illegalities specified in the Ada 95 Reference Manual, but without generation of any object code (no object file is generated).

Because dependent files must be accessed, you must follow the GNAT semantic restrictions on file structuring to operate in this mode:

- The needed source files must be accessible (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56).
- Each file must contain only one compilation unit.
- The file name and unit name must match (see Section 2.3 [File Naming Rules], page 14).

The output consists of error messages as appropriate. No object file is generated. An 'ALI' file is generated for use in the context of cross-reference tools, but this file is marked as not being suitable for binding (since no object file is generated). The checking corresponds exactly to the notion of legality in the Ada 95 Reference Manual.

Any unit can be compiled in semantics-checking-only mode, including units that would not normally be compiled (subunits, and specifications where a separate body is present).

3.2.10 Compiling Ada 83 Programs

/83

Although GNAT is primarily an Ada 95 compiler, it accepts this qualifier to specify that an Ada 83 program is to be compiled in Ada83 mode. If you specify this qualifier, GNAT rejects most Ada 95 extensions and applies Ada 83 semantics where this can be done easily. It is not possible to guarantee this qualifier does a perfect job; for example, some subtle tests, such as are found in earlier ACVC tests (that have been removed from the ACVC suite for Ada 95), may not compile correctly. However, for most purposes, using this qualifier should help to ensure that programs that compile correctly under the ‘/83’ qualifier can be ported easily to an Ada 83 compiler. This is the main use of the qualifier.

With few exceptions (most notably the need to use <> on unconstrained generic formal parameters, the use of the new Ada 95 keywords, and the use of packages with optional bodies), it is not necessary to use the ‘/83’ qualifier when compiling Ada 83 programs, because, with rare exceptions, Ada 95 is upwardly compatible with Ada 83. This means that a correct Ada 83 program is usually also a correct Ada 95 program.

3.2.11 Character Set Control

/IDENTIFIER_CHARACTER_SET=c

Normally GNAT recognizes the Latin-1 character set in source program identifiers, as described in the Ada 95 Reference Manual. This qualifier causes GNAT to recognize alternate character sets in identifiers. *c* is a single character or word indicating the character set, as follows:

1	Latin-1 identifiers
2	Latin-2 letters allowed in identifiers
3	Latin-3 letters allowed in identifiers
4	Latin-4 letters allowed in identifiers
5	Latin-5 (Cyrillic) letters allowed in identifiers
9	Latin-9 letters allowed in identifiers
PC	IBM PC letters (code page 437) allowed in identifiers
PC850	IBM PC letters (code page 850) allowed in identifiers
FULL_UPPER	Full upper-half codes allowed in identifiers
NO_UPPER	No upper-half codes allowed in identifiers

WIDE Wide-character codes (that is, codes greater than 255) allowed in identifiers

See Section 2.2 [Foreign Language Representation], page 11, for full details on the implementation of these character sets.

/WIDE_CHARACTER_ENCODING=e

Specify the method of encoding for wide characters. *e* is one of the following:

HEX Hex encoding (brackets coding also recognized)

UPPER Upper half encoding (brackets encoding also recognized)

SHIFT_JIS Shift/JIS encoding (brackets encoding also recognized)

EUC EUC encoding (brackets encoding also recognized)

UTF8 UTF-8 encoding (brackets encoding also recognized)

BRACKETS Brackets encoding only (default value)

For full details on these encoding methods see Section 2.2.3 [Wide Character Encodings], page 13. Note that brackets coding is always accepted, even if one of the other options is specified, so for example `'/WIDE_CHARACTER_ENCODING=UTF8'` specifies that both brackets and UTF-8 encodings will be recognized. The units that are with'd directly or indirectly will be scanned using the specified representation scheme, and so if one of the non-brackets scheme is used, it must be used consistently throughout the program. However, since brackets encoding is always recognized, it may be conveniently used in standard libraries, allowing these libraries to be used with any of the available coding schemes. scheme. If no `'/WIDE_CHARACTER_ENCODING=?'` parameter is present, then the default representation is Brackets encoding only.

Note that the wide character representation that is specified (explicitly or by default) for the main program also acts as the default encoding used for Wide_Text_IO files if not specifically overridden by a WCEM form parameter.

3.2.12 File Naming Control

/FILE_NAME_MAX_LENGTH=n

Activates file name "krunching". *n*, a decimal integer in the range 1-999, indicates the maximum allowable length of a file name (not including the `' .ADS'` or `' .ADB'` extension). The default is not to enable file name krunching.

For the source file naming rules, See Section 2.3 [File Naming Rules], page 14.

3.2.13 Subprogram Inlining Control

/INLINE=PRAGMA

GNAT recognizes and processes `Inline` pragmas. However, for the inlining to actually occur, optimization must be enabled. To enable inlining across

unit boundaries, this is, inlining a call in one unit of a subprogram declared in a `with`'ed unit, you must also specify this qualifier. In the absence of this qualifier, GNAT does not attempt inlining across units and does not need to access the bodies of subprograms for which `pragma Inline` is specified if they are not in the current unit.

If you specify this qualifier the compiler will access these bodies, creating an extra source dependency for the resulting object file, and where possible, the call will be inlined. For further details on when inlining is possible see Section 24.4 [Inlining of Subprograms], page 269.

`-gnatN` The front end inlining activated by this qualifier is generally more extensive, and quite often more effective than the standard `/INLINE=PRAGMA` inlining mode. It will also generate additional dependencies.

3.2.14 Auxiliary Output Control

`/TREE_OUTPUT`

Causes GNAT to write the internal tree for a unit to a file (with the extension `.adt`). This is not normally required, but is used by separate analysis tools. Typically these tools do the necessary compilations automatically, so you should not have to specify this qualifier in normal operation.

`/UNITS_LIST`

Print a list of units required by this compilation on `'SYS$OUTPUT'`. The listing includes all units on which the unit being compiled depends either directly or indirectly.

3.2.15 Debugging Control

`/EXPAND_SOURCE`

This qualifier causes the compiler to generate auxiliary output containing a pseudo-source listing of the generated expanded code. Like most Ada compilers, GNAT works by first transforming the high level Ada code into lower level constructs. For example, tasking operations are transformed into calls to the tasking run-time routines. A unique capability of GNAT is to list this expanded code in a form very close to normal Ada source. This is very useful in understanding the implications of various Ada usage on the efficiency of the generated code. There are many cases in Ada (e.g. the use of controlled types), where simple Ada statements can generate a lot of run-time code. By using `/EXPAND_SOURCE` you can identify these cases, and consider whether it may be desirable to modify the coding approach to improve efficiency.

The format of the output is very similar to standard Ada source, and is easily understood by an Ada programmer. The following special syntactic additions correspond to low level features used in the generated code that do not have any exact analogies in pure Ada source form. The following is a partial list of these special constructions. See the specification of package `Sprint` in file `'SPRINT.ADS'` for a full list.

`new xxx [storage_pool = yyy]`
Shows the storage pool being used for an allocator.

`at end procedure-name;`
Shows the finalization (cleanup) procedure for a scope.

`(if expr then expr else expr)`
Conditional expression equivalent to the `x?y:z` construction in C.

`target^(source)`
A conversion with floating-point truncation instead of rounding.

`target?(source)`
A conversion that bypasses normal Ada semantic checking. In particular enumeration types and fixed-point types are treated simply as integers.

`target?^(source)`
Combines the above two cases.

`x #/ y`
`x #mod y`
`x #* y`
`x #rem y` A division or multiplication of fixed-point values which are treated as integers without any kind of scaling.

`free expr [storage_pool = xxx]`
Shows the storage pool associated with a `free` statement.

`freeze typename [actions]`
Shows the point at which `typename` is frozen, with possible associated actions to be performed at the freeze point.

`reference itype`
Reference (and hence definition) to internal type `itype`.

`function-name! (arg, arg, arg)`
Intrinsic function call.

`labelname : label`
Declaration of label `labelname`.

`expr && expr && expr ... && expr`
A multiple concatenation (same effect as `expr & expr & expr`, but handled more efficiently).

`[constraint_error]`
Raise the `Constraint_Error` exception.

`expression'reference`
A pointer to the result of evaluating `expression`.

`target-type!(source-expression)`
An unchecked conversion of `source-expression` to `target-type`.

[*numerator/denominator*]

Used to represent internal real literals (that) have no exact representation in base 2-16 (for example, the result of compile time evaluation of the expression 1.0/27.0).

/XDEBUG This qualifier is used in conjunction with **/EXPAND_SOURCE** to cause the expanded source, as described above to be written to files with names **'XXX_DG'**, where **'xxx'** is the normal file name, for example, if the source file name is **'HELLO.ADB'**, then a file **'HELLO.ADB_DG'** will be written. The debugging information generated by the GNAT COMPILE **/DEBUG** qualifier will refer to the generated **'XXX_DG'** file. This allows you to do source level debugging using the generated code which is sometimes useful for complex code, for example to find out exactly which part of a complex construction raised an exception. This qualifier also suppress generation of cross-reference information (see **/XREF=SUPPRESS**).

/COMPRESS_NAMES

In the generated debugging information, and also in the case of long external names, the compiler uses a compression mechanism if the name is very long. This compression method uses a checksum, and avoids trouble on some operating systems which have difficulty with very long names. The **/COMPRESS_NAMES** qualifier forces this compression approach to be used on all external names and names in the debugging information tables. This reduces the size of the generated executable, at the expense of making the naming scheme more complex. The compression only affects the qualification of the name. Thus a name in the source:

```
Very_Long_Package.Very_Long_Inner_Package.Var
```

would normally appear in these tables as:

```
very_long_package__very_long_inner_package__var
```

but if the **/COMPRESS_NAMES** qualifier is used, then the name appears as

```
XCb7e0c705__var
```

Here **b7e0c705** is a compressed encoding of the qualification prefix. The GNAT Ada aware version of GDB understands these encoded prefixes, so if this debugger is used, the encoding is largely hidden from the user of the compiler.

/REPRESENTATION_INFO[0|1|2|3][s]

This qualifier controls output from the compiler of a listing showing representation information for declared types and objects. For **'/REPRESENTATION_INFO=NONE'**, no information is output (equivalent to omitting the **'/REPRESENTATION_INFO'** qualifier). For **'/REPRESENTATION_INFO=ARRAYS'** (which is the default, so **'/REPRESENTATION_INFO'** with no parameter has the same effect), size and alignment information is listed for declared array and record types. For **'/REPRESENTATION_INFO=OBJECTS'**, size and alignment information is listed

for all expression information for values that are computed at run time for variant records. These symbolic expressions have a mostly obvious format with #n being used to represent the value of the n'th discriminant. See source files 'REPINFO.ADS/adb' in the GNAT sources for full details on the format of '/REPRESENTATION_INFO=SYMBOLIC' output. If the qualifier is followed by an s (e.g. '-gnatR2s'), then the output is to a file with the name 'file_REP' where file is the name of the corresponding source file.

/XREF=SUPPRESS

Normally the compiler generates full cross-referencing information in the 'ALI' file. This information is used by a number of tools, including GNAT FIND and GNAT XREF. The /XREF=SUPPRESS qualifier suppresses this information. This saves some space and may slightly speed up compilation, but means that these tools cannot be used.

3.2.16 Units to Sources Mapping Files

-gnatempath

A mapping file is a way to communicate to the compiler two mappings: from unit names to file names (without any directory information) and from file names to path names (with full directory information). These mappings are used by the compiler to short-circuit the path search.

A mapping file is a sequence of sets of three lines. In each set, the first line is the unit name, in lower case, with "%s" appended for specifications and "%b" appended for bodies; the second line is the file name; and the third line is the path name.

Example:

```
main%b
main.2.ADA
/gnat/project1/sources/main.2.ADA
```

When the qualifier '-gnatem' is specified, the compiler will create in memory the two mappings from the specified file. If there is any problem (non existent file, truncated file or duplicate entries), no mapping will be created.

Several '-gnatem' qualifiers may be specified; however, only the last one on the command line will be taken into account.

When using a project file, GNAT MAKE create a temporary mapping file and communicates it to the compiler using this qualifier.

3.3 Search Paths and the Run-Time Library (RTL)

With the GNAT source-based library system, the compiler must be able to find source files for units that are needed by the unit being compiled. Search paths are used to guide this process.

The compiler compiles one source file whose name must be given explicitly on the command line. In other words, no searching is done for this file. To find all other source files that are needed (the most common being the specs of units), the compiler examines the following directories, in the following order:

1. The directory containing the source file of the main unit being compiled (the file name on the command line).
2. Each directory named by an `/SOURCE_SEARCH` qualifier given on the `GNAT COMPILE` command line, in the order given.
3. Each of the directories listed in the value of the `ADA_INCLUDE_PATH` logical name. Normally, define this value as a logical name containing a comma separated list of directory names.

This variable can also be defined by means of an environment string (an argument to the DEC C `exec*` set of functions).

Logical Name:

```
DEFINE ANOTHER_PATH FOO: [BAG]
DEFINE ADA_INCLUDE_PATH ANOTHER_PATH, FOO: [BAM], FOO: [BAR]
```

By default, the path includes `GNU:[LIB.OPENVMS7_x.2.8_x.DECLIB]` first, followed by the standard Ada 95 libraries in `GNU:[LIB.OPENVMS7_x.2.8_x.ADAINCLUDE]`. If this is not redefined, the user will obtain the DEC Ada83 IO packages (`Text_IO`, `Sequential_IO`, etc) instead of the Ada95 packages. Thus, in order to get the Ada 95 packages by default, `ADA_INCLUDE_PATH` must be redefined.

4. The content of the `"ada_source_path"` file which is part of the GNAT installation tree and is used to store standard libraries such as the GNAT Run Time Library (RTL) source files.

Specifying the qualifier `/NOCURRENT_DIRECTORY` inhibits the use of the directory containing the source file named in the command line. You can still have this directory on your search path, but in this case it must be explicitly requested with a `/SOURCE_SEARCH` qualifier.

Specifying the qualifier `/NOSTD_INCLUDES` inhibits the search of the default location for the GNAT Run Time Library (RTL) source files.

The compiler outputs its object files and ALI files in the current working directory.

The packages `Ada`, `System`, and `Interfaces` and their children make up the GNAT RTL, together with the simple `System.IO` package used in the "Hello World" example. The sources for these units are needed by the compiler and are kept together in one directory. Not all of the bodies are needed, but all of the sources are kept together anyway. In a normal installation, you need not specify these directory names when compiling or binding. Either the environment variables or the built-in defaults cause these files to be found.

In addition to the language-defined hierarchies (`System`, `Ada` and `Interfaces`), the GNAT distribution provides a fourth hierarchy, consisting of child units of GNAT. This is a collection of generally useful routines. See the GNAT Reference Manual for further details.

Besides simplifying access to the RTL, a major use of search paths is in compiling sources from multiple directories. This can make development environments much more flexible.

3.4 Order of Compilation Issues

If, in our earlier example, there was a spec for the `hello` procedure, it would be contained in the file `'HELLO.ADS'`; yet this file would not have to be explicitly compiled. This is the result of the model we chose to implement library management. Some of the consequences of this model are as follows:

- There is no point in compiling specs (except for package specs with no bodies) because these are compiled as needed by clients. If you attempt a useless compilation, you will receive an error message. It is also useless to compile subunits because they are compiled as needed by the parent.
- There are no order of compilation requirements: performing a compilation never obsoletes anything. The only way you can obsolete something and require recompilations is to modify one of the source files on which it depends.
- There is no library as such, apart from the ALI files (see Section 2.8 [The Ada Library Information Files], page 19, for information on the format of these files). For now we find it convenient to create separate ALI files, but eventually the information therein may be incorporated into the object file directly.
- When you compile a unit, the source files for the specs of all units that it `with`'s, all its subunits, and the bodies of any generics it instantiates must be available (reachable by the search-paths mechanism described above), or you will receive a fatal error message.

3.5 Examples

The following are some typical Ada compilation command line examples:

```
$ GNAT COMPILE XYZ.ADB
```

Compile body in file 'XYZ.ADB' with all default options.

```
$ GNAT COMPILE /OPTIMIZE=ALL /CHECKS=ASSERTIONS XYZ-DEF.ADB
```

Compile the child unit package in file 'XYZ-DEF.ADB' with extensive optimizations, and pragma `Assert/Debug` statements enabled.

```
$ GNAT COMPILE /NOLOAD ABC-DEF.ADB
```

Compile the subunit in file 'ABC-DEF.ADB' in semantic-checking-only mode.

4 Binding Using GNAT BIND

This chapter describes the GNAT binder, GNAT BIND, which is used to bind compiled GNAT objects. The GNAT BIND program performs four separate functions:

1. Checks that a program is consistent, in accordance with the rules in Chapter 10 of the Ada 95 Reference Manual. In particular, error messages are generated if a program uses inconsistent versions of a given unit.
2. Checks that an acceptable order of elaboration exists for the program and issues an error message if it cannot find an order of elaboration that satisfies the rules in Chapter 10 of the Ada 95 Language Manual.
3. Generates a main program incorporating the given elaboration order. This program is a small Ada package (body and spec) that must be subsequently compiled using the GNAT compiler. The necessary compilation step is usually performed automatically by GNAT LINK. The two most important functions of this program are to call the elaboration routines of units in an appropriate order and to call the main program.
4. Determines the set of object files required by the given main program. This information is output in the forms of comments in the generated program, to be read by the GNAT LINK utility used to link the Ada application.

4.1 Running GNAT BIND

The form of the GNAT BIND command is

```
$ GNAT BIND [qualifiers] mainprog[.ALI] [qualifiers]
```

where *mainprog*.ADB is the Ada file containing the main program unit body. If no qualifiers are specified, GNAT BIND constructs an Ada package in two files which names are 'B\$ada_main.ADS', and 'B\$ada_main.ADB'. For example, if given the parameter 'HELLO.ALI', for a main program contained in file 'HELLO.ADB', the binder output files would be 'B~HELLO.ADS' and 'B~HELLO.ADB'.

When doing consistency checking, the binder takes into consideration any source files it can locate. For example, if the binder determines that the given main program requires the package Pack, whose '.ALI' file is 'PACK.ALI' and whose corresponding source spec file is 'PACK.ADS', it attempts to locate the source file 'PACK.ADS' (using the same search path conventions as previously described for the GNAT COMPILE command). If it can locate this source file, it checks that the time stamps or source checksums of the source and its references to in 'ali' files match. In other words, any 'ali' files that mentions this spec must have resulted from compiling this version of the source file (or in the case where the source checksums match, a version close enough that the difference does not matter).

The effect of this consistency checking, which includes source files, is that the binder ensures that the program is consistent with the latest version of the source files that can be located at bind time. Editing a source file without compiling files that depend on the source file cause error messages to be generated by the binder.

For example, suppose you have a main program 'HELLO.ADB' and a package P, from file 'P.ADS' and you perform the following steps:

1. Enter GNAT COMPILE HELLO.ADB to compile the main program.

2. Enter GNAT COMPILE P.ADS to compile package P.
3. Edit file 'P.ADS'.
4. Enter GNAT BIND hello.

At this point, the file 'P.ALI' contains an out-of-date time stamp because the file 'P.ADS' has been edited. The attempt at binding fails, and the binder generates the following error messages:

```
error: "HELLO.ADB" must be recompiled ("P.ADS" has been modified)
error: "P.ADS" has been modified and must be recompiled
```

Now both files must be recompiled as indicated, and then the bind can succeed, generating a main program. You need not normally be concerned with the contents of this file, but it is similar to the following which is the binder file generated for a simple "hello world" program.

```
-- The package is called Ada_Main unless this name is actually used
-- as a unit name in the partition, in which case some other unique
-- name is used.

with System;
package ada_main is

  Elab_Final_Code : Integer;
  pragma Import (C, Elab_Final_Code, "__gnat_inside_elab_final_code");

  -- The main program saves the parameters (argument count,
  -- argument values, environment pointer) in global variables
  -- for later access by other units including
  -- Ada.Command_Line.

  gnat_argc : Integer;
  gnat_argv : System.Address;
  gnat_envp : System.Address;

  -- The actual variables are stored in a library routine. This
  -- is useful for some shared library situations, where there
  -- are problems if variables are not in the library.

  pragma Import (C, gnat_argc);
  pragma Import (C, gnat_argv);
  pragma Import (C, gnat_envp);

  -- The exit status is similarly an external location

  gnat_exit_status : Integer;
  pragma Import (C, gnat_exit_status);

  GNAT_Version : constant String :=
    "GNAT Version: 3.15w (20010315)";
  pragma Export (C, GNAT_Version, "__gnat_version");

  -- This is the generated adafinal routine that performs
  -- finalization at the end of execution. In the case where
  -- Ada is the main program, this main program makes a call
  -- to adafinal at program termination.
```

```

procedure adafinal;
pragma Export (C, adafinal, "adafinal");

-- This is the generated adainit routine that performs
-- initialization at the start of execution. In the case
-- where Ada is the main program, this main program makes
-- a call to adainit at program startup.

procedure adainit;
pragma Export (C, adainit, "adainit");

-- This routine is called at the start of execution. It is
-- a dummy routine that is used by the debugger to breakpoint
-- at the start of execution.

procedure Break_Start;
pragma Import (C, Break_Start, "__gnat_break_start");

-- This is the actual generated main program (it would be
-- suppressed if the no main program qualifier were used). As
-- required by standard system conventions, this program has
-- the external name main.

function main
  (argc : Integer;
   argv : System.Address;
   envp : System.Address)
  return Integer;
pragma Export (C, main, "main");

-- The following set of constants give the version
-- identification values for every unit in the bound
-- partition. This identification is computed from all
-- dependent semantic units, and corresponds to the
-- string that would be returned by use of the
-- Body_Version or Version attributes.

type Version_32 is mod 2 ** 32;
u00001 : constant Version_32 := 16#7880BEB3#;
u00002 : constant Version_32 := 16#0D24CBD0#;
u00003 : constant Version_32 := 16#3283DBEB#;
u00004 : constant Version_32 := 16#2359F9ED#;
u00005 : constant Version_32 := 16#664FB847#;
u00006 : constant Version_32 := 16#68E803DF#;
u00007 : constant Version_32 := 16#5572E604#;
u00008 : constant Version_32 := 16#46B173D8#;
u00009 : constant Version_32 := 16#156A40CF#;
u00010 : constant Version_32 := 16#033DABE0#;
u00011 : constant Version_32 := 16#6AB38FEA#;
u00012 : constant Version_32 := 16#22B6217D#;
u00013 : constant Version_32 := 16#68A22947#;
u00014 : constant Version_32 := 16#18CC4A56#;
u00015 : constant Version_32 := 16#08258E1B#;
u00016 : constant Version_32 := 16#367D5222#;
u00017 : constant Version_32 := 16#20C9ECA4#;
u00018 : constant Version_32 := 16#50D32CB6#;
u00019 : constant Version_32 := 16#39A8BB77#;

```

```

u00020 : constant Version_32 := 16#5CF8FA2B#;
u00021 : constant Version_32 := 16#2F1EB794#;
u00022 : constant Version_32 := 16#31AB6444#;
u00023 : constant Version_32 := 16#1574B6E9#;
u00024 : constant Version_32 := 16#5109C189#;
u00025 : constant Version_32 := 16#56D770CD#;
u00026 : constant Version_32 := 16#02F9DE3D#;
u00027 : constant Version_32 := 16#08AB6B2C#;
u00028 : constant Version_32 := 16#3FA37670#;
u00029 : constant Version_32 := 16#476457A0#;
u00030 : constant Version_32 := 16#731E1B6E#;
u00031 : constant Version_32 := 16#23C2E789#;
u00032 : constant Version_32 := 16#0F1BD6A1#;
u00033 : constant Version_32 := 16#7C25DE96#;
u00034 : constant Version_32 := 16#39ADFFA2#;
u00035 : constant Version_32 := 16#571DE3E7#;
u00036 : constant Version_32 := 16#5EB646AB#;
u00037 : constant Version_32 := 16#4249379B#;
u00038 : constant Version_32 := 16#0357E00A#;
u00039 : constant Version_32 := 16#3784FB72#;
u00040 : constant Version_32 := 16#2E723019#;
u00041 : constant Version_32 := 16#623358EA#;
u00042 : constant Version_32 := 16#107F9465#;
u00043 : constant Version_32 := 16#6843F68A#;
u00044 : constant Version_32 := 16#63305874#;
u00045 : constant Version_32 := 16#31E56CE1#;
u00046 : constant Version_32 := 16#02917970#;
u00047 : constant Version_32 := 16#6CCBA70E#;
u00048 : constant Version_32 := 16#41CD4204#;
u00049 : constant Version_32 := 16#572E3F58#;
u00050 : constant Version_32 := 16#20729FF5#;
u00051 : constant Version_32 := 16#1D4F93E8#;
u00052 : constant Version_32 := 16#30B2EC3D#;
u00053 : constant Version_32 := 16#34054F96#;
u00054 : constant Version_32 := 16#5A199860#;
u00055 : constant Version_32 := 16#0E7F912B#;
u00056 : constant Version_32 := 16#5760634A#;
u00057 : constant Version_32 := 16#5D851835#;

-- The following Export pragmas export the version numbers
-- with symbolic names ending in B (for body) or S
-- (for spec) so that they can be located in a link. The
-- information provided here is sufficient to track down
-- the exact versions of units used in a given build.

pragma Export (C, u00001, "helloB");
pragma Export (C, u00002, "system__standard_libraryB");
pragma Export (C, u00003, "system__standard_libraryS");
pragma Export (C, u00004, "adaS");
pragma Export (C, u00005, "ada__text_ioB");
pragma Export (C, u00006, "ada__text_ioS");
pragma Export (C, u00007, "ada__exceptionsB");
pragma Export (C, u00008, "ada__exceptionsS");
pragma Export (C, u00009, "gnatS");
pragma Export (C, u00010, "gnat__heap_sort_aB");
pragma Export (C, u00011, "gnat__heap_sort_aS");
pragma Export (C, u00012, "systemS");
pragma Export (C, u00013, "system__exception_tableB");

```



```

pragma Export (C, u00014, "system__exception_tableS");
pragma Export (C, u00015, "gnat__htableB");
pragma Export (C, u00016, "gnat__htableS");
pragma Export (C, u00017, "system__exceptionsS");
pragma Export (C, u00018, "system__machine_state_operationsB");
pragma Export (C, u00019, "system__machine_state_operationsS");
pragma Export (C, u00020, "system__machine_codeS");
pragma Export (C, u00021, "system__storage_elementsB");
pragma Export (C, u00022, "system__storage_elementsS");
pragma Export (C, u00023, "system__secondary_stackB");
pragma Export (C, u00024, "system__secondary_stackS");
pragma Export (C, u00025, "system__parametersB");
pragma Export (C, u00026, "system__parametersS");
pragma Export (C, u00027, "system__soft_linksB");
pragma Export (C, u00028, "system__soft_linksS");
pragma Export (C, u00029, "system__stack_checkingB");
pragma Export (C, u00030, "system__stack_checkingS");
pragma Export (C, u00031, "system__tracebackB");
pragma Export (C, u00032, "system__tracebackS");
pragma Export (C, u00033, "ada__streamsS");
pragma Export (C, u00034, "ada__tagsB");
pragma Export (C, u00035, "ada__tagsS");
pragma Export (C, u00036, "system__string_opsB");
pragma Export (C, u00037, "system__string_opsS");
pragma Export (C, u00038, "interfacesS");
pragma Export (C, u00039, "interfaces__c_streamsB");
pragma Export (C, u00040, "interfaces__c_streamsS");
pragma Export (C, u00041, "system__file_ioB");
pragma Export (C, u00042, "system__file_ioS");
pragma Export (C, u00043, "ada__finalizationB");
pragma Export (C, u00044, "ada__finalizationS");
pragma Export (C, u00045, "system__finalization_rootB");
pragma Export (C, u00046, "system__finalization_rootS");
pragma Export (C, u00047, "system__finalization_implementationB");
pragma Export (C, u00048, "system__finalization_implementationS");
pragma Export (C, u00049, "system__string_ops_concat_3B");
pragma Export (C, u00050, "system__string_ops_concat_3S");
pragma Export (C, u00051, "system__stream_attributesB");
pragma Export (C, u00052, "system__stream_attributesS");
pragma Export (C, u00053, "ada__io_exceptionsS");
pragma Export (C, u00054, "system__unsigned_typesS");
pragma Export (C, u00055, "system__file_control_blockS");
pragma Export (C, u00056, "ada__finalization__list_controllerB");
pragma Export (C, u00057, "ada__finalization__list_controllerS");

-- BEGIN ELABORATION ORDER
-- ada (spec)
-- gnat (spec)
-- gnat.heap_sort_a (spec)
-- gnat.heap_sort_a (body)
-- gnat.htable (spec)
-- gnat.htable (body)
-- interfaces (spec)
-- system (spec)
-- system.machine_code (spec)
-- system.parameters (spec)
-- system.parameters (body)
-- interfaces.c_streams (spec)

```

```

-- interfaces.c_streams (body)
-- system.standard_library (spec)
-- ada.exceptions (spec)
-- system.exception_table (spec)
-- system.exception_table (body)
-- ada.io_exceptions (spec)
-- system.exceptions (spec)
-- system.storage_elements (spec)
-- system.storage_elements (body)
-- system.machine_state_operations (spec)
-- system.machine_state_operations (body)
-- system.secondary_stack (spec)
-- system.stack_checking (spec)
-- system.soft_links (spec)
-- system.soft_links (body)
-- system.stack_checking (body)
-- system.secondary_stack (body)
-- system.standard_library (body)
-- system.string_ops (spec)
-- system.string_ops (body)
-- ada.tags (spec)
-- ada.tags (body)
-- ada.streams (spec)
-- system.finalization_root (spec)
-- system.finalization_root (body)
-- system.string_ops_concat_3 (spec)
-- system.string_ops_concat_3 (body)
-- system.traceback (spec)
-- system.traceback (body)
-- ada.exceptions (body)
-- system.unsigned_types (spec)
-- system.stream_attributes (spec)
-- system.stream_attributes (body)
-- system.finalization_implementation (spec)
-- system.finalization_implementation (body)
-- ada.finalization (spec)
-- ada.finalization (body)
-- ada.finalization.list_controller (spec)
-- ada.finalization.list_controller (body)
-- system.file_control_block (spec)
-- system.file_io (spec)
-- system.file_io (body)
-- ada.text_io (spec)
-- ada.text_io (body)
-- hello (body)
-- END ELABORATION ORDER

end ada_main;

-- The following source file name pragmas allow the generated file
-- names to be unique for different main programs. They are needed
-- since the package name will always be Ada_Main.

pragma Source_File_Name (ada_main, Spec_File_Name => "B~HELLO.ADS");
pragma Source_File_Name (ada_main, Body_File_Name => "B~HELLO.ADB");

-- Generated package body for Ada_Main starts here

```

```

package body ada_main is

  -- The actual finalization is performed by calling the
  -- library routine in System.Standard_Library.Adafinal

  procedure Do_Finalize;
  pragma Import (C, Do_Finalize, "system__standard_library__adafinal");

  -----
  -- adainit --
  -----

  procedure adainit is

    -- These booleans are set to True once the associated unit has
    -- been elaborated. It is also used to avoid elaborating the
    -- same unit twice.

    E040 : Boolean; pragma Import (Ada, E040, "interfaces__c_streams_E");
    E008 : Boolean; pragma Import (Ada, E008, "ada__exceptions_E");
    E014 : Boolean; pragma Import (Ada, E014, "system__exception_table_E");
    E053 : Boolean; pragma Import (Ada, E053, "ada__io_exceptions_E");
    E017 : Boolean; pragma Import (Ada, E017, "system__exceptions_E");
    E024 : Boolean; pragma Import (Ada, E024, "system__secondary_stack_E");
    E030 : Boolean; pragma Import (Ada, E030, "system__stack_checking_E");
    E028 : Boolean; pragma Import (Ada, E028, "system__soft_links_E");
    E035 : Boolean; pragma Import (Ada, E035, "ada__tags_E");
    E033 : Boolean; pragma Import (Ada, E033, "ada__streams_E");
    E046 : Boolean; pragma Import (Ada, E046, "system__finalization_root_E");
    E048 : Boolean; pragma Import (Ada, E048, "system__finalization_implementation_E");
    E044 : Boolean; pragma Import (Ada, E044, "ada__finalization_E");
    E057 : Boolean; pragma Import (Ada, E057, "ada__finalization_list_controller_E");
    E055 : Boolean; pragma Import (Ada, E055, "system__file_control_block_E");
    E042 : Boolean; pragma Import (Ada, E042, "system__file_io_E");
    E006 : Boolean; pragma Import (Ada, E006, "ada__text_io_E");

    -- Set_Globals is a library routine that stores away the
    -- value of the indicated set of global values in global
    -- variables within the library.

    procedure Set_Globals
      (Main_Priority           : Integer;
       Time_Slice_Value       : Integer;
       WC_Encoding             : Character;
       Locking_Policy          : Character;
       Queuing_Policy          : Character;
       Task_Dispatching_Policy : Character;
       Adafinal                : System.Address;
       Unreserve_All_Interrupts : Integer;
       Exception_Tracebacks    : Integer);
    pragma Import (C, Set_Globals, "__gnat_set_globals");

    -- SDP_Table_Build is a library routine used to build the
    -- exception tables. See unit Ada.Exceptions in files
    -- A-EXCEPT.ADS/adb for full details of how zero cost
    -- exception handling works. This procedure, the call to
    -- it, and the two following tables are all omitted if the
    -- build is in longjmp/setjump exception mode.

```

```

procedure SDP_Table_Build
  (SDP_Addresses   : System.Address;
   SDP_Count       : Natural;
   Elab_Addresses  : System.Address;
   Elab_Addr_Count : Natural);
pragma Import (C, SDP_Table_Build, "__gnat_SDP_Table_Build");

-- Table of Unit_Exception_Table addresses. Used for zero
-- cost exception handling to build the top level table.

ST : aliased constant array (1 .. 23) of System.Address := (
  Hello'UET_Address,
  Ada.Text_Io'UET_Address,
  Ada.Exceptions'UET_Address,
  Gnat.Heap_Sort_A'UET_Address,
  System.Exception_Table'UET_Address,
  System.Machine_State_Operations'UET_Address,
  System.Secondary_Stack'UET_Address,
  System.Parameters'UET_Address,
  System.Soft_Links'UET_Address,
  System.Stack_Checking'UET_Address,
  System.Traceback'UET_Address,
  Ada.Streams'UET_Address,
  Ada.Tags'UET_Address,
  System.String_Ops'UET_Address,
  Interfaces.C.Streams'UET_Address,
  System.File_Io'UET_Address,
  Ada.Finalization'UET_Address,
  System.Finalization_Root'UET_Address,
  System.Finalization_Implementation'UET_Address,
  System.String_Ops_Concat_3'UET_Address,
  System.Stream_Attributes'UET_Address,
  System.File_Control_Block'UET_Address,
  Ada.Finalization.List_Controller'UET_Address);

-- Table of addresses of elaboration routines. Used for
-- zero cost exception handling to make sure these
-- addresses are included in the top level procedure
-- address table.

EA : aliased constant array (1 .. 23) of System.Address := (
  adainit'Code_Address,
  Do_Finalize'Code_Address,
  Ada.Exceptions'Elab_Spec'Address,
  System.Exceptions'Elab_Spec'Address,
  Interfaces.C.Streams'Elab_Spec'Address,
  System.Exception_Table'Elab_Body'Address,
  Ada.Io_Exceptions'Elab_Spec'Address,
  System.Stack_Checking'Elab_Spec'Address,
  System.Soft_Links'Elab_Body'Address,
  System.Secondary_Stack'Elab_Body'Address,
  Ada.Tags'Elab_Spec'Address,
  Ada.Tags'Elab_Body'Address,
  Ada.Streams'Elab_Spec'Address,
  System.Finalization_Root'Elab_Spec'Address,
  Ada.Exceptions'Elab_Body'Address,
  System.Finalization_Implementation'Elab_Spec'Address,

```

```

    System.Finalization_Implementation'Elab_Body'Address,
    Ada.Finalization'Elab_Spec'Address,
    Ada.Finalization.List_Controller'Elab_Spec'Address,
    System.File_Control_Block'Elab_Spec'Address,
    System.File_Io'Elab_Body'Address,
    Ada.Text_Io'Elab_Spec'Address,
    Ada.Text_Io'Elab_Body'Address);

-- Start of processing for adainit

begin

    -- Call SDP_Table_Build to build the top level procedure
    -- table for zero cost exception handling (omitted in
    -- longjmp/setjump mode).

    SDP_Table_Build (ST'Address, 23, EA'Address, 23);

    -- Call Set_Globals to record various information for
    -- this partition. The values are derived by the binder
    -- from information stored in the ali files by the compiler.

    Set_Globals
    (Main_Priority          => -1,
      -- Priority of main program, -1 if no pragma Priority used

      Time_Slice_Value      => -1,
      -- Time slice from Time_Slice pragma, -1 if none used

      WC_Encoding           => 'b',
      -- Wide_Character encoding used, default is brackets

      Locking_Policy        => ' ',
      -- Locking_Policy used, default of space means not
      -- specified, otherwise it is the first character of
      -- the policy name.

      Queuing_Policy        => ' ',
      -- Queuing_Policy used, default of space means not
      -- specified, otherwise it is the first character of
      -- the policy name.

      Task_Dispatching_Policy => ' ',
      -- Task_Dispatching_Policy used, default of space means
      -- not specified, otherwise first character of the
      -- policy name.

      Adafinal              => System.Null_Address,
      -- Address of Adafinal routine, not used anymore

      Unreserve_All_Interrupts => 0,
      -- Set true if pragma Unreserve_All_Interrupts was used

      Exception_Tracebacks  => 0);
      -- Indicates if exception tracebacks are enabled

    Elab_Final_Code := 1;

```

```
-- Now we have the elaboration calls for all units in the partition.
-- The Elab_Spec and Elab_Body attributes generate references to the
-- implicit elaboration procedures generated by the compiler for
-- each unit that requires elaboration.
```

```
if not E040 then
    Interfaces.C.Streams'Elab_Spec;
end if;
E040 := True;
if not E008 then
    Ada.Exceptions'Elab_Spec;
end if;
if not E014 then
    System.Exception_Table'Elab_Body;
    E014 := True;
end if;
if not E053 then
    Ada.Io_Exceptions'Elab_Spec;
    E053 := True;
end if;
if not E017 then
    System.Exceptions'Elab_Spec;
    E017 := True;
end if;
if not E030 then
    System.Stack_Checking'Elab_Spec;
end if;
if not E028 then
    System.Soft_Links'Elab_Body;
    E028 := True;
end if;
E030 := True;
if not E024 then
    System.Secondary_Stack'Elab_Body;
    E024 := True;
end if;
if not E035 then
    Ada.Tags'Elab_Spec;
end if;
if not E035 then
    Ada.Tags'Elab_Body;
    E035 := True;
end if;
if not E033 then
    Ada.Streams'Elab_Spec;
    E033 := True;
end if;
if not E046 then
    System.Finalization_Root'Elab_Spec;
end if;
E046 := True;
if not E008 then
    Ada.Exceptions'Elab_Body;
    E008 := True;
end if;
if not E048 then
    System.Finalization_Implementation'Elab_Spec;
end if;
```

```

    if not E048 then
        System.Finalization_Implementation'Elab_Body;
        E048 := True;
    end if;
    if not E044 then
        Ada.Finalization'Elab_Spec;
    end if;
    E044 := True;
    if not E057 then
        Ada.Finalization.List_Controller'Elab_Spec;
    end if;
    E057 := True;
    if not E055 then
        System.File_Control_Block'Elab_Spec;
        E055 := True;
    end if;
    if not E042 then
        System.File_Io'Elab_Body;
        E042 := True;
    end if;
    if not E006 then
        Ada.Text_Io'Elab_Spec;
    end if;
    if not E006 then
        Ada.Text_Io'Elab_Body;
        E006 := True;
    end if;

    Elab_Final_Code := 0;
end adainit;

-----
-- adafinal --
-----

procedure adafinal is
begin
    Do_Finalize;
end adafinal;

-----
-- main --
-----

-- main is actually a function, as in the ANSI C standard,
-- defined to return the exit status. The three parameters
-- are the argument count, argument values and environment
-- pointer.

function main
    (argc : Integer;
     argv : System.Address;
     envp : System.Address)
    return Integer
is
    -- The initialize routine performs low level system
    -- initialization using a standard library routine which
    -- sets up signal handling and performs any other

```

```

-- required setup. The routine can be found in file
-- A-INIT.C.

procedure initialize;
pragma Import (C, initialize, "__gnat_initialize");

-- The finalize routine performs low level system
-- finalization using a standard library routine. The
-- routine is found in file A-FINAL.C and in the standard
-- distribution is a dummy routine that does nothing, so
-- really this is a hook for special user finalization.

procedure finalize;
pragma Import (C, finalize, "__gnat_finalize");

-- We get to the main program of the partition by using
-- pragma Import because if we try to with the unit and
-- call it Ada style, then not only do we waste time
-- recompiling it, but also, we don't really know the right
-- qualifiers (e.g. identifier character set) to be used
-- to compile it.

procedure Ada_Main_Program;
pragma Import (Ada, Ada_Main_Program, "_ada_hello");

-- Start of processing for main

begin
  -- Save global variables

  gnat_argc := argc;
  gnat_argv := argv;
  gnat_envp := envp;

  -- Call low level system initialization

  Initialize;

  -- Call our generated Ada initialization routine

  adainit;

  -- This is the point at which we want the debugger to get
  -- control

  Break_Start;

  -- Now we call the main program of the partition

  Ada_Main_Program;

  -- Perform Ada finalization

  adafinal;

  -- Perform low level system finalization

  Finalize;

```



```

    -- Return the proper exit status
    return (gnat_exit_status);
end;

-- This section is entirely comments, so it has no effect on the
-- compilation of the Ada_Main package. It provides the list of
-- object files and linker options, as well as some standard
-- libraries needed for the link. The GNAT LINK utility parses
-- this B~HELLO.ADB file to read these comment lines to generate
-- the appropriate command line arguments for the call to the
-- system linker. The BEGIN/END lines are used for sentinels for
-- this parsing operation.

-- The exact file names will of course depend on the environment,
-- host/target and location of files on the host system.

-- BEGIN Object file/option list
--  ./HELLO.OBJ
--  -L./
--  -L/usr/local/gnat/lib/gcc-lib/i686-pc-linux-gnu/2.8.1/adalib/
--  /usr/local/gnat/lib/gcc-lib/i686-pc-linux-gnu/2.8.1/adalib/libgnat.a
-- END Object file/option list

end ada_main;

```

The Ada code in the above example is exactly what is generated by the binder. We have added comments to more clearly indicate the function of each part of the generated `Ada_Main` package.

The code is standard Ada in all respects, and can be processed by any tools that handle Ada. In particular, it is possible to use the debugger in Ada mode to debug the generated `Ada_Main` package. For example, suppose that for reasons that you do not understand, your program is blowing up during elaboration of the body of `Ada.Text_IO`. To chase this bug down, you can place a breakpoint on the call:

```
Ada.Text_IO'Elab_Body;
```

and trace the elaboration routine for this package to find out where the problem might be (more usually of course you would be debugging elaboration code in your own application).

4.2 Generating the Binder Program in C

In most normal usage, the default mode of GNAT BIND which is to generate the main package in Ada, as described in the previous section. In particular, this means that any Ada programmer can read and understand the generated main program. It can also be debugged just like any other Ada code provided the `-g` qualifier is used for GNAT BIND and GNAT LINK.

However for some purposes it may be convenient to generate the main program in C rather than Ada. This may for example be helpful when you are generating a mixed language program with the main program in C. The GNAT compiler itself is an example. The use of the `-C` qualifier for both GNAT BIND and GNAT LINK will cause the program to be generated in C (and compiled using the gnu C compiler). The following shows the C code generated for the same "Hello World" program:

```

#ifdef __STDC__
#define PARAMS(paramlist) paramlist
#else
#define PARAMS(paramlist) ()
#endif

extern void __gnat_set_globals
  PARAMS ((int, int, int, int, int, int,
           void (*) PARAMS ((void)), int, int));
extern void adafinal PARAMS ((void));
extern void adainit PARAMS ((void));
extern void system__standard_library__adafinal PARAMS ((void));
extern int main PARAMS ((int, char **, char **));
extern void exit PARAMS ((int));
extern void __gnat_break_start PARAMS ((void));
extern void _ada_hello PARAMS ((void));
extern void __gnat_initialize PARAMS ((void));
extern void __gnat_finalize PARAMS ((void));

extern void ada__exceptions___elabs PARAMS ((void));
extern void system__exceptions___elabs PARAMS ((void));
extern void interfaces__c_streams___elabs PARAMS ((void));
extern void system__exception_table___elabb PARAMS ((void));
extern void ada__io_exceptions___elabs PARAMS ((void));
extern void system__stack_checking___elabs PARAMS ((void));
extern void system__soft_links___elabb PARAMS ((void));
extern void system__secondary_stack___elabb PARAMS ((void));
extern void ada__tags___elabs PARAMS ((void));
extern void ada__tags___elabb PARAMS ((void));
extern void ada__streams___elabs PARAMS ((void));
extern void system__finalization_root___elabs PARAMS ((void));
extern void ada__exceptions___elabb PARAMS ((void));
extern void system__finalization_implementation___elabs PARAMS ((void));
extern void system__finalization_implementation___elabb PARAMS ((void));
extern void ada__finalization___elabs PARAMS ((void));
extern void ada__finalization__list_controller___elabs PARAMS ((void));
extern void system__file_control_block___elabs PARAMS ((void));
extern void system__file_io___elabb PARAMS ((void));
extern void ada__text_io___elabs PARAMS ((void));
extern void ada__text_io___elabb PARAMS ((void));

extern int __gnat_inside_elab_final_code;

extern int gnat_argc;
extern char **gnat_argv;
extern char **gnat_envp;
extern int gnat_exit_status;

char __gnat_version[] = "GNAT Version: 3.15w (20010315)";
void adafinal () {
  system__standard_library__adafinal ();
}

void adainit ()
{
  extern char ada__exceptions_E;
  extern char system__exceptions_E;

```

```

extern char interfaces__c_streams_E;
extern char system__exception_table_E;
extern char ada__io_exceptions_E;
extern char system__secondary_stack_E;
extern char system__stack_checking_E;
extern char system__soft_links_E;
extern char ada__tags_E;
extern char ada__streams_E;
extern char system__finalization_root_E;
extern char system__finalization_implementation_E;
extern char ada__finalization_E;
extern char ada__finalization__list_controller_E;
extern char system__file_control_block_E;
extern char system__file_io_E;
extern char ada__text_io_E;

extern void *__gnat_hello__SDP;
extern void *__gnat_ada__text_io__SDP;
extern void *__gnat_ada__exceptions__SDP;
extern void *__gnat_gnat__heap_sort_a__SDP;
extern void *__gnat_system__exception_table__SDP;
extern void *__gnat_system__machine_state_operations__SDP;
extern void *__gnat_system__secondary_stack__SDP;
extern void *__gnat_system__parameters__SDP;
extern void *__gnat_system__soft_links__SDP;
extern void *__gnat_system__stack_checking__SDP;
extern void *__gnat_system__traceback__SDP;
extern void *__gnat_ada__streams__SDP;
extern void *__gnat_ada__tags__SDP;
extern void *__gnat_system__string_ops__SDP;
extern void *__gnat_interfaces__c_streams__SDP;
extern void *__gnat_system__file_io__SDP;
extern void *__gnat_ada__finalization__SDP;
extern void *__gnat_system__finalization_root__SDP;
extern void *__gnat_system__finalization_implementation__SDP;
extern void *__gnat_system__string_ops_concat_3__SDP;
extern void *__gnat_system__stream_attributes__SDP;
extern void *__gnat_system__file_control_block__SDP;
extern void *__gnat_ada__finalization__list_controller__SDP;

void **st[23] = {
    &__gnat_hello__SDP,
    &__gnat_ada__text_io__SDP,
    &__gnat_ada__exceptions__SDP,
    &__gnat_gnat__heap_sort_a__SDP,
    &__gnat_system__exception_table__SDP,
    &__gnat_system__machine_state_operations__SDP,
    &__gnat_system__secondary_stack__SDP,
    &__gnat_system__parameters__SDP,
    &__gnat_system__soft_links__SDP,
    &__gnat_system__stack_checking__SDP,
    &__gnat_system__traceback__SDP,
    &__gnat_ada__streams__SDP,
    &__gnat_ada__tags__SDP,
    &__gnat_system__string_ops__SDP,
    &__gnat_interfaces__c_streams__SDP,
    &__gnat_system__file_io__SDP,
    &__gnat_ada__finalization__SDP,

```

```

    &__gnat_system__finalization_root__SDP,
    &__gnat_system__finalization_implementation__SDP,
    &__gnat_system__string_ops_concat_3__SDP,
    &__gnat_system__stream_attributes__SDP,
    &__gnat_system__file_control_block__SDP,
    &__gnat_ada__finalization__list_controller__SDP};

extern void ada__exceptions___elabs ();
extern void system__exceptions___elabs ();
extern void interfaces__c_streams___elabs ();
extern void system__exception_table___elabb ();
extern void ada__io_exceptions___elabs ();
extern void system__stack_checking___elabs ();
extern void system__soft_links___elabb ();
extern void system__secondary_stack___elabb ();
extern void ada__tags___elabs ();
extern void ada__tags___elabb ();
extern void ada__streams___elabs ();
extern void system__finalization_root___elabs ();
extern void ada__exceptions___elabb ();
extern void system__finalization_implementation___elabs ();
extern void system__finalization_implementation___elabb ();
extern void ada__finalization___elabs ();
extern void ada__finalization__list_controller___elabs ();
extern void system__file_control_block___elabs ();
extern void system__file_io___elabb ();
extern void ada__text_io___elabs ();
extern void ada__text_io___elabb ();

void (*ea[23]) () = {
    adainit,
    system__standard_library__adafinal,
    ada__exceptions___elabs,
    system__exceptions___elabs,
    interfaces__c_streams___elabs,
    system__exception_table___elabb,
    ada__io_exceptions___elabs,
    system__stack_checking___elabs,
    system__soft_links___elabb,
    system__secondary_stack___elabb,
    ada__tags___elabs,
    ada__tags___elabb,
    ada__streams___elabs,
    system__finalization_root___elabs,
    ada__exceptions___elabb,
    system__finalization_implementation___elabs,
    system__finalization_implementation___elabb,
    ada__finalization___elabs,
    ada__finalization__list_controller___elabs,
    system__file_control_block___elabs,
    system__file_io___elabb,
    ada__text_io___elabs,
    ada__text_io___elabb};

__gnat_SDP_Table_Build (&st, 23, ea, 23);
__gnat_set_globals (
    -1,      /* Main_Priority          */
    -1,      /* Time_Slice_Value          */

```

```

    'b',      /* WC_Encoding          */
    ' ',      /* Locking_Policy          */
    ' ',      /* Queuing_Policy          */
    ' ',      /* Tasking_Dispatching_Policy */
    0,        /* Finalization routine address, not used anymore */
    0,        /* Unreserve_All_Interrupts */
    0);       /* Exception_Tracebacks */

__gnat_inside_elab_final_code = 1;

if (ada__exceptions_E == 0) {
    ada__exceptions___elabs ();
}
if (system__exceptions_E == 0) {
    system__exceptions___elabs ();
    system__exceptions_E++;
}
if (interfaces__c_streams_E == 0) {
    interfaces__c_streams___elabs ();
}
interfaces__c_streams_E = 1;
if (system__exception_table_E == 0) {
    system__exception_table___elabb ();
    system__exception_table_E++;
}
if (ada__io_exceptions_E == 0) {
    ada__io_exceptions___elabs ();
    ada__io_exceptions_E++;
}
if (system__stack_checking_E == 0) {
    system__stack_checking___elabs ();
}
if (system__soft_links_E == 0) {
    system__soft_links___elabb ();
    system__soft_links_E++;
}
system__stack_checking_E = 1;
if (system__secondary_stack_E == 0) {
    system__secondary_stack___elabb ();
    system__secondary_stack_E++;
}
if (ada__tags_E == 0) {
    ada__tags___elabs ();
}
if (ada__tags_E == 0) {
    ada__tags___elabb ();
    ada__tags_E++;
}
if (ada__streams_E == 0) {
    ada__streams___elabs ();
    ada__streams_E++;
}
if (system__finalization_root_E == 0) {
    system__finalization_root___elabs ();
}
system__finalization_root_E = 1;
if (ada__exceptions_E == 0) {
    ada__exceptions___elabb ();
}

```

```

    ada__exceptions_E++;
}
if (system__finalization_implementation_E == 0) {
    system__finalization_implementation___elabs ();
}
if (system__finalization_implementation_E == 0) {
    system__finalization_implementation___elabb ();
    system__finalization_implementation_E++;
}
if (ada__finalization_E == 0) {
    ada__finalization___elabs ();
}
ada__finalization_E = 1;
if (ada__finalization__list_controller_E == 0) {
    ada__finalization__list_controller___elabs ();
}
ada__finalization__list_controller_E = 1;
if (system__file_control_block_E == 0) {
    system__file_control_block___elabs ();
    system__file_control_block_E++;
}
if (system__file_io_E == 0) {
    system__file_io___elabb ();
    system__file_io_E++;
}
if (ada__text_io_E == 0) {
    ada__text_io___elabs ();
}
if (ada__text_io_E == 0) {
    ada__text_io___elabb ();
    ada__text_io_E++;
}

__gnat_inside_elab_final_code = 0;
}
int main (argc, argv, envp)
    int argc;
    char **argv;
    char **envp;
{
    gnat_argc = argc;
    gnat_argv = argv;
    gnat_envp = envp;

    __gnat_initialize ();
    adainit ();
    __gnat_break_start ();

    _ada_hello ();

    system__standard_library__adafinal ();
    __gnat_finalize ();
    exit (gnat_exit_status);
}
unsigned helloB = 0x7880BEB3;
unsigned system__standard_libraryB = 0x0D24CBD0;
unsigned system__standard_libraryS = 0x3283DBEB;
unsigned adaS = 0x2359F9ED;

```

```

unsigned ada__text_ioB = 0x47C85FC4;
unsigned ada__text_ioS = 0x496FE45C;
unsigned ada__exceptionsB = 0x74F50187;
unsigned ada__exceptionsS = 0x6736945B;
unsigned gnatS = 0x156A40CF;
unsigned gnat__heap_sort_aB = 0x033DABE0;
unsigned gnat__heap_sort_aS = 0x6AB38FEA;
unsigned systemS = 0x0331C6FE;
unsigned system__exceptionsS = 0x20C9ECA4;
unsigned system__exception_tableB = 0x68A22947;
unsigned system__exception_tableS = 0x394BADD5;
unsigned gnat__htableB = 0x08258E1B;
unsigned gnat__htableS = 0x367D5222;
unsigned system__machine_state_operationsB = 0x4F3B7492;
unsigned system__machine_state_operationsS = 0x182F5CF4;
unsigned system__storage_elementsB = 0x2F1EB794;
unsigned system__storage_elementsS = 0x102C83C7;
unsigned system__secondary_stackB = 0x1574B6E9;
unsigned system__secondary_stackS = 0x708E260A;
unsigned system__parametersB = 0x56D770CD;
unsigned system__parametersS = 0x237E39BE;
unsigned system__soft_linksB = 0x08AB6B2C;
unsigned system__soft_linksS = 0x1E2491F3;
unsigned system__stack_checkingB = 0x476457A0;
unsigned system__stack_checkingS = 0x5299FCED;
unsigned system__tracebackB = 0x2971EBDE;
unsigned system__tracebackS = 0x2E9C3122;
unsigned ada__streamsS = 0x7C25DE96;
unsigned ada__tagsB = 0x39ADFFA2;
unsigned ada__tagsS = 0x769A0464;
unsigned system__string_opsB = 0x5EB646AB;
unsigned system__string_opsS = 0x63CED018;
unsigned interfacesS = 0x0357E00A;
unsigned interfaces__c_streamsB = 0x3784FB72;
unsigned interfaces__c_streamsS = 0x2E723019;
unsigned system__file_ioB = 0x623358EA;
unsigned system__file_ioS = 0x31F873E6;
unsigned ada__finalizationB = 0x6843F68A;
unsigned ada__finalizationS = 0x63305874;
unsigned system__finalization_rootB = 0x31E56CE1;
unsigned system__finalization_rootS = 0x23169EF3;
unsigned system__finalization_implementationB = 0x6CCBA70E;
unsigned system__finalization_implementationS = 0x604AA587;
unsigned system__string_ops_concat_3B = 0x572E3F58;
unsigned system__string_ops_concat_3S = 0x01F57876;
unsigned system__stream_attributesB = 0x1D4F93E8;
unsigned system__stream_attributesS = 0x30B2EC3D;
unsigned ada__io_exceptionsS = 0x34054F96;
unsigned system__unsigned_typesS = 0x7B9E7FE3;
unsigned system__file_control_blockS = 0x2FF876A8;
unsigned ada__finalization__list_controllerB = 0x5760634A;
unsigned ada__finalization__list_controllerS = 0x5D851835;

/* BEGIN ELABORATION ORDER
ada (spec)
gnat (spec)
gnat.heap_sort_a (spec)
gnat.htable (spec)

```

```

gnat.htable (body)
interfaces (spec)
system (spec)
system.parameters (spec)
system.standard_library (spec)
ada.exceptions (spec)
system.exceptions (spec)
system.parameters (body)
gnat.heap_sort_a (body)
interfaces.c_streams (spec)
interfaces.c_streams (body)
system.exception_table (spec)
system.exception_table (body)
ada.io_exceptions (spec)
system.storage_elements (spec)
system.storage_elements (body)
system.machine_state_operations (spec)
system.machine_state_operations (body)
system.secondary_stack (spec)
system.stack_checking (spec)
system.soft_links (spec)
system.soft_links (body)
system.stack_checking (body)
system.secondary_stack (body)
system.standard_library (body)
system.string_ops (spec)
system.string_ops (body)
ada.tags (spec)
ada.tags (body)
ada.streams (spec)
system.finalization_root (spec)
system.finalization_root (body)
system.string_ops_concat_3 (spec)
system.string_ops_concat_3 (body)
system.traceback (spec)
system.traceback (body)
ada.exceptions (body)
system.unsigned_types (spec)
system.stream_attributes (spec)
system.stream_attributes (body)
system.finalization_implementation (spec)
system.finalization_implementation (body)
ada.finalization (spec)
ada.finalization (body)
ada.finalization.list_controller (spec)
ada.finalization.list_controller (body)
system.file_control_block (spec)
system.file_io (spec)
system.file_io (body)
ada.text_io (spec)
ada.text_io (body)
hello (body)
    END ELABORATION ORDER */

/* BEGIN Object file/option list
./HELLO.OBJ
-L./
-L/usr/local/gnat/lib/gcc-lib/alpha-dec-osf5.1/2.8.1/adalib/

```



```

/usr/local/gnat/lib/gcc-lib/alpha-dec-osf5.1/2.8.1/adalib/libgnat.a
-lexc
END Object file/option list */

```

Here again, the C code is exactly what is generated by the binder. The functions of the various parts of this code correspond in an obvious manner with the commented Ada code shown in the example in the previous section.

4.3 Consistency-Checking Modes

As described in the previous section, by default GNAT BIND checks that object files are consistent with one another and are consistent with any source files it can locate. The following qualifiers control binder access to sources.

/READ_SOURCES=ALL

Require source files to be present. In this mode, the binder must be able to locate all source files that are referenced, in order to check their consistency. In normal mode, if a source file cannot be located it is simply ignored. If you specify this qualifier, a missing source file is an error.

/READ_SOURCES=NONE

Exclude source files. In this mode, the binder only checks that ALI files are consistent with one another. Source files are not accessed. The binder runs faster in this mode, and there is still a guarantee that the resulting program is self-consistent. If a source file has been edited since it was last compiled, and you specify this qualifier, the binder will not detect that the object file is out of date with respect to the source file. Note that this is the mode that is automatically used by GNAT MAKE because in this case the checking against sources has already been performed by GNAT MAKE in the course of compilation (i.e. before binding).

/READ_SOURCES=AVAILABLE

This is the default mode in which source files are checked if they are available, and ignored if they are not available.

4.4 Binder Error Message Control

The following qualifiers provide control over the generation of error messages from the binder:

/REPORT_ERRORS=VERBOSE

Verbose mode. In the normal mode, brief error messages are generated to 'SYS\$ERROR'. If this qualifier is present, a header is written to 'SYS\$OUTPUT' and any error messages are directed to 'SYS\$OUTPUT'. All that is written to 'SYS\$ERROR' is a brief summary message.

/REPORT_ERRORS=BRIEF

Generate brief error messages to 'SYS\$ERROR' even if verbose mode is specified. This is relevant only when used with the /REPORT_ERRORS=VERBOSE qualifier.

`/WARNINGS=SUPPRESS`

Suppress all warning messages.

`/WARNINGS=ERROR`

Treat any warning messages as fatal errors.

`/WARNINGS=NORMAL`

Standard mode with warnings generated, but warnings do not get treated as errors.

`/NOTIME_STAMP_CHECK`

The binder performs a number of consistency checks including:

- Check that time stamps of a given source unit are consistent
- Check that checksums of a given source unit are consistent
- Check that consistent versions of GNAT were used for compilation
- Check consistency of configuration pragmas as required

Normally failure of such checks, in accordance with the consistency requirements of the Ada Reference Manual, causes error messages to be generated which abort the binder and prevent the output of a binder file and subsequent link to obtain an executable.

The `/NOTIME_STAMP_CHECK` qualifier converts these error messages into warnings, so that binding and linking can continue to completion even in the presence of such errors. The result may be a failed link (due to missing symbols), or a non-functional executable which has undefined semantics. *This means that `/NOTIME_STAMP_CHECK` should be used only in unusual situations, with extreme care.*

4.5 Elaboration Control

The following qualifiers provide additional control over the elaboration order. For full details see See Chapter 11 [Elaboration Order Handling in GNAT], page 141.

`/PESSIMISTIC_ELABORATION`

Normally the binder attempts to choose an elaboration order that is likely to minimize the likelihood of an elaboration order error resulting in raising a `Program_Error` exception. This qualifier reverses the action of the binder, and requests that it deliberately choose an order that is likely to maximize the likelihood of an elaboration error. This is useful in ensuring portability and avoiding dependence on accidental fortuitous elaboration ordering.

Normally it only makes sense to use the `-p` qualifier if dynamic elaboration checking is used (`/CHECKS=ELABORATION` qualifier used for compilation). This is because in the default static elaboration mode, all necessary `Elaborate_All` pragmas are implicitly inserted. These implicit pragmas are still respected by the binder in `-p` mode, so a safe elaboration order is assured.

4.6 Output Control

The following qualifiers allow additional control over the output generated by the binder.

/BIND_FILE=ADA

Generate binder program in Ada (default). The binder program is named ‘B\$mainprog.ADB’ by default. This can be changed with `-o GNAT BIND` option.

/NOOUTPUT

Check only. Do not generate the binder output file. In this mode the binder performs all error checks but does not generate an output file.

/BIND_FILE=C

Generate binder program in C. The binder program is named ‘B_mainprog.C’. This can be changed with `-o GNAT BIND` option.

/ELABORATION_DEPENDENCIES

Output complete list of elaboration-order dependencies, showing the reason for each dependency. This output can be rather extensive but may be useful in diagnosing problems with elaboration order. The output is written to ‘SYS\$OUTPUT’.

/HELP

Output usage information. The output is written to ‘SYS\$OUTPUT’.

/LINKER_OPTION_LIST

Output linker options to ‘SYS\$OUTPUT’. Includes library search paths, contents of pragmas `Ident` and `Linker_Options`, and libraries added by GNAT BIND.

/ORDER_OF_ELABORATION

Output chosen elaboration order. The output is written to ‘SYS\$OUTPUT’.

/OBJECT_LIST

Output full names of all the object files that must be linked to provide the Ada component of the program. The output is written to ‘SYS\$OUTPUT’. This list includes the files explicitly supplied and referenced by the user as well as implicitly referenced run-time unit files. The latter are omitted if the corresponding units reside in shared libraries. The directory names for the run-time units depend on the system configuration.

/OUTPUT=file

Set name of output file to *file* instead of the normal ‘B\$mainprog.ADB’ default. Note that *file* denote the Ada binder generated body filename. In C mode you would normally give *file* an extension of ‘.C’ because it will be a C source program. Note that if this option is used, then linking must be done manually. It is not possible to use GNAT LINK in this case, since it cannot locate the binder file.

/RESTRICTION_LIST

Generate list of `pragma Restrictions` that could be applied to the current unit. This is useful for code audit purposes, and also may be used to improve code generation in some cases.

4.7 Binding with Non-Ada Main Programs

In our description so far we have assumed that the main program is in Ada, and that the task of the binder is to generate a corresponding function `main` that invokes this Ada main program. GNAT also supports the building of executable programs where the main program is not in Ada, but some of the called routines are written in Ada and compiled using GNAT (see Section 2.10 [Mixed Language Programming], page 20). The following qualifier is used in this situation:

`/NOMAIN` No main program. The main program is not in Ada.

In this case, most of the functions of the binder are still required, but instead of generating a main program, the binder generates a file containing the following callable routines:

adainit You must call this routine to initialize the Ada part of the program by calling the necessary elaboration routines. A call to **adainit** is required before the first call to an Ada subprogram.

Note that it is assumed that the basic execution environment must be setup to be appropriate for Ada execution at the point where the first Ada subprogram is called. In particular, if the Ada code will do any floating-point operations, then the FPU must be setup in an appropriate manner. For the case of the x86, for example, full precision mode is required. The procedure `GNAT.Float_Control.Reset` may be used to ensure that the FPU is in the right state.

adafinal You must call this routine to perform any library-level finalization required by the Ada subprograms. A call to **adafinal** is required after the last call to an Ada subprogram, and before the program terminates.

If the `/NOMAIN` qualifier is given, more than one ALI file may appear on the command line for `GNAT BIND`. The normal *closure* calculation is performed for each of the specified units. Calculating the closure means finding out the set of units involved by tracing `with` references. The reason it is necessary to be able to specify more than one ALI file is that a given program may invoke two or more quite separate groups of Ada units.

The binder takes the name of its output file from the last specified ALI file, unless overridden by the use of the `/OUTPUT=file`. The output is an Ada unit in source form that can be compiled with GNAT unless the `-C` qualifier is used in which case the output is a C source file, which must be compiled using the C compiler. This compilation occurs automatically as part of the `GNAT LINK` processing.

Currently the GNAT run time requires a FPU using 80 bits mode precision. Under targets where this is not the default it is required to call `GNAT.Float_Control.Reset` before using floating point numbers (this include float computation, float input and output) in the Ada code. A side effect is that this could be the wrong mode for the foreign code where floating point computation could be broken after this call.

4.8 Binding Programs with No Main Subprogram

It is possible to have an Ada program which does not have a main subprogram. This program will call the elaboration routines of all the packages, then the finalization routines.

The following qualifier is used to bind programs organized in this manner:

`/ZERO_MAIN`

Normally the binder checks that the unit name given on the command line corresponds to a suitable main subprogram. When this qualifier is used, a list of ALI files can be given, and the execution of the program consists of elaboration of these units in an appropriate order.

4.9 Summary of Binder Qualifiers

The following are the qualifiers available with GNAT BIND:

`/OBJECT_SEARCH`

Specify directory to be searched for ALI files.

`/SOURCE_SEARCH`

Specify directory to be searched for source file.

`/BIND_FILE=ADA`

Generate binder program in Ada (default)

`/REPORT_ERRORS=BRIEF`

Generate brief messages to ‘SYS\$ERROR’ even if verbose mode set.

`/NOOUTPUT`

Check only, no generation of binder output file.

`/BIND_FILE=C`

Generate binder program in C

`/ELABORATION_DEPENDENCIES`

Output complete list of elaboration-order dependencies.

`-E`

Store tracebacks in exception occurrences when the target supports it. This is the default with the zero cost exception mechanism. This option is currently supported on the following targets: all x86 ports, Solaris, Windows, HP-UX, AIX, PowerPC VxWorks and Alpha VxWorks. See also the packages GNAT.Traceback and GNAT.Traceback.Symbolic for more information. Note that on x86 ports, you must not use `-fomit-frame-pointer` GNAT COMPILE option.

`-h`

Output usage (help) information

`/SEARCH`

Specify directory to be searched for source and ALI files.

`/NOCURRENT_DIRECTORY`

Do not look for sources in the current directory where GNAT BIND was invoked, and do not look for ALI files in the directory containing the ALI file named in the GNAT BIND command line.

`/ORDER_OF_ELABORATION`

Output chosen elaboration order.

- Lxxx Binds the units for library building. In this case the `adainit` and `adafinal` procedures (See see Section 4.7 [Binding with Non-Ada Main Programs], page 82) are renamed to `xxxinit` and `xxxfinal`. Implies `-n`.
- Mxyz Rename generated main program from `main` to `xyz`
- /ERROR_LIMIT=*n* Limit number of detected errors to *n* (1-999).
- /NOMAIN No main program.
- /NOSTD_INCLUDES Do not look for sources in the system default directory.
- /NOSTD_LIBRARIES Do not look for library files in the system default directory.
- /RUNTIME_SYSTEM=*rts-path* Specifies the default location of the runtime library. Same meaning as the equivalent GNAT MAKE flag (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).
- /OUTPUT=*file* Name the output file *file* (default is 'B\$xxx.ADB'). Note that if this option is used, then linking must be done manually, GNAT LINK cannot be used.
- /OBJECT_LIST Output object list.
- p Pessimistic (worst-case) elaboration order
- /READ_SOURCES=ALL Require all source files to be present.
- /NOTIME_STAMP_CHECK Tolerate time stamp and other consistency errors
- T*n* Set the time slice value to *n* microseconds. A value of zero means no time slicing and also indicates to the tasking run time to match as close as possible to the annex D requirements of the RM.
- /REPORT_ERRORS=VERBOSE Verbose mode. Write error messages, header, summary output to 'SYS\$OUTPUT'.
- /WARNINGS=NORMAL Normal warnings mode. Warnings are issued but ignored
- /WARNINGS=SUPPRESS All warning messages are suppressed
- /WARNINGS=ERROR Warning messages are treated as fatal errors
- /READ_SOURCES=NONE Exclude source files (check object consistency only).
- /READ_SOURCES=AVAILABLE Default mode, in which sources are checked for consistency only if they are available.

`/ZERO_MAIN`

No main subprogram.

4.10 Command-Line Access

The package `Ada.Command_Line` provides access to the command-line arguments and program name. In order for this interface to operate correctly, the two variables

```
int gnat_argc;
char **gnat_argv;
```

are declared in one of the GNAT library routines. These variables must be set from the actual `argc` and `argv` values passed to the main program. With no `/NOMAIN` present, GNAT BIND generates the C main program to automatically set these variables. If the `/NOMAIN` qualifier is used, there is no automatic way to set these variables. If they are not set, the procedures in `Ada.Command_Line` will not be available, and any attempt to use them will raise `Constraint_Error`. If command line access is required, your main program must set `gnat_argc` and `gnat_argv` from the `argc` and `argv` values passed to it.

4.11 Search Paths for GNAT BIND

The binder takes the name of an ALI file as its argument and needs to locate source files as well as other ALI files to verify object consistency.

For source files, it follows exactly the same search rules as GNAT COMPILE (see Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56). For ALI files the directories searched are:

1. The directory containing the ALI file named in the command line, unless the qualifier `/NOCURRENT_DIRECTORY` is specified.
2. All directories specified by `/SEARCH` qualifiers on the GNAT BIND command line, in the order given.
3. Each of the directories listed in the value of the `ADA_OBJECTS_PATH` logical name. Normally, define this value as a logical name containing a comma separated list of directory names.

This variable can also be defined by means of an environment string (an argument to the DEC C `exec*` set of functions).

Logical Name:

```
DEFINE ANOTHER_PATH FOO: [BAG]
DEFINE ADA_OBJECTS_PATH ANOTHER_PATH, FOO: [BAM], FOO: [BAR]
```

By default, the path includes GNU:[LIB.OPENVMS7_x.2.8_x.DECLIB] first, followed by the standard Ada 95 libraries in GNU:[LIB.OPENVMS7_x.2.8_x.ADALIB]. If this is not redefined, the user will obtain the DEC Ada83 IO packages (Text_IO, Sequential_IO, etc) instead of the Ada95 packages. Thus, in order to get the Ada 95 packages by default, `ADA_OBJECTS_PATH` must be redefined.

4. The content of the "ada.object.path" file which is part of the GNAT installation tree and is used to store standard libraries such as the GNAT Run Time Library (RTL) unless the qualifier `/NOSTD_LIBRARIES` is specified.

In the binder the qualifier `/SEARCH` is used to specify both source and library file paths. Use `/SOURCE_SEARCH` instead if you want to specify source paths only, and `/LIBRARY_SEARCH` if you want to specify library paths only. This means that for the binder `/SEARCH=dir` is equivalent to `/SOURCE_SEARCH=dir /OBJECT_SEARCH=dir`. The binder generates the bind file (a C language source file) in the current working directory.

The packages `Ada`, `System`, and `Interfaces` and their children make up the GNAT Run-Time Library, together with the package `GNAT` and its children, which contain a set of useful additional library functions provided by GNAT. The sources for these units are needed by the compiler and are kept together in one directory. The ALI files and object files generated by compiling the RTL are needed by the binder and the linker and are kept together in one directory, typically different from the directory containing the sources. In a normal installation, you need not specify these directory names when compiling or binding. Either the environment variables or the built-in defaults cause these files to be found.

Besides simplifying access to the RTL, a major use of search paths is in compiling sources from multiple directories. This can make development environments much more flexible.

4.12 Examples of GNAT BIND Usage

This section contains a number of examples of using the GNAT binding utility `GNAT BIND`.

GNAT BIND hello

The main program `Hello` (source program in `'HELLO.ADB'`) is bound using the standard qualifier settings. The generated main program is `'B~HELLO.ADB'`. This is the normal, default use of the binder.

GNAT BIND HELLO.ALI /OUTPUT=Mainprog.ADB

The main program `Hello` (source program in `'HELLO.ADB'`) is bound using the standard qualifier settings. The generated main program is `'MAINPROG.ADB'` with the associated spec in `'MAINPROG.ADS'`. Note that you must specify the body here not the spec, in the case where the output is in Ada. Note that if this option is used, then linking must be done manually, since `GNAT LINK` will not be able to find the generated file.

GNAT BIND MAIN.ALI /BIND_FILE=C /OUTPUT=Mainprog.C /READ_SOURCES=NONE

The main program `Main` (source program in `'MAIN.ADB'`) is bound, excluding source files from the consistency checking, generating the file `'MAINPROG.C'`.

GNAT BIND /NOMAIN math dbase /BIND_FILE=C /OUTPUT=ADA-CONTROL.C

The main program is in a language other than Ada, but calls to subprograms in packages `Math` and `Dbase` appear. This call to `GNAT BIND` generates the file `'ADA-CONTROL.C'` containing the `adainit` and `adafinal` routines to be called before and after accessing the Ada units.

5 Linking Using GNAT LINK

This chapter discusses **GNAT LINK**, a utility program used to link Ada programs and build an executable file. This is a simple program that invokes the Unix linker (via the **GNAT COMPILE** command) with a correct list of object files and library references. **GNAT LINK** automatically determines the list of files and references for the Ada part of a program. It uses the binder file generated by the binder to determine this list.

5.1 Running GNAT LINK

The form of the **GNAT LINK** command is

```
$ GNAT LINK [qualifiers] mainprog[.ALI] [non-Ada objects]
           [linker options]
```

‘*mainprog.ALI*’ references the ALI file of the main program. The ‘.ALI’ extension of this file can be omitted. From this reference, **GNAT LINK** locates the corresponding binder file ‘*B\$mainprog.ADB*’ and, using the information in this file along with the list of non-Ada objects and linker options, constructs a Unix linker command file to create the executable.

The arguments following ‘*mainprog.ALI*’ are passed to the linker uninterpreted. They typically include the names of object files for units written in other languages than Ada and any library references required to resolve references in any of these foreign language units, or in **pragma Import** statements in any Ada units.

linker options is an optional list of linker specific qualifiers. The default linker called by **GNAT LINK** is *GNAT COMPILE* which in turn calls the appropriate system linker usually called *ld*. Standard options for the linker such as *-lmy_lib* or *-Ldir* can be added as is. For options that are not recognized by *GNAT COMPILE* as linker options, the *GNAT COMPILE* qualifiers *-Xlinker* or *-Wl*, shall be used. Refer to the GCC documentation for details. Here is an example showing how to generate a linker map assuming that the underlying linker is GNU *ld*:

```
$ GNAT LINK my_prog -Wl,-Map,MAPFILE
```

Using *linker options* it is possible to set the program stack and heap size. See Section 5.3 [Setting Stack Size from **GNAT LINK**], page 88 and see Section 5.4 [Setting Heap Size from **GNAT LINK**], page 89.

GNAT LINK determines the list of objects required by the Ada program and prepends them to the list of objects passed to the linker. **GNAT LINK** also gathers any arguments set by the use of **pragma Linker_Options** and adds them to the list of arguments presented to the linker.

GNAT LINK accepts the following types of extra files on the command line: objects (.OBJ), libraries (.OLB), shareable images (.EXE), and options files (.OPT). These are recognized and handled according to their extension.

5.2 Qualifiers for GNAT LINK

The following qualifiers are available with the **GNAT LINK** utility:

```
/BIND_FILE=ADA
```

The binder has generated code in Ada. This is the default.

/BIND_FILE=C

If instead of generating a file in Ada, the binder has generated one in C, then the linker needs to know about it. Use this qualifier to signal to GNAT LINK that the binder has generated C code rather than Ada code.

-f

On some targets, the command line length is limited, and GNAT LINK will generate a separate file for the linker if the list of object files is too long. The **-f** flag forces this file to be generated even if the limit is not exceeded. This is useful in some cases to deal with special situations where the command line length is exceeded.

/DEBUG

The option to include debugging information causes the Ada bind file (in other words, 'B\$mainprog.ADB') to be compiled with /DEBUG. In addition, the binder does not delete the 'B\$mainprog.ADB', 'B\$mainprog.OBJ' and 'B\$mainprog.ALI' files. Without /DEBUG, the binder removes these files by default. The same procedure apply if a C bind file was generated using /BIND_FILE=C GNAT BIND option, in this case the filenames are 'B_mainprog.C' and 'B_mainprog.OBJ'.

/VERBOSE

Causes additional information to be output, including a full list of the included object files. This qualifier option is most useful when you want to see what set of object files are being used in the link step.

/EXECUTABLE=exec-name

exec-name specifies an alternate name for the generated executable program. If this qualifier is omitted, the executable has the same name as the main unit. For example, GNAT LINK TRY.ALI creates an executable called 'TRY.EXE'.

/DEBUG=TRACEBACK

This qualifier causes sufficient information to be included in the executable file to allow a traceback, but does not include the full symbol information needed by the debugger.

/IDENTIFICATION="<string>"

"<string>" specifies the string to be stored in the image file identification field in the image header. It overrides any pragma Ident specified string.

/NOINHIBIT-EXEC

Generate the executable file even if there are linker warnings.

/NOSTART_FILES

Don't link in the object file containing the "main" transfer address. Used when linking with a foreign language main program compiled with a Digital compiler.

/STATIC

Prefer linking with object libraries over shareable images, even without /DEBUG.

5.3 Setting Stack Size from GNAT LINK

It is possible to specify the program stack size from GNAT LINK. Assuming that the underlying linker is GNU ld there is two ways to do so:

- using `-Xlinker` linker option

```
$ GNAT LINK hello -Xlinker --stack=0x10000,0x1000
```

This set the stack reserve size to 0x10000 bytes and the stack commit size to 0x1000 bytes.

- using `-Wl` linker option

```
$ GNAT LINK hello -Wl,--stack=0x1000000
```

This set the stack reserve size to 0x1000000 bytes. Note that with `-Wl` option it is not possible to set the stack commit size because the coma is a separator for this option.

5.4 Setting Heap Size from GNAT LINK

It is possible to specify the program heap size from GNAT LINK. Assuming that the underlying linker is GNU ld there is two ways to do so:

- using `-Xlinker` linker option

```
$ GNAT LINK hello -Xlinker --heap=0x10000,0x1000
```

This set the heap reserve size to 0x10000 bytes and the heap commit size to 0x1000 bytes.

- using `-Wl` linker option

```
$ GNAT LINK hello -Wl,--heap=0x1000000
```

This set the heap reserve size to 0x1000000 bytes. Note that with `-Wl` option it is not possible to set the heap commit size because the coma is a separator for this option.

6 The GNAT Make Program GNAT MAKE

A typical development cycle when working on an Ada program consists of the following steps:

1. Edit some sources to fix bugs.
2. Add enhancements.
3. Compile all sources affected.
4. Rebind and relink.
5. Test.

The third step can be tricky, because not only do the modified files have to be compiled, but any files depending on these files must also be recompiled. The dependency rules in Ada can be quite complex, especially in the presence of overloading, `use` clauses, generics and inlined subprograms.

GNAT MAKE automatically takes care of the third and fourth steps of this process. It determines which sources need to be compiled, compiles them, and binds and links the resulting object files.

Unlike some other Ada make programs, the dependencies are always accurately recomputed from the new sources. The source based approach of the GNAT compilation model makes this possible. This means that if changes to the source program cause corresponding changes in dependencies, they will always be tracked exactly correctly by GNAT MAKE.

6.1 Running GNAT MAKE

The usual form of the GNAT MAKE command is

```
$ GNAT MAKE [qualifiers] file_name [file_names] [mode_qualifiers]
```

The only required argument is one *file_name*, which specifies a compilation unit that is a main program. Several *file_names* can be specified: this will result in several executables being built. If *qualifiers* are present, they can be placed before the first *file_name*, between *file_names* or after the last *file_name*. If *mode_qualifiers* are present, they must always be placed after the last *file_name* and all *qualifiers*.

If you are using standard file extensions (`.ADB` and `.ADS`), then the extension may be omitted from the *file_name* arguments. However, if you are using non-standard extensions, then it is required that the extension be given. A relative or absolute directory path can be specified in a *file_name*, in which case, the input source file will be searched for in the specified directory only. Otherwise, the input source file will first be searched in the directory where GNAT MAKE was invoked and if it is not found, it will be search on the source path of the compiler as described in Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56.

When several *file_names* are specified, if an executable needs to be rebuilt and relinked, all subsequent executables will be rebuilt and relinked, even if this would not be absolutely necessary.

All GNAT MAKE output (except when you specify `/DEPENDENCIES_LIST`) is to `'SYS$ERROR'`. The output produced by the `/DEPENDENCIES_LIST` qualifier is send to `'SYS$OUTPUT'`.

6.2 Qualifiers for GNAT MAKE

You may specify any of the following qualifiers to GNAT MAKE:

/ALL_FILES

Consider all files in the make process, even the GNAT internal system files (for example, the predefined Ada library files), as well as any locked files. Locked files are files whose ALI file is write-protected. By default, GNAT MAKE does not check these files, because the assumption is that the GNAT internal files are properly up to date, and also that any write protected ALI files have been properly installed. Note that if there is an installation problem, such that one of these files is not up to date, it will be properly caught by the binder. You may have to specify this qualifier if you are working on GNAT itself. **/ALL_FILES** is also useful in conjunction with **/FORCE_COMPILE** if you need to re-compile an entire application, including run-time files, using special configuration pragma settings, such as a non-standard **Float_Representation** pragma. By default GNAT MAKE **/ALL_FILES** compiles all GNAT internal files with the **/CHECKS=SUPPRESS_ALL /STYLE_CHECKS=GNAT** qualifier.

/ACTIONS=BIND

Bind only. Can be combined with **/ACTIONS=COMPILE** to do compilation and binding, but no link. Can be combined with **/ACTIONS=LINK** to do binding and linking. When not combined with **/ACTIONS=COMPILE** all the units in the closure of the main program must have been previously compiled and must be up to date. The root unit specified by *file_name* may be given without extension, with the source extension or, if no GNAT Project File is specified, with the ALI file extension.

/ACTIONS=COMPILE

Compile only. Do not perform binding, except when **/ACTIONS=BIND** is also specified. Do not perform linking, except if both **/ACTIONS=BIND** and **/ACTIONS=LINK** are also specified. If the root unit specified by *file_name* is not a main unit, this is the default. Otherwise GNAT MAKE will attempt binding and linking unless all objects are up to date and the executable is more recent than the objects.

/MAPPING

Use a mapping file. A mapping file is a way to communicate to the compiler two mappings: from unit names to file names (without any directory information) and from file names to path names (with full directory information). These mappings are used by the compiler to short-circuit the path search. When GNAT MAKE is invoked with this qualifier, it will create a mapping file, initially populated by the project manager, if **-P** is used, otherwise initially empty. Each invocation of the compiler will add the newly accessed sources to the mapping file. This will improve the source search during the next invocation of the compiler.

/FORCE_COMPILE

Force recompilations. Recompile all sources, even though some object files may be up to date, but don't recompile predefined or GNAT internal files or locked

files (files with a write-protected ALI file), unless the `/ALL_FILES` qualifier is also specified.

`/IN_PLACE`

In normal mode, GNAT MAKE compiles all object files and ALI files into the current directory. If the `/IN_PLACE` qualifier is used, then instead object files and ALI files that already exist are overwritten in place. This means that once a large project is organized into separate directories in the desired manner, then GNAT MAKE will automatically maintain and update this organization. If no ALI files are found on the Ada object path (Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56), the new object and ALI files are created in the directory containing the source being compiled. If another organization is desired, where objects and sources are kept in different directories, a useful technique is to create dummy ALI files in the desired directories. When detecting such a dummy file, GNAT MAKE will be forced to recompile the corresponding source file, and it will be put the resulting object and ALI files in the directory where it found the dummy file.

`/PROCESSES=n`

Use *n* processes to carry out the (re)compilations. On a multiprocessor machine compilations will occur in parallel. In the event of compilation errors, messages from various compilations might get interspersed (but GNAT MAKE will give you the full ordered list of failing compiles at the end). If this is problematic, rerun the make process with *n* set to 1 to get a clean list of messages.

`/CONTINUE_ON_ERROR`

Keep going. Continue as much as possible after a compilation error. To ease the programmer's task in case of compilation errors, the list of sources for which the compile fails is given when GNAT MAKE terminates.

If GNAT MAKE is invoked with several 'file_names' and with this qualifier, if there are compilation errors when building an executable, GNAT MAKE will not attempt to build the following executables.

`/ACTIONS=LINK`

Link only. Can be combined with `/ACTIONS=BIND` to binding and linking. Linking will not be performed if combined with `/ACTIONS=COMPILE` but not with `/ACTIONS=BIND`. When not combined with `/ACTIONS=BIND` all the units in the closure of the main program must have been previously compiled and must be up to date, and the main program need to have been bound. The root unit specified by *file_name* may be given without extension, with the source extension or, if no GNAT Project File is specified, with the ALI file extension.

`/MINIMAL_RECOMPILE`

Specifies that the minimum necessary amount of recompilations be performed. In this mode GNAT MAKE ignores time stamp differences when the only modifications to a source file consist in adding/removing comments, empty lines, spaces or tabs. This means that if you have changed the comments in a source file or have simply reformatted it, using this qualifier will tell GNAT MAKE not to

recompile files that depend on it (provided other sources on which these files depend have undergone no semantic modifications). Note that the debugging information may be out of date with respect to the sources if the `-m` qualifier causes a compilation to be switched, so the use of this qualifier represents a trade-off between compilation time and accurate debugging information.

`/DEPENDENCIES_LIST`

Check if all objects are up to date. If they are, output the object dependences to `'SYS$OUTPUT'` in a form that can be directly exploited in a `'Makefile'`. By default, each source file is prefixed with its (relative or absolute) directory name. This name is whatever you specified in the various `/SOURCE_SEARCH` and `/SEARCH` qualifiers. If you use `GNAT MAKE /DEPENDENCIES_LIST /QUIET` (see below), only the source file names, without relative paths, are output. If you just specify the `/DEPENDENCIES_LIST` qualifier, dependencies of the GNAT internal system files are omitted. This is typically what you want. If you also specify the `/ALL_FILES` qualifier, dependencies of the GNAT internal files are also listed. Note that dependencies of the objects in external Ada libraries (see qualifier `/SKIP_MISSING=dir` in the following list) are never reported.

`/DO_OBJECT_CHECK`

Don't compile, bind, or link. Checks if all objects are up to date. If they are not, the full name of the first file that needs to be recompiled is printed. Repeated use of this option, followed by compiling the indicated source file, will eventually result in recompiling all required units.

`/EXECUTABLE=exec_name`

Output executable name. The name of the final executable program will be `exec_name`. If the `/EXECUTABLE` qualifier is omitted the default name for the executable will be the name of the input file in appropriate form for an executable file on the host system.

This qualifier cannot be used when invoking `GNAT MAKE` with several `'file_names'`.

`/QUIET` Quiet. When this flag is not set, the commands carried out by `GNAT MAKE` are displayed.

`/SWITCH_CHECK/`

Recompile if compiler qualifiers have changed since last compilation. All compiler qualifiers but `-I` and `-o` are taken into account in the following way: orders between different "first letter" qualifiers are ignored, but orders between same qualifiers are taken into account. For example, `-O /OPTIMIZE=ALL` is different than `/OPTIMIZE=ALL -O`, but `-g -O` is equivalent to `-O -g`.

`/UNIQUE` Unique. Recompile at most the main file. It implies `-c`. Combined with `-f`, it is equivalent to calling the compiler directly.

`/REASONS` Verbose. Displays the reason for all recompilations `GNAT MAKE` decides are necessary.

`/NOMAIN` No main subprogram. Bind and link the program even if the unit name given on the command line is a package name. The resulting executable will execute

the elaboration routines of the package and its closure, then the finalization routines.

GNAT COMPILE qualifiers

Any qualifier that cannot be recognized as a qualifier for GNAT MAKE but is recognizable as a valid qualifier for GNAT COMPILE is automatically treated as a compiler qualifier, and passed on to all compilations that are carried out.

Source and library search path qualifiers:

/SOURCE_SEARCH=*dir*

When looking for source files also look in directory *dir*. The order in which source files search is undertaken is described in Section 3.3 [Search Paths and the Run-Time Library (RTL)], page 56.

/SKIP_MISSING=*dir*

Consider *dir* as being an externally provided Ada library. Instructs GNAT MAKE to skip compilation units whose '.ALI' files have been located in directory *dir*. This allows you to have missing bodies for the units in *dir* and to ignore out of date bodies for the same units. You still need to specify the location of the specs for these units by using the qualifiers /SOURCE_SEARCH=*dir* or /SEARCH=*dir*. Note: this qualifier is provided for compatibility with previous versions of GNAT MAKE. The easier method of causing standard libraries to be excluded from consideration is to write-protect the corresponding ALI files.

/OBJECT_SEARCH=*dir*

When searching for library and object files, look in directory *dir*. The order in which library files are searched is described in Section 4.11 [Search Paths for GNAT BIND], page 85.

/CONDITIONAL_SOURCE_SEARCH=*dir*

Equivalent to /SKIP_MISSING=*dir* /SOURCE_SEARCH=*dir*.

/SEARCH=*dir*

Equivalent to /OBJECT_SEARCH=*dir* /SOURCE_SEARCH=*dir*.

/NOCURRENT_DIRECTORY

Do not look for source files in the directory containing the source file named in the command line. Do not look for ALI or object files in the directory where GNAT MAKE was invoked.

/LIBRARY_SEARCH=*dir*

Add directory *dir* to the list of directories in which the linker will search for libraries. This is equivalent to /LINKER_QUALIFIERS /LIBRARY_SEARCH=*dir*.

/NOSTD_INCLUDES

Do not look for source files in the system default directory.

/NOSTD_LIBRARIES

Do not look for library files in the system default directory.

/RUNTIME_SYSTEM=*rts-path*

Specifies the default location of the runtime library. We look for the runtime in the following directories, and stop as soon as a valid runtime is found ("adainclude" or "ada_source_path", and "adalib" or "ada_object_path" present):

- <current directory>/\$rts_path
- <default-search-dir>/\$rts_path
- <default-search-dir>/rts-\$rts_path

The selected path is handled like a normal RTS path.

6.3 Mode Qualifiers for GNAT MAKE

The mode qualifiers (referred to as `mode_qualifiers`) allow the inclusion of qualifiers that are to be passed to the compiler itself, the binder or the linker. The effect of a mode qualifier is to cause all subsequent qualifiers up to the end of the qualifier list, or up to the next mode qualifier, to be interpreted as qualifiers to be passed on to the designated component of GNAT.

`/COMPILER_QUALIFIERS qualifiers`

Compiler qualifiers. Here *qualifiers* is a list of qualifiers that are valid qualifiers for GNAT COMPILER. They will be passed on to all compile steps performed by GNAT MAKE.

`/BINDER_QUALIFIERS qualifiers`

Binder qualifiers. Here *qualifiers* is a list of qualifiers that are valid qualifiers for GNAT COMPILER. They will be passed on to all bind steps performed by GNAT MAKE.

`/LINKER_QUALIFIERS qualifiers`

Linker qualifiers. Here *qualifiers* is a list of qualifiers that are valid qualifiers for GNAT COMPILER. They will be passed on to all link steps performed by GNAT MAKE.

6.4 Notes on the Command Line

This section contains some additional useful notes on the operation of the GNAT MAKE command.

- If GNAT MAKE finds no ALI files, it recompiles the main program and all other units required by the main program. This means that GNAT MAKE can be used for the initial compile, as well as during subsequent steps of the development cycle.
- If you enter GNAT MAKE *file.ADB*, where '*file.ADB*' is a subunit or body of a generic unit, GNAT MAKE recompiles '*file.ADB*' (because it finds no ALI) and stops, issuing a warning.
- In GNAT MAKE the qualifier /SEARCH is used to specify both source and library file paths. Use /SOURCE_SEARCH instead if you just want to specify source paths only and /OBJECT_SEARCH if you want to specify library paths only.
- GNAT MAKE examines both an ALI file and its corresponding object file for consistency. If an ALI is more recent than its corresponding object, or if the object file is missing, the corresponding source will be recompiled. Note that GNAT MAKE expects an ALI and the corresponding object file to be in the same directory.

- GNAT MAKE will ignore any files whose ALI file is write-protected. This may conveniently be used to exclude standard libraries from consideration and in particular it means that the use of the `/FORCE_COMPILE` qualifier will not recompile these files unless `/ALL_FILES` is also specified.
- GNAT MAKE has been designed to make the use of Ada libraries particularly convenient. Assume you have an Ada library organized as follows: `[OBJ_DIR]` contains the objects and ALI files for of your Ada compilation units, whereas `[INCLUDE_DIR]` contains the specs of these units, but no bodies. Then to compile a unit stored in `MAIN.ADB`, which uses this Ada library you would just type


```
$ GNAT MAKE /SOURCE_SEARCH=[INCLUDE_DIR]
               /SKIP_MISSING=[OBJ_DIR] main
```
- Using GNAT MAKE along with the `/MINIMAL_RECOMPILATION` qualifier provides a mechanism for avoiding unnecessary recompilations. Using this qualifier, you can update the comments/format of your source files without having to recompile everything. Note, however, that adding or deleting lines in a source files may render its debugging info obsolete. If the file in question is a spec, the impact is rather limited, as that debugging info will only be useful during the elaboration phase of your program. For bodies the impact can be more significant. In all events, your debugger will warn you if a source file is more recent than the corresponding object, and alert you to the fact that the debugging information may be out of date.

6.5 How GNAT MAKE Works

Generally GNAT MAKE automatically performs all necessary recompilations and you don't need to worry about how it works. However, it may be useful to have some basic understanding of the GNAT MAKE approach and in particular to understand how it uses the results of previous compilations without incorrectly depending on them.

First a definition: an object file is considered *up to date* if the corresponding ALI file exists and its time stamp predates that of the object file and if all the source files listed in the dependency section of this ALI file have time stamps matching those in the ALI file. This means that neither the source file itself nor any files that it depends on have been modified, and hence there is no need to recompile this file.

GNAT MAKE works by first checking if the specified main unit is up to date. If so, no compilations are required for the main unit. If not, GNAT MAKE compiles the main program to build a new ALI file that reflects the latest sources. Then the ALI file of the main unit is examined to find all the source files on which the main program depends, and GNAT MAKE recursively applies the above procedure on all these files.

This process ensures that GNAT MAKE only trusts the dependencies in an existing ALI file if they are known to be correct. Otherwise it always recompiles to determine a new, guaranteed accurate set of dependencies. As a result the program is compiled "upside down" from what may be more familiar as the required order of compilation in some other Ada systems. In particular, clients are compiled before the units on which they depend. The ability of GNAT to compile in any order is critical in allowing an order of compilation to be chosen that guarantees that GNAT MAKE will recompute a correct set of new dependencies if necessary.

When invoking **GNAT MAKE** with several *file_names*, if a unit is imported by several of the executables, it will be recompiled at most once.

6.6 Examples of GNAT MAKE Usage

GNAT MAKE HELLO.ADB

Compile all files necessary to bind and link the main program 'HELLO.ADB' (containing unit `Hello`) and bind and link the resulting object files to generate an executable file 'HELLO.EXE'.

GNAT MAKE main1 main2 main3

Compile all files necessary to bind and link the main programs 'MAIN1.ADB' (containing unit `Main1`), 'MAIN2.ADB' (containing unit `Main2`) and 'MAIN3.ADB' (containing unit `Main3`) and bind and link the resulting object files to generate three executable files 'MAIN1.EXE', 'MAIN2.EXE' and 'MAIN3.EXE'.

**GNAT MAKE Main_Unit /QUIET /COMPILER_QUALIFIERS /OPTIMIZE=ALL
/BINDER_QUALIFIERS /ORDER_OF_ELABORATION**

Compile all files necessary to bind and link the main program unit `Main_Unit` (from file 'MAIN_UNIT.ADB'). All compilations will be done with optimization level 2 and the order of elaboration will be listed by the binder. **GNAT MAKE** will operate in quiet mode, not displaying commands it is executing.

7 Renaming Files Using GNAT CHOP

This chapter discusses how to handle files with multiple units by using the **GNAT CHOP** utility. This utility is also useful in renaming files to meet the standard GNAT default file naming conventions.

7.1 Handling Files with Multiple Units

The basic compilation model of GNAT requires that a file submitted to the compiler have only one unit and there be a strict correspondence between the file name and the unit name.

The **GNAT CHOP** utility allows both of these rules to be relaxed, allowing GNAT to process files which contain multiple compilation units and files with arbitrary file names. **GNAT CHOP** reads the specified file and generates one or more output files, containing one unit per file. The unit and the file name correspond, as required by GNAT.

If you want to permanently restructure a set of "foreign" files so that they match the GNAT rules, and do the remaining development using the GNAT structure, you can simply use **GNAT CHOP** once, generate the new set of files and work with them from that point on.

Alternatively, if you want to keep your files in the "foreign" format, perhaps to maintain compatibility with some other Ada compilation system, you can set up a procedure where you use **GNAT CHOP** each time you compile, regarding the source files that it writes as temporary files that you throw away.

7.2 Operating GNAT CHOP in Compilation Mode

The basic function of **GNAT CHOP** is to take a file with multiple units and split it into separate files. The boundary between files is reasonably clear, except for the issue of comments and pragmas. In default mode, the rule is that any pragmas between units belong to the previous unit, except that configuration pragmas always belong to the following unit. Any comments belong to the following unit. These rules almost always result in the right choice of the split point without needing to mark it explicitly and most users will find this default to be what they want. In this default mode it is incorrect to submit a file containing only configuration pragmas, or one that ends in configuration pragmas, to **GNAT CHOP**.

However, using a special option to activate "compilation mode", **GNAT CHOP** can perform another function, which is to provide exactly the semantics required by the RM for handling of configuration pragmas in a compilation. In the absence of configuration pragmas (at the main file level), this option has no effect, but it causes such configuration pragmas to be handled in a quite different manner.

First, in compilation mode, if **GNAT CHOP** is given a file that consists of only configuration pragmas, then this file is appended to the `'GNAT.ADC'` file in the current directory. This behavior provides the required behavior described in the RM for the actions to be taken on submitting such a file to the compiler, namely that these pragmas should apply to all subsequent compilations in the same compilation environment. Using GNAT, the current directory, possibly containing a `'GNAT.ADC'` file is the representation of a compilation environment. For more information on the `'GNAT.ADC'` file, see the section on handling of configuration pragmas see Section 8.1 [Handling of Configuration Pragmas], page 103.

Second, in compilation mode, if **GNAT CHOP** is given a file that starts with configuration pragmas, and contains one or more units, then these configuration pragmas are prepended to each of the chopped files. This behavior provides the required behavior described in the RM for the actions to be taken on compiling such a file, namely that the pragmas apply to all units in the compilation, but not to subsequently compiled units.

Finally, if configuration pragmas appear between units, they are appended to the previous unit. This results in the previous unit being illegal, since the compiler does not accept configuration pragmas that follow a unit. This provides the required RM behavior that forbids configuration pragmas other than those preceding the first compilation unit of a compilation.

For most purposes, **GNAT CHOP** will be used in default mode. The compilation mode described above is used only if you need exactly accurate behavior with respect to compilations, and you have files that contain multiple units and configuration pragmas. In this circumstance the use of **GNAT CHOP** with the compilation mode qualifier provides the required behavior, and is for example the mode in which GNAT processes the ACVC tests.

7.3 Command Line for GNAT CHOP

The **GNAT CHOP** command has the form:

```
$ GNAT CHOP qualifiers file name [file name file name ...]
    [directory]
```

The only required argument is the file name of the file to be chopped. There are no restrictions on the form of this file name. The file itself contains one or more Ada units, in normal GNAT format, concatenated together. As shown, more than one file may be presented to be chopped.

When run in default mode, **GNAT CHOP** generates one output file in the current directory for each unit in each of the files.

directory, if specified, gives the name of the directory to which the output files will be written. If it is not specified, all files are written to the current directory.

For example, given a file called 'hellofiles' containing

```
procedure hello;

with Text_IO; use Text_IO;
procedure hello is
begin
  Put_Line ("Hello");
end hello;
```

the command

```
$ GNAT CHOP HELLOFILES.
```

generates two files in the current directory, one called 'HELLO.ADS' containing the single line that is the procedure spec, and the other called 'HELLO.ADB' containing the remaining text. The original file is not affected. The generated files can be compiled in the normal manner.

7.4 Qualifiers for GNAT CHOP

GNAT CHOP recognizes the following qualifiers:

/COMPILATION

Causes GNAT CHOP to operate in compilation mode, in which configuration pragmas are handled according to strict RM rules. See previous section for a full description of this mode.

/HELP

Causes GNAT CHOP to generate a brief help summary to the standard output file showing usage information.

/FILE_NAME_MAX_LENGTH=*mm*

Limit generated file names to the specified number *mm* of characters. This is useful if the resulting set of files is required to be interoperable with systems which limit the length of file names. If no value is given, or if no **/FILE_NAME_MAX_LENGTH** qualifier is given, a default of 39, suitable for OpenVMS Alpha Systems, is assumed

/PRESERVE

Causes the file creation time stamp of the input file to be preserved and used for the time stamp of the output file(s). This may be useful for preserving coherency of time stamps in an environment where GNAT CHOP is used as part of a standard build process.

/QUIET

Causes output of informational messages indicating the set of generated files to be suppressed. Warnings and error messages are unaffected.

/REFERENCE

Generate **Source_Reference** pragmas. Use this qualifier if the output files are regarded as temporary and development is to be done in terms of the original unchopped file. This qualifier causes **Source_Reference** pragmas to be inserted into each of the generated files to refer back to the original file name and line number. The result is that all error messages refer back to the original unchopped file. In addition, the debugging information placed into the object file (when the **/DEBUG** qualifier of GNAT COMPILE or GNAT MAKE is specified) also refers back to this original file so that tools like profilers and debuggers will give information in terms of the original unchopped file.

If the original file to be chopped itself contains a **Source_Reference** pragma referencing a third file, then GNAT CHOP respects this pragma, and the generated **Source_Reference** pragmas in the chopped file refer to the original file, with appropriate line numbers. This is particularly useful when GNAT CHOP is used in conjunction with GNAT PREPROCESS to compile files that contain pre-processing statements and multiple units.

/VERBOSE

Causes GNAT CHOP to operate in verbose mode. The version number and copyright notice are output, as well as exact copies of the GNAT1 commands spawned to obtain the chop control information.

/OVERWRITE

Overwrite existing file names. Normally GNAT CHOP regards it as a fatal error if there is already a file with the same name as a file it would otherwise output,

in other words if the files to be chopped contain duplicated units. This qualifier bypasses this check, and causes all but the last instance of such duplicated units to be skipped.

7.5 Examples of GNAT CHOP Usage

GNAT CHOP /OVERWRITE HELLO_S.ADA [ICHBIAH.FILES]

Chops the source file 'HELLO_S.ADA'. The output files will be placed in the directory '[ICHBIAH.FILES]', overwriting any files with matching names in that directory (no files in the current directory are modified).

GNAT CHOP ARCHIVE.

Chops the source file 'ARCHIVE.' into the current directory. One useful application of GNAT CHOP is in sending sets of sources around, for example in email messages. The required sources are simply concatenated (for example, using a VMS APPEND/NEW command), and then GNAT CHOP is used at the other end to reconstitute the original file names.

GNAT CHOP file1 file2 file3 direc

Chops all units in files 'file1', 'file2', 'file3', placing the resulting files in the directory 'direc'. Note that if any units occur more than once anywhere within this set of files, an error message is generated, and no files are written. To override this check, use the /OVERWRITE qualifier, in which case the last occurrence in the last file will be the one that is output, and earlier duplicate occurrences for a given unit will be skipped.

8 Configuration Pragmas

In Ada 95, configuration pragmas include those pragmas described as such in the Ada 95 Reference Manual, as well as implementation-dependent pragmas that are configuration pragmas. See the individual descriptions of pragmas in the GNAT Reference Manual for details on these additional GNAT-specific configuration pragmas. Most notably, the pragma `Source_File_Name`, which allows specifying non-default names for source files, is a configuration pragma. The following is a complete list of configuration pragmas recognized by GNAT:

```
Ada_83
Ada_95
C_Pass_By_Copy
Component_Alignment
Discard_Names
Elaboration_Checks
Eliminate
Extend_System
Extensions_Allowed
External_Name_Casing
Float_Representation
InitializeScalars
License
Locking_Policy
Long_Float
No_Run_Time
NormalizeScalars
Polling
Propagate_Exceptions
Queuing_Policy
Ravenscar
Restricted_Run_Time
Restrictions
Reviewable
Source_File_Name
Style_Checks
Suppress
Task_Dispatching_Policy
Unsuppress
Use_VADS_Size
Warnings
Validity_Checks
```

8.1 Handling of Configuration Pragmas

Configuration pragmas may either appear at the start of a compilation unit, in which case they apply only to that unit, or they may apply to all compilations performed in a given compilation environment.

GNAT also provides the GNAT CHOP utility to provide an automatic way to handle configuration pragmas following the semantics for compilations (that is, files with multiple units), described in the RM. See section see Section 7.2 [Operating GNAT CHOP in Compilation Mode], page 99 for details. However, for most purposes, it will be more convenient to edit the ‘GNAT.ADC’ file that contains configuration pragmas directly, as described in the following section.

8.2 The Configuration Pragmas Files

In GNAT a compilation environment is defined by the current directory at the time that a compile command is given. This current directory is searched for a file whose name is 'GNAT.ADC'. If this file is present, it is expected to contain one or more configuration pragmas that will be applied to the current compilation. However, if the qualifier '-gnatA' is used, 'GNAT.ADC' is not considered.

Configuration pragmas may be entered into the 'GNAT.ADC' file either by running GNAT CHOP on a source file that consists only of configuration pragmas, or more conveniently by direct editing of the 'GNAT.ADC' file, which is a standard format source file.

In addition to 'GNAT.ADC', one additional file containing configuration pragmas may be applied to the current compilation using the qualifier '-gnatA' *path*. *path* must designate an existing file that contains only configuration pragmas. These configuration pragmas are in addition to those found in 'GNAT.ADC' (provided 'GNAT.ADC' is present and qualifier '-gnatA' is not used).

It is allowed to specify several qualifiers '-gnatA', however only the last one on the command line will be taken into account.

Of special interest to GNAT OpenVMS Alpha is the following configuration pragma:

```
pragma Extend_System (Aux_DEC);
```

In the presence of this pragma, GNAT adds to the definition of the predefined package SYSTEM all the additional types and subprograms that are defined in DEC Ada. See see Chapter 22 [Compatibility with DEC Ada], page 221 for details.

9 Handling Arbitrary File Naming Conventions Using `gnatname`

9.1 Arbitrary File Naming Conventions

The GNAT compiler must be able to know the source file name of a compilation unit. When using the standard GNAT default file naming conventions (`.ADS` for specs, `.ADB` for bodies), the GNAT compiler does not need additional information.

When the source file names do not follow the standard GNAT default file naming conventions, the GNAT compiler must be given additional information through a configuration pragmas file (see Chapter 8 [Configuration Pragmas], page 103) or a project file. When the non standard file naming conventions are well-defined, a small number of pragmas `Source_File_Name` specifying a naming pattern (see Section 2.5 [Alternative File Naming Schemes], page 16) may be sufficient. However, if the file naming conventions are irregular or arbitrary, a number of pragma `Source_File_Name` for individual compilation units must be defined. To help maintain the correspondence between compilation unit names and source file names within the compiler, GNAT provides a tool `gnatname` to generate the required pragmas for a set of files.

9.2 Running `gnatname`

The usual form of the `gnatname` command is

```
$ gnatname [qualifiers] naming_pattern [naming_patterns]
```

All of the arguments are optional. If invoked without any argument, `gnatname` will display its usage.

When used with at least one naming pattern, `gnatname` will attempt to find all the compilation units in files that follow at least one of the naming patterns. To find these compilation units, `gnatname` will use the GNAT compiler in syntax-check-only mode on all regular files.

One or several Naming Patterns may be given as arguments to `gnatname`. Each Naming Pattern is enclosed between double quotes. A Naming Pattern is a regular expression similar to the wildcard patterns used in file names by the Unix shells or the DOS prompt.

Examples of Naming Patterns are

```
"*. [12] .ADA"
"* .ad[sb] *"
"body_*"      "spec_*
```

For a more complete description of the syntax of Naming Patterns, see the second kind of regular expressions described in ‘`G-REGEXP.ADS`’ (the “Glob” regular expressions).

When invoked with no qualifiers, `gnatname` will create a configuration pragmas file ‘`GNAT.ADC`’ in the current working directory, with pragmas `Source_File_Name` for each file that contains a valid Ada unit.

9.3 Qualifiers for gnatname

Qualifiers for **gnatname** must precede any specified Naming Pattern.

You may specify any of the following qualifiers to **gnatname**:

- c'file' Create a configuration pragmas file 'file' (instead of the default 'GNAT.ADC'). There may be zero, one or more space between -c and 'file'. 'file' may include directory information. 'file' must be writeable. There may be only one qualifier -c. When a qualifier -c is specified, no qualifier -P may be specified (see below).
- d'dir' Look for source files in directory 'dir'. There may be zero, one or more spaces between -d and 'dir'. When a qualifier -d is specified, the current working directory will not be searched for source files, unless it is explicitly specified with a -d or -D qualifier. Several qualifiers -d may be specified. If 'dir' is a relative path, it is relative to the directory of the configuration pragmas file specified with qualifier -c, or to the directory of the project file specified with qualifier -P or, if neither qualifier -c nor qualifier -P are specified, it is relative to the current working directory. The directory specified with qualifier -c must exist and be readable.
- D'file' Look for source files in all directories listed in text file 'file'. There may be zero, one or more spaces between -d and 'dir'. 'file' must be an existing, readable text file. Each non empty line in 'file' must be a directory. Specifying qualifier -D is equivalent to specifying as many qualifiers -d as there are non empty lines in 'file'.
- h Output usage (help) information. The output is written to 'SYS\$OUTPUT'.
- P'proj' Create or update project file 'proj'. There may be zero, one or more space between -P and 'proj'. 'proj' may include directory information. 'proj' must be writeable. There may be only one qualifier -P. When a qualifier -P is specified, no qualifier -c may be specified.
- v Verbose mode. Output detailed explanation of behavior to 'SYS\$OUTPUT'. This includes name of the file written, the name of the directories to search and, for each file in those directories whose name matches at least one of the Naming Patterns, an indication of whether the file contains a unit, and if so the name of the unit.
- v -v Very Verbose mode. In addition to the output produced in verbose mode, for each file in the searched directories whose name matches none of the Naming Patterns, an indication is given that there is no match.
- x'pattern' Excluded patterns. Using this qualifier, it is possible to exclude some files that would match the name patterns. For example, "**gnatname -x "*_NT.ADA" "*.ADA"** will look for Ada units in all files with the '.ADA' extension, except those whose names end with '_NT.ADA'.

9.4 Examples of gnatname Usage

```
$ gnatname -c /home/me/NAMES.ADC -d sources "[a-z]*.ADA*"
```

In this example, the directory ‘/home/me’ must already exist and be writeable. In addition, the directory ‘/home/me/sources’ (specified by **-d sources**) must exist and be readable. Note the optional spaces after **-c** and **-d**.

```
$ gnatname -P/home/me/proj -x "*_NT_BODY.ADA" -dsources -dsources/plus -Dcommon_dirs.txt "body_*" "spec_*
```

Note that several qualifiers **-d** may be used, even in conjunction with one or several qualifiers **-D**. Several Naming Patterns and one excluded pattern are used in this example.

10 GNAT Project Manager

10.1 Introduction

This chapter describes GNAT's *Project Manager*, a facility that lets you configure various properties for a collection of source files. In particular, you can specify:

- The directory or set of directories containing the source files, and/or the names of the specific source files themselves
- The directory in which the compiler's output ('ALI' files, object files, tree files) will be placed
- The directory in which the executable programs will be placed
- Qualifier settings for any of the project-enabled tools (GNAT MAKE, compiler, binder, linker, GNAT LIST, GNAT XREF, GNAT FIND); you can apply these settings either globally or to individual units
- The source files containing the main subprogram(s) to be built
- The source programming language(s) (currently Ada and/or C)
- Source file naming conventions; you can specify these either globally or for individual units

10.1.1 Project Files

A *project* is a specific set of values for these properties. You can define a project's settings in a *project file*, a text file with an Ada-like syntax; a property value is either a string or a list of strings. Properties that are not explicitly set receive default values. A project file may interrogate the values of *external variables* (user-defined command-line qualifiers or environment variables), and it may specify property settings conditionally, based on the value of such variables.

In simple cases, a project's source files depend only on other source files in the same project, or on the predefined libraries. ("Dependence" is in the technical sense; for example, one Ada unit "with"ing another.) However, the Project Manager also allows much more sophisticated arrangements, with the source files in one project depending on source files in other projects:

- One project can *import* other projects containing needed source files.
- You can organize GNAT projects in a hierarchy: a *child* project can extend a *parent* project, inheriting the parent's source files and optionally overriding any of them with alternative versions

More generally, the Project Manager lets you structure large development efforts into hierarchical subsystems, with build decisions deferred to the subsystem level and thus different compilation environments (qualifier settings) used for different subsystems.

The Project Manager is invoked through the '**-Pprojectfile**' qualifier to GNAT MAKE or to the **gnat** front driver. If you want to define (on the command line) an external variable that is queried by the project file, additionally use the '**-Xvbl=value**' qualifier. The Project

Manager parses and interprets the project file, and drives the invoked tool based on the project settings.

The Project Manager supports a wide range of development strategies, for systems of all sizes. Some typical practices that are easily handled:

- Using a common set of source files, but generating object files in different directories via different qualifier settings
- Using a mostly-shared set of source files, but with different versions of some unit or units

The destination of an executable can be controlled inside a project file using the ‘-o’ qualifier. In the absence of such a qualifier either inside the project file or on the command line, any executable files generated by GNAT MAKE will be placed in the directory `Exec_Dir` specified in the project file. If no `Exec_Dir` is specified, they will be placed in the object directory of the project.

You can use project files to achieve some of the effects of a source versioning system (for example, defining separate projects for the different sets of sources that comprise different releases) but the Project Manager is independent of any source configuration management tools that might be used by the developers.

The next section introduces the main features of GNAT's project facility through a sequence of examples; subsequent sections will present the syntax and semantics in more detail.

10.2 Examples of Project Files

This section illustrates some of the typical uses of project files and explains their basic structure and behavior.

10.2.1 Common Sources with Different Qualifiers and Different Output Directories

Assume that the Ada source files ‘PACK.ADS’, ‘PACK.ADB’, and ‘PROC.ADB’ are in the ‘/common’ directory. The file ‘PROC.ADB’ contains an Ada main subprogram `Proc` that "with"s package `Pack`. We want to compile these source files under two sets of qualifiers:

- When debugging, we want to pass the ‘-g’ qualifier to GNAT MAKE, and the ‘/CHECKS=ASSERTIONS’, ‘/CHECKS=OVERFLOW’, and ‘/CHECKS=ELABORATION’ qualifiers to the compiler; the compiler's output is to appear in ‘/common/debug’
- When preparing a release version, we want to pass the ‘/OPTIMIZE=ALL’ qualifier to the compiler; the compiler's output is to appear in ‘/common/release’

The GNAT project files shown below, respectively ‘debug.gpr’ and ‘release.gpr’ in the ‘/common’ directory, achieve these effects.

Diagrammatically:

```

/common
  debug.gpr
  release.gpr
  PACK.ADS
  PACK.ADB
  PROC.ADB

```



```

/common/debug {-g, /CHECKS=ASSERTIONS, /CHECKS=OVERFLOW, /CHECKS=ELABORATION}
  PROC.ALI, PROC.OBJ
  PACK.ALI, PACK.OBJ
/common/release {/OPTIMIZE=ALL}
  PROC.ALI, PROC.OBJ
  PACK.ALI, PACK.OBJ

```

Here are the project files:

```

project Debug is
  for Object_Dir use "debug";
  for Main use ("proc");

  package Builder is
    for Default_Qualifiers ("Ada") use ("-g");
  end Builder;

  package Compiler is
    for Default_Qualifiers ("Ada")
      use ("-fstack-check", "/CHECKS=ASSERTIONS", "/CHECKS=OVERFLOW", "/CHECKS=ELABORATION");
  end Compiler;
end Debug;

project Release is
  for Object_Dir use "release";
  for Exec_Dir use ".";
  for Main use ("proc");

  package Compiler is
    for Default_Qualifiers ("Ada") use ("/OPTIMIZE=ALL");
  end Compiler;
end Release;

```

The name of the project defined by ‘debug.gpr’ is “Debug” (case insensitive), and analogously the project defined by ‘release.gpr’ is “Release”. For consistency the file should have the same name as the project, and the project file’s extension should be “gpr”. These conventions are not required, but a warning is issued if they are not followed.

If the current directory is ‘/temp’, then the command

```
GNAT MAKE -P/common/debug.gpr
```

generates object and ALI files in ‘/common/debug’, and the `proc` executable also in ‘/common/debug’, using the qualifier settings defined in the project file.

Likewise, the command

```
GNAT MAKE -P/common/release.gpr
```

generates object and ALI files in ‘/common/release’, and the `proc` executable in ‘/common’, using the qualifier settings from the project file.

Source Files

If a project file does not explicitly specify a set of source directories or a set of source files, then by default the project’s source files are the Ada source files in the project file directory. Thus ‘PACK.ADS’, ‘PACK.ADB’, and ‘PROC.ADB’ are the source files for both projects.

Specifying the Object Directory

Several project properties are modeled by Ada-style *attributes*; you define the property by supplying the equivalent of an Ada attribute definition clause in the project file. A project’s

object directory is such a property; the corresponding attribute is `Object_Dir`, and its value is a string expression. A directory may be specified either as absolute or as relative; in the latter case, it is relative to the project file directory. Thus the compiler's output is directed to `'/common/debug'` (for the `Debug` project) and to `'/common/release'` (for the `Release` project). If `Object_Dir` is not specified, then the default is the project file directory.

Specifying the Exec Directory

- A project's exec directory is another property; the corresponding attribute is `Exec_Dir`, and its value is also a string expression, either specified as relative or absolute. If `Exec_Dir` is not specified, then the default is the object directory (which may also be the project file directory if attribute `Object_Dir` is not specified). Thus the executable is placed in `'/common/debug'` for the `Debug` project (attribute `Exec_Dir` not specified) and in `'/common'` for the `Release` project.

Project File Packages

- A GNAT tool integrated with the Project Manager is modeled by a corresponding package in the project file. The `Debug` project defines the packages `Builder` (for `GNAT MAKE`) and `Compiler`; the `Release` project defines only the `Compiler` package.

The Ada package syntax is not to be taken literally. Although packages in project files bear a surface resemblance to packages in Ada source code, the notation is simply a way to convey a grouping of properties for a named entity. Indeed, the package names permitted in project files are restricted to a predefined set, corresponding to the project-aware tools, and the contents of packages are limited to a small set of constructs. The packages in the example above contain attribute definitions.

Specifying Qualifier Settings

- Qualifier settings for a project-aware tool can be specified through attributes in the package corresponding to the tool. The example above illustrates one of the relevant attributes, `Default_Qualifiers`, defined in the packages in both project files. Unlike simple attributes like `Source_Dirs`, `Default_Qualifiers` is known as an *associative array*. When you define this attribute, you must supply an "index" (a literal string), and the effect of the attribute definition is to set the value of the "array" at the specified "index". For the `Default_Qualifiers` attribute, the index is a programming language (in our case, Ada), and the value specified (after `use`) must be a list of string expressions.

The attributes permitted in project files are restricted to a predefined set. Some may appear at project level, others in packages. For any attribute that is an associative array, the index must always be a literal string, but the restrictions on this string (e.g., a file name or a language name) depend on the individual attribute. Also depending on the attribute, its specified value will need to be either a string or a string list.

In the `Debug` project, we set the qualifiers for two tools, `GNAT MAKE` and the compiler, and thus we include corresponding packages, with each package defining the `Default_Qualifiers` attribute with index `"Ada"`. Note that the package corresponding to `GNAT MAKE` is named `Builder`. The `Release` project is similar, but with just the `Compiler` package.

In project `Debug` above the qualifiers starting with ‘-gnat’ that are specified in package `Compiler` could have been placed in package `Builder`, since `GNAT MAKE` transmits all such qualifiers to the compiler.

Main Subprograms

One of the properties of a project is its list of main subprograms (actually a list of names of source files containing main subprograms, with the file extension optional). This property is captured in the `Main` attribute, whose value is a list of strings. If a project defines the `Main` attribute, then you do not need to identify the main subprogram(s) when invoking `GNAT MAKE` (see Section 10.13.1 [GNAT MAKE and Project Files], page 132).

Source File Naming Conventions

Since the project files do not specify any source file naming conventions, the GNAT defaults are used. The mechanism for defining source file naming conventions – a package named `Naming` – will be described below (see Section 10.10 [Naming Schemes], page 129).

Source Language(s)

Since the project files do not specify a `Languages` attribute, by default the GNAT tools assume that the language of the project file is Ada. More generally, a project can comprise source files in Ada, C, and/or other languages.

10.2.2 Using External Variables

Instead of supplying different project files for debug and release, we can define a single project file that queries an external variable (set either on the command line or via an environment variable) in order to conditionally define the appropriate settings. Again, assume that the source files ‘`PACK.ADS`’, ‘`PACK.ADB`’, and ‘`PROC.ADB`’ are located in directory ‘`/common`’. The following project file, ‘`build.gpr`’, queries the external variable named `STYLE` and defines an object directory and qualifier settings based on whether the value is “`deb`” (debug) or “`rel`” (release), where the default is “`deb`”.

```
project Build is
  for Main use ("proc");

  type Style_Type is ("deb", "rel");
  Style : Style_Type := external ("STYLE", "deb");

  case Style is
    when "deb" =>
      for Object_Dir use "debug";

    when "rel" =>
      for Object_Dir use "release";
      for Exec_Dir use ".";
  end case;
```

```

package Builder is

  case Style is
    when "deb" =>
      for Default_Qualifiers ("Ada") use ("-g");
    end case;

end Builder;

package Compiler is

  case Style is
    when "deb" =>
      for Default_Qualifiers ("Ada") use ("/CHECKS=ASSERTIONS", "/CHECKS=OVERFLOW", "/CHECKS=ELABORATION");

    when "rel" =>
      for Default_Qualifiers ("Ada") use ("/OPTIMIZE=ALL");
    end case;

end Compiler;

end Build;

```

`Style_Type` is an example of a *string type*, which is the project file analog of an Ada enumeration type but containing string literals rather than identifiers. `Style` is declared as a variable of this type.

The form `external("STYLE", "deb")` is known as an *external reference*; its first argument is the name of an *external variable*, and the second argument is a default value to be used if the external variable doesn't exist. You can define an external variable on the command line via the `-X` qualifier, or you can use an environment variable as an external variable.

Each `case` construct is expanded by the Project Manager based on the value of `Style`. Thus the command

```
GNAT MAKE -P/common/build.gpr -XSTYLE=deb
```

is equivalent to the GNAT MAKE invocation using the project file `'debug.gpr'` in the earlier example. So is the command

```
GNAT MAKE -P/common/build.gpr
```

since `"deb"` is the default for `STYLE`.

Analogously,

```
GNAT MAKE -P/common/build.gpr -XSTYLE=rel
```

is equivalent to the GNAT MAKE invocation using the project file `'release.gpr'` in the earlier example.

10.2.3 Importing Other Projects

A compilation unit in a source file in one project may depend on compilation units in source files in other projects. To obtain this behavior, the dependent project must *import* the projects containing the needed source files. This effect is embodied in syntax similar to an Ada `with` clause, but the `"with"`ed entities are strings denoting project files.

As an example, suppose that the two projects `GUI_Proj` and `Comm_Proj` are defined in the project files `'gui_proj.gpr'` and `'comm_proj.gpr'` in directories `'/gui'` and `'/comm'`, respectively. Assume that the source files for `GUI_Proj` are `'GUI.ADS'` and `'GUI.ADB'`, and

that the source files for `Comm_Proj` are `'COMM.ADS'` and `'COMM.ADB'`, with each set of files located in its respective project file directory. Diagrammatically:

```

/gui
  gui_proj.gpr
  GUI.ADS
  GUI.ADB

/comm
  comm_proj.gpr
  COMM.ADS
  COMM.ADB

```

We want to develop an application in directory `'/app'` that "with"s the packages `GUI` and `Comm`, using the properties of the corresponding project files (e.g. the qualifier settings and object directory). Skeletal code for a main procedure might be something like the following:

```

with GUI, Comm;
procedure App_Main is
  ...
begin
  ...
end App_Main;

```

Here is a project file, `'app_proj.gpr'`, that achieves the desired effect:

```

with "/gui/gui_proj", "/comm/comm_proj";
project App_Proj is
  for Main use ("app_main");
end App_Proj;

```

Building an executable is achieved through the command:

```
GNAT MAKE -P/app/app_proj
```

which will generate the `app_main` executable in the directory where `'app_proj.gpr'` resides.

If an imported project file uses the standard extension (`gpr`) then (as illustrated above) the `with` clause can omit the extension.

Our example specified an absolute path for each imported project file. Alternatively, you can omit the directory if either

- The imported project file is in the same directory as the importing project file, or
- You have defined an environment variable `ADA_PROJECT_PATH` that includes the directory containing the needed project file.

Thus, if we define `ADA_PROJECT_PATH` to include `'/gui'` and `'/comm'`, then our project file `'app_proj.gpr'` could be written as follows:

```

with "gui_proj", "comm_proj";
project App_Proj is
  for Main use ("app_main");
end App_Proj;

```

Importing other projects raises the possibility of ambiguities. For example, the same unit might be present in different imported projects, or it might be present in both the importing project and an imported project. Both of these conditions are errors. Note that in the current version of the Project Manager, it is illegal to have an ambiguous unit even if the unit is never referenced by the importing project. This restriction may be relaxed in a future release.

10.2.4 Extending a Project

A common situation in large software systems is to have multiple implementations for a common interface; in Ada terms, multiple versions of a package body for the same specification. For example, one implementation might be safe for use in tasking programs, while another might only be used in sequential applications. This can be modeled in GNAT using the concept of *project extension*. If one project (the "child") *extends* another project (the "parent") then by default all source files of the parent project are inherited by the child, but the child project can override any of the parent's source files with new versions, and can also add new files. This facility is the project analog of extension in Object-Oriented Programming. Project hierarchies are permitted (a child project may be the parent of yet another project), and a project that inherits one project can also import other projects.

As an example, suppose that directory `'/seq'` contains the project file `'seq_proj.gpr'` and the source files `'PACK.ADS'`, `'PACK.ADB'`, and `'PROC.ADB'`:

```
/seq
  PACK.ADS
  PACK.ADB
  PROC.ADB
  seq_proj.gpr
```

Note that the project file can simply be empty (that is, no attribute or package is defined):

```
project Seq_Proj is
end Seq_Proj;
```

implying that its source files are all the Ada source files in the project directory.

Suppose we want to supply an alternate version of `'PACK.ADB'`, in directory `'/tasking'`, but use the existing versions of `'PACK.ADS'` and `'PROC.ADB'`. We can define a project `Tasking_Proj` that inherits `Seq_Proj`:

```
/tasking
  PACK.ADB
  tasking_proj.gpr
project Tasking_Proj extends "/seq/seq_proj" is
end Tasking_Proj;
```

The version of `'PACK.ADB'` used in a build depends on which project file is specified.

Note that we could have designed this using project import rather than project inheritance; a `base` project would contain the sources for `'PACK.ADS'` and `'PROC.ADB'`, a sequential project would import `base` and add `'PACK.ADB'`, and likewise a tasking project would import `base` and add a different version of `'PACK.ADB'`. The choice depends on whether other sources in the original project need to be overridden. If they do, then project extension is necessary, otherwise, importing is sufficient.

10.3 Project File Syntax

This section describes the structure of project files.

A project may be an *independent project*, entirely defined by a single project file. Any Ada source file in an independent project depends only on the predefined library and other Ada source files in the same project.

A project may also *depend on* other projects, in either or both of the following ways:

- It may import any number of projects
- It may extend at most one other project

The dependence relation is a directed acyclic graph (the subgraph reflecting the "extends" relation is a tree).

A project's *immediate sources* are the source files directly defined by that project, either implicitly by residing in the project file's directory, or explicitly through any of the source-related attributes described below. More generally, a project *proj*'s *sources* are the immediate sources of *proj* together with the immediate sources (unless overridden) of any project on which *proj* depends (either directly or indirectly).

10.3.1 Basic Syntax

As seen in the earlier examples, project files have an Ada-like syntax. The minimal project file is:

```
project Empty is  
  
end Empty;
```

The identifier `Empty` is the name of the project. This project name must be present after the reserved word `end` at the end of the project file, followed by a semi-colon.

Any name in a project file, such as the project name or a variable name, has the same syntax as an Ada identifier.

The reserved words of project files are the Ada reserved words plus `extends`, `external`, and `project`. Note that the only Ada reserved words currently used in project file syntax are:

- `case`
- `end`
- `for`
- `is`
- `others`
- `package`
- `renames`
- `type`
- `use`
- `when`
- `with`

Comments in project files have the same syntax as in Ada, two consecutive hyphens through the end of the line.

10.3.2 Packages

A project file may contain *packages*. The name of a package must be one of the identifiers (case insensitive) from a predefined list, and a package with a given name may only appear once in a project file. The predefined list includes the following packages:

- Naming
- Builder
- Compiler
- Binder
- Linker
- Finder
- Cross_Reference
- GNAT LIST

(The complete list of the package names and their attributes can be found in file 'PRJ-ATTR.ADB').

In its simplest form, a package may be empty:

```
project Simple is
  package Builder is
    end Builder;
end Simple;
```

A package may contain *attribute declarations*, *variable declarations* and *case constructions*, as will be described below.

When there is ambiguity between a project name and a package name, the name always designates the project. To avoid possible confusion, it is always a good idea to avoid naming a project with one of the names allowed for packages or any name that starts with **gnat**.

10.3.3 Expressions

An *expression* is either a *string expression* or a *string list expression*.

A *string expression* is either a *simple string expression* or a *compound string expression*.

A *simple string expression* is one of the following:

- A literal string; e.g. "comm/my_proj.gpr"
- A string-valued variable reference (see Section 10.3.5 [Variables], page 119)
- A string-valued attribute reference (see Section 10.3.6 [Attributes], page 120)
- An external reference (see Section 10.7 [External References in Project Files], page 127)

A *compound string expression* is a concatenation of string expressions, using "&"

```
Path & "/" & File_Name & ".ADS"
```

A *string list expression* is either a *simple string list expression* or a *compound string list expression*.

A *simple string list expression* is one of the following:

- A parenthesized list of zero or more string expressions, separated by commas


```
File_Names := (File_Name, "GNAT.ADC", File_Name & ".orig");
Empty_List := ();
```
- A string list-valued variable reference
- A string list-valued attribute reference

A *compound string list expression* is the concatenation (using "&") of a simple string list expression and an expression. Note that each term in a compound string list expression, except the first, may be either a string expression or a string list expression.


```

File_Name_List := () & File_Name; -- One string in this list
Extended_File_Name_List := File_Name_List & (File_Name & ".orig");
-- Two strings
Big_List := File_Name_List & Extended_File_Name_List;
-- Concatenation of two string lists: three strings
Illegal_List := "GNAT.ADC" & Extended_File_Name_List;
-- Illegal: must start with a string list

```

10.3.4 String Types

The value of a variable may be restricted to a list of string literals. The restricted list of string literals is given in a *string type declaration*.

Here is an example of a string type declaration:

```
type OS is ("NT", "nt", "Unix", "Linux", "other OS");
```

Variables of a string type are called *typed variables*; all other variables are called *untyped variables*. Typed variables are particularly useful in **case** constructions (see Section 10.3.8 [case Constructions], page 122).

A string type declaration starts with the reserved word **type**, followed by the name of the string type (case-insensitive), followed by the reserved word **is**, followed by a parenthesized list of one or more string literals separated by commas, followed by a semicolon.

The string literals in the list are case sensitive and must all be different. They may include any graphic characters allowed in Ada, including spaces.

A string type may only be declared at the project level, not inside a package.

A string type may be referenced by its name if it has been declared in the same project file, or by its project name, followed by a dot, followed by the string type name.

10.3.5 Variables

A variable may be declared at the project file level, or in a package. Here are some examples of variable declarations:

```

This_OS : OS := external ("OS"); -- a typed variable declaration
That_OS := "Linux";              -- an untyped variable declaration

```

A *typed variable declaration* includes the variable name, followed by a colon, followed by the name of a string type, followed by **:=**, followed by a simple string expression.

An *untyped variable declaration* includes the variable name, followed by **:=**, followed by an expression. Note that, despite the terminology, this form of "declaration" resembles more an assignment than a declaration in Ada. It is a declaration in several senses:

- The variable name does not need to be defined previously
- The declaration establishes the *kind* (string versus string list) of the variable, and later declarations of the same variable need to be consistent with this

A string variable declaration (typed or untyped) declares a variable whose value is a string. This variable may be used as a string expression.

```

File_Name      := "readme.txt";
Saved_File_Name := File_Name & ".saved";

```

A string list variable declaration declares a variable whose value is a list of strings. The list may contain any number (zero or more) of strings.

```

Empty_List := ();
List_With_One_Element := ("/STYLE=");
List_With_Two_Elements := List_With_One_Element & "/STYLE=GNAT";
Long_List := ("MAIN.ADA", "PACK1_.ADA", "PACK1.ADA", "PACK2_.ADA"
              "PACK2.ADA", "UTIL_.ADA", "UTIL.ADA");

```

The same typed variable may not be declared more than once at project level, and it may not be declared more than once in any package; it is in effect a constant or a readonly variable.

The same untyped variable may be declared several times. In this case, the new value replaces the old one, and any subsequent reference to the variable uses the new value. However, as noted above, if a variable has been declared as a string, all subsequent declarations must give it a string value. Similarly, if a variable has been declared as a string list, all subsequent declarations must give it a string list value.

A *variable reference* may take several forms:

- The simple variable name, for a variable in the current package (if any) or in the current project
- A context name, followed by a dot, followed by the variable name.

A *context* may be one of the following:

- The name of an existing package in the current project
- The name of an imported project of the current project
- The name of an ancestor project (i.e., a project extended by the current project, either directly or indirectly)
- An imported/parent project name, followed by a dot, followed by a package name

A variable reference may be used in an expression.

10.3.6 Attributes

A project (and its packages) may have *attributes* that define the project's properties. Some attributes have values that are strings; others have values that are string lists.

There are two categories of attributes: *simple attributes* and *associative arrays* (see Section 10.3.7 [Associative Array Attributes], page 122).

The names of the attributes are restricted; there is a list of project attributes, and a list of package attributes for each package. The names are not case sensitive.

The project attributes are as follows (all are simple attributes):

<i>Attribute Name</i>	<i>Value</i>
Source_Files	string list
Source_Dirs	string list
Source_List_File	string
Object_Dir	string
Exec_Dir	string
Main	string list
Languages	string list
Library_Dir	string
Library_Name	string

<code>Library_Kind</code>	string
<code>Library_Elaboration</code>	string
<code>Library_Version</code>	string

The attributes for package `Naming` are as follows (see Section 10.10 [Naming Schemes], page 129):

Attribute Name	Category	Index	Value
<code>Specification_Suffix</code>	associative	language name	string
<code>Implementation_Suffix</code>	array associative	language name	string
<code>Separate_Suffix</code>	array simple	n/a	string
<code>Casing</code>	attribute simple	n/a	string
<code>Dot_Replacement</code>	attribute simple	n/a	string
<code>Specification</code>	attribute associative	Ada unit name	string
<code>Implementation</code>	array associative	Ada unit name	string
<code>Specification_Exceptions</code>	array associative	language name	string list
<code>Implementation_Exceptions</code>	array associative	language name	string list
	array		

The attributes for package `Builder`, `Compiler`, `Binder`, `Linker`, `Cross_Reference`, and `Finder` are as follows (see Section 10.13.1.1 [Qualifiers and Project Files], page 132).

Attribute Name	Category	Index	Value
<code>Default_Qualifiers</code>	associative	language name	string list
<code>Qualifiers</code>	array associative	file name	string list
	array		

In addition, package `Builder` has a single string attribute `Local_Configuration_Pragmas` and package `Builder` has a single string attribute `Global_Configuration_Pragmas`.

The attribute for package `Glide` are not documented: they are for internal use only.

Each simple attribute has a default value: the empty string (for string-valued attributes) and the empty list (for string list-valued attributes).

Similar to variable declarations, an attribute declaration defines a new value for an attribute.

Examples of simple attribute declarations:

```
for Object_Dir use "objects";
for Source_Dirs use ("units", "test/drivers");
```

A *simple attribute declaration* starts with the reserved word `for`, followed by the name of the attribute, followed by the reserved word `use`, followed by an expression (whose kind depends on the attribute), followed by a semicolon.

Attributes may be referenced in expressions. The general form for such a reference is `<entity>'<attribute>`: the entity for which the attribute is defined, followed by an

apostrophe, followed by the name of the attribute. For associative array attributes, a literal string between parentheses need to be supplied as index.

Examples are:

```
project'Object_Dir
Naming'Dot_Replacement
Imported_Project'Source_Dirs
Imported_Project.Naming'Casing
Builder'Default_Qualifiers("Ada")
```

The entity may be:

- `project` for an attribute of the current project
- The name of an existing package of the current project
- The name of an imported project
- The name of a parent project (extended by the current project)
- An imported/parent project name, followed by a dot, followed by a package name

Example:

```
project Prj is
  for Source_Dirs use project'Source_Dirs & "units";
  for Source_Dirs use project'Source_Dirs & "test/drivers"
end Prj;
```

In the first attribute declaration, initially the attribute `Source_Dirs` has the default value: an empty string list. After this declaration, `Source_Dirs` is a string list of one element: "units". After the second attribute declaration `Source_Dirs` is a string list of two elements: "units" and "test/drivers".

Note: this example is for illustration only. In practice, the project file would contain only one attribute declaration:

```
for Source_Dirs use ("units", "test/drivers");
```

10.3.7 Associative Array Attributes

Some attributes are defined as *associative arrays*. An associative array may be regarded as a function that takes a string as a parameter and delivers a string or string list value as its result.

Here are some examples of associative array attribute declarations:

```
for Implementation ("main") use "MAIN.ADA";
for Qualifiers ("MAIN.ADA") use ("-v", "/REPORT_ERRORS=VERBOSE");
for Qualifiers ("MAIN.ADA") use Builder'Qualifiers ("MAIN.ADA") & "-g";
```

Like untyped variables and simple attributes, associative array attributes may be declared several times. Each declaration supplies a new value for the attribute, replacing the previous setting.

10.3.8 case Constructions

A `case` construction is used in a project file to effect conditional behavior. Here is a typical example:

```

project MyProj is
  type OS_Type is ("Linux", "Unix", "NT", "VMS");

  OS : OS_Type := external ("OS", "Linux");

  package Compiler is
    case OS is
      when "Linux" | "Unix" =>
        for Default_Qualifiers ("Ada") use ("-gnath");
      when "NT" =>
        for Default_Qualifiers ("Ada") use ("/POLLING_ENABLE");
      when others =>
        end case;
    end Compiler;
  end MyProj;

```

The syntax of a **case** construction is based on the Ada case statement (although there is no **null** construction for empty alternatives).

Following the reserved word **case** there is the case variable (a typed string variable), the reserved word **is**, and then a sequence of one or more alternatives. Each alternative comprises the reserved word **when**, either a list of literal strings separated by the **|** character or the reserved word **others**, and the **=>** token. Each literal string must belong to the string type that is the type of the case variable. An **others** alternative, if present, must occur last. The **end case;** sequence terminates the case construction.

After each **=>**, there are zero or more constructions. The only constructions allowed in a case construction are other case constructions and attribute declarations. String type declarations, variable declarations and package declarations are not allowed.

The value of the case variable is often given by an external reference (see Section 10.7 [External References in Project Files], page 127).

10.4 Objects and Sources in Project Files

Each project has exactly one object directory and one or more source directories. The source directories must contain at least one source file, unless the project file explicitly specifies that no source files are present (see Section 10.4.4 [Source File Names], page 124).

10.4.1 Object Directory

The object directory for a project is the directory containing the compiler's output (such as **'ALI'** files and object files) for the project's immediate sources. Note that for inherited sources (when extending a parent project) the parent project's object directory is used.

The object directory is given by the value of the attribute **Object_Dir** in the project file.

```

for Object_Dir use "objects";

```

The attribute *Object_Dir* has a string value, the path name of the object directory. The path name may be absolute or relative to the directory of the project file. This directory must already exist, and be readable and writable.

By default, when the attribute **Object_Dir** is not given an explicit value or when its value is the empty string, the object directory is the same as the directory containing the project file.

10.4.2 Exec Directory

The exec directory for a project is the directory containing the executables for the project's main subprograms.

The exec directory is given by the value of the attribute `Exec_Dir` in the project file.

```
for Exec_Dir use "executables";
```

The attribute *Exec_Dir* has a string value, the path name of the exec directory. The path name may be absolute or relative to the directory of the project file. This directory must already exist, and be writable.

By default, when the attribute `Exec_Dir` is not given an explicit value or when its value is the empty string, the exec directory is the same as the object directory of the project file.

10.4.3 Source Directories

The source directories of a project are specified by the project file attribute `Source_Dirs`.

This attribute's value is a string list. If the attribute is not given an explicit value, then there is only one source directory, the one where the project file resides.

A `Source_Dirs` attribute that is explicitly defined to be the empty list, as in

```
for Source_Dirs use ();
```

indicates that the project contains no source files.

Otherwise, each string in the string list designates one or more source directories.

```
for Source_Dirs use ("sources", "test/drivers");
```

If a string in the list ends with `/**`, then the directory whose path name precedes the two asterisks, as well as all its subdirectories (recursively), are source directories.

```
for Source_Dirs use ("/system/sources/**");
```

Here the directory `/system/sources` and all of its subdirectories (recursively) are source directories.

To specify that the source directories are the directory of the project file and all of its subdirectories, you can declare `Source_Dirs` as follows:

```
for Source_Dirs use ("/**");
```

Each of the source directories must exist and be readable.

10.4.4 Source File Names

In a project that contains source files, their names may be specified by the attributes `Source_Files` (a string list) or `Source_List_File` (a string). Source file names never include any directory information.

If the attribute `Source_Files` is given an explicit value, then each element of the list is a source file name.

```
for Source_Files use ("MAIN.ADB");
for Source_Files use ("MAIN.ADB", "PACK1.ADS", "PACK2.ADB");
```

If the attribute `Source_Files` is not given an explicit value, but the attribute `Source_List_File` is given a string value, then the source file names are contained in the text file

whose path name (absolute or relative to the directory of the project file) is the value of the attribute `Source_List_File`.

Each line in the file that is not empty or is not a comment contains a source file name. A comment line starts with two hyphens.

```
for Source_List_File use "source_list.txt";
```

By default, if neither the attribute `Source_Files` nor the attribute `Source_List_File` is given an explicit value, then each file in the source directories that conforms to the project's naming scheme (see Section 10.10 [Naming Schemes], page 129) is an immediate source of the project.

A warning is issued if both attributes `Source_Files` and `Source_List_File` are given explicit values. In this case, the attribute `Source_Files` prevails.

Each source file name must be the name of one and only one existing source file in one of the source directories.

A `Source_Files` attribute defined with an empty list as its value indicates that there are no source files in the project.

Except for projects that are clearly specified as containing no Ada source files (`Source_Dirs` or `Source_Files` specified as an empty list, or `Languages` specified without "Ada" in the list)

```
for Source_Dirs use ();
for Source_Files use ();
for Languages use ("C", "C++");
```

a project must contain at least one immediate source.

Projects with no source files are useful as template packages (see Section 10.8 [Packages in Project Files], page 127) for other projects; in particular to define a package `Naming` (see Section 10.10 [Naming Schemes], page 129).

10.5 Importing Projects

An immediate source of a project `P` may depend on source files that are neither immediate sources of `P` nor in the predefined library. To get this effect, `P` must *import* the projects that contain the needed source files.

```
with "project1", "utilities.gpr";
with "/namings/apex.gpr";
project Main is
...
```

As can be seen in this example, the syntax for importing projects is similar to the syntax for importing compilation units in Ada. However, project files use literal strings instead of names, and the `with` clause identifies project files rather than packages.

Each literal string is the file name or path name (absolute or relative) of a project file. If a string is simply a file name, with no path, then its location is determined by the *project path*:

- If the environment variable `ADA_PROJECT_PATH` exists, then the project path includes all the directories in this environment variable, plus the directory of the project file.
- If the environment variable `ADA_PROJECT_PATH` does not exist, then the project path contains only one directory, namely the one where the project file is located.

If a relative pathname is used as in

```
with "tests/proj";
```

then the path is relative to the directory where the importing project file is located. Any symbolic link will be fully resolved in the directory of the importing project file before the imported project file is looked up.

When the `with`'ed project file name does not have an extension, the default is `'.gpr'`. If a file with this extension is not found, then the file name as specified in the `with` clause (no extension) will be used. In the above example, if a file `project1.gpr` is found, then it will be used; otherwise, if a file `project1` exists then it will be used; if neither file exists, this is an error.

A warning is issued if the name of the project file does not match the name of the project; this check is case insensitive.

Any source file that is an immediate source of the imported project can be used by the immediate sources of the importing project, and recursively. Thus if `A` imports `B`, and `B` imports `C`, the immediate sources of `A` may depend on the immediate sources of `C`, even if `A` does not import `C` explicitly. However, this is not recommended, because if and when `B` ceases to import `C`, some sources in `A` will no longer compile.

A side effect of this capability is that cyclic dependences are not permitted: if `A` imports `B` (directly or indirectly) then `B` is not allowed to import `A`.

10.6 Project Extension

During development of a large system, it is sometimes necessary to use modified versions of some of the source files without changing the original sources. This can be achieved through a facility known as *project extension*.

```
project Modified_Uutilities extends "/baseline/utilities.gpr" is ...
```

The project file for the project being extended (the *parent*) is identified by the literal string that follows the reserved word `extends`, which itself follows the name of the extending project (the *child*).

By default, a child project inherits all the sources of its parent. However, inherited sources can be overridden: a unit with the same name as one in the parent will hide the original unit. Inherited sources are considered to be sources (but not immediate sources) of the child project; see Section 10.3 [Project File Syntax], page 116.

An inherited source file retains any qualifiers specified in the parent project.

For example if the project `Utilities` contains the specification and the body of an Ada package `Util_IO`, then the project `Modified_Uutilities` can contain a new body for package `Util_IO`. The original body of `Util_IO` will not be considered in program builds. However, the package specification will still be found in the project `Utilities`.

A child project can have only one parent but it may import any number of other projects.

A project is not allowed to import directly or indirectly at the same time a child project and any of its ancestors.

10.7 External References in Project Files

A project file may contain references to external variables; such references are called *external references*.

An external variable is either defined as part of the environment (an environment variable in Unix, for example) or else specified on the command line via the `'-Xvbl=value'` qualifier. If both, then the command line value is used.

An external reference is denoted by the built-in function `external`, which returns a string value. This function has two forms:

- `external (external_variable_name)`
- `external (external_variable_name, default_value)`

Each parameter must be a string literal. For example:

```
external ("USER")
external ("OS", "Linux")
```

In the form with one parameter, the function returns the value of the external variable given as parameter. If this name is not present in the environment, then the returned value is an empty string.

In the form with two string parameters, the second parameter is the value returned when the variable given as the first parameter is not present in the environment. In the example above, if "OS" is not the name of an environment variable and is not passed on the command line, then the returned value will be "Linux".

An external reference may be part of a string expression or of a string list expression, to define variables or attributes.

```
type Mode_Type is ("Debug", "Release");
Mode : Mode_Type := external ("MODE");
case Mode is
  when "Debug" =>
    ...
```

10.8 Packages in Project Files

The *package* is the project file feature that defines the settings for project-aware tools. For each such tool you can declare a corresponding package; the names for these packages are preset (see Section 10.3.2 [Packages], page 117) but are not case sensitive. A package may contain variable declarations, attribute declarations, and case constructions.

```
project Proj is
  package Builder is -- used by GNAT MAKE
    for Default_Qualifiers ("Ada") use ("-v", "-g");
  end Builder;
end Proj;
```

A package declaration starts with the reserved word `package`, followed by the package name (case insensitive), followed by the reserved word `is`. It ends with the reserved word `end`, followed by the package name, finally followed by a semi-colon.

Most of the packages have an attribute `Default_Qualifiers`. This attribute is an associative array, and its value is a string list. The index of the associative array is the

name of a programming language (case insensitive). This attribute indicates the qualifier or qualifiers to be used with the corresponding tool.

Some packages also have another attribute, **Qualifiers**, an associative array whose value is a string list. The index is the name of a source file. This attribute indicates the qualifier or qualifiers to be used by the corresponding tool when dealing with this specific file.

Further information on these qualifier-related attributes is found in Section 10.13.1.1 [Qualifiers and Project Files], page 132.

A package may be declared as a *renaming* of another package; e.g., from the project file for an imported project.

```
with "/global/apex.gpr";
project Example is
  package Naming renames Apex.Naming;
  ...
end Example;
```

Packages that are renamed in other project files often come from project files that have no sources: they are just used as templates. Any modification in the template will be reflected automatically in all the project files that rename a package from the template.

In addition to the tool-oriented packages, you can also declare a package named **Naming** to establish specialized source file naming conventions (see Section 10.10 [Naming Schemes], page 129).

10.9 Variables from Imported Projects

An attribute or variable defined in an imported or parent project can be used in expressions in the importing / extending project. Such an attribute or variable is prefixed with the name of the project and (if relevant) the name of package where it is defined.

```
with "imported";
project Main extends "base" is
  Var1 := Imported.Var;
  Var2 := Base.Var & ".new";

  package Builder is
    for Default_Qualifiers ("Ada") use Imported.Builder.Ada_Qualifiers &
      "/STYLE=GNAT" & "-v";
  end Builder;

  package Compiler is
    for Default_Qualifiers ("Ada") use Base.Compiler.Ada_Qualifiers;
  end Compiler;
end Main;
```

In this example:

- **Var1** is a copy of the variable **Var** defined in the project file `"imported.gpr"`
- the value of **Var2** is a copy of the value of variable **Var** defined in the project file `'base.gpr'`, concatenated with `".new"`
- attribute **Default_Qualifiers ("Ada")** in package **Builder** is a string list that includes in its value a copy of variable **Ada_Qualifiers** defined in the **Builder** package in project file `'imported.gpr'` plus two new elements: `"/STYLE=GNAT"` and `"-v"`;

- attribute `Default_Qualifiers` ("Ada") in package `Compiler` is a copy of the variable `Ada_Qualifiers` defined in the `Compiler` package in project file 'base.gpr', the project being extended.

10.10 Naming Schemes

Sometimes an Ada software system is ported from a foreign compilation environment to GNAT, with file names that do not use the default GNAT conventions. Instead of changing all the file names (which for a variety of reasons might not be possible), you can define the relevant file naming scheme in the `Naming` package in your project file. For example, the following package models the Apex file naming rules:

```
package Naming is
  for Casing use "lowercase";
  for Dot_Replacement use ".";
  for Specification_Suffix ("Ada") use ".1.ADA";
  for Implementation_Suffix ("Ada") use ".2.ADA";
end Naming;
```

You can define the following attributes in package `Naming`:

Casing This must be a string with one of the three values "lowercase", "uppercase" or "mixedcase"; these strings are case insensitive.

If *Casing* is not specified, then the default is "lowercase".

Dot_Replacement

This must be a string whose value satisfies the following conditions:

- It must not be empty
- It cannot start or end with an alphanumeric character
- It cannot be a single underscore
- It cannot start with an underscore followed by an alphanumeric
- It cannot contain a dot '.' except if it the entire string is "."

If *Dot_Replacement* is not specified, then the default is "-".

Specification_Suffix

This is an associative array (indexed by the programming language name, case insensitive) whose value is a string that must satisfy the following conditions:

- It must not be empty
- It cannot start with an alphanumeric character
- It cannot start with an underscore followed by an alphanumeric character

If *Specification_Suffix* ("Ada") is not specified, then the default is ".ADS".

Implementation_Suffix

This is an associative array (indexed by the programming language name, case insensitive) whose value is a string that must satisfy the following conditions:

- It must not be empty
- It cannot start with an alphanumeric character
- It cannot start with an underscore followed by an alphanumeric character

- It cannot be a suffix of `Specification_Suffix`

If `Implementation_Suffix` ("Ada") is not specified, then the default is ".ADB".

Separate_Suffix

This must be a string whose value satisfies the same conditions as `Implementation_Suffix`.

If `Separate_Suffix` ("Ada") is not specified, then it defaults to same value as `Implementation_Suffix` ("Ada").

Specification

You can use the `Specification` attribute, an associative array, to define the source file name for an individual Ada compilation unit's spec. The array index must be a string literal that identifies the Ada unit (case insensitive). The value of this attribute must be a string that identifies the file that contains this unit's spec (case sensitive or insensitive depending on the operating system).

```
for Specification ("MyPack.MyChild") use "mypack.mychild.spec";
```

Implementation

You can use the `Implementation` attribute, an associative array, to define the source file name for an individual Ada compilation unit's body (possibly a subunit). The array index must be a string literal that identifies the Ada unit (case insensitive). The value of this attribute must be a string that identifies the file that contains this unit's body or subunit (case sensitive or insensitive depending on the operating system).

```
for Implementation ("MyPack.MyChild") use "mypack.mychild.body";
```

10.11 Library Projects

Library projects are projects whose object code is placed in a library. (Note that this facility is not yet supported on all platforms)

To create a library project, you need to define in its project file two project-level attributes: `Library_Name` and `Library_Dir`. Additionally, you may define the library-related attributes `Library_Kind`, `Library_Version` and `Library_Elaboration`.

The `Library_Name` attribute has a string value that must start with a letter and include only letters and digits.

The `Library_Dir` attribute has a string value that designates the path (absolute or relative) of the directory where the library will reside. It must designate an existing directory, and this directory needs to be different from the project's object directory. It also needs to be writable.

If both `Library_Name` and `Library_Dir` are specified and are legal, then the project file defines a library project. The optional library-related attributes are checked only for such project files.

The `Library_Kind` attribute has a string value that must be one of the following (case insensitive): "static", "dynamic" or "relocatable". If this attribute is not specified, the library is a static library. Otherwise, the library may be dynamic or relocatable. Depending on the operating system, there may or may not be a distinction between dynamic and relocatable libraries. For example, on Unix there is no such distinction.

The `Library_Version` attribute has a string value whose interpretation is platform dependent. On Unix, it is used only for dynamic/relocatable libraries as the internal name of the library (the "`soname`"). If the library file name (built from the `Library_Name`) is different from the `Library_Version`, then the library file will be a symbolic link to the actual file whose name will be `Library_Version`.

Example (on Unix):

```
project Plib is

  Version := "1";

  for Library_Dir use "lib_dir";
  for Library_Name use "dummy";
  for Library_Kind use "relocatable";
  for Library_Version use "libdummy.so." & Version;

end Plib;
```

Directory '`lib_dir`' will contain the internal library file whose name will be '`libdummy.so.1`', and '`libdummy.so`' will be a symbolic link to '`libdummy.so.1`'.

When GNAT MAKE detects that a project file (not the main project file) is a library project file, it will check all immediate sources of the project and rebuild the library if any of the sources have been recompiled. All '`ALI`' files will also be copied from the object directory to the library directory. To build executables, GNAT MAKE will use the library rather than the individual object files.

10.12 Qualifiers Related to Project Files

The following qualifiers are used by GNAT tools that support project files:

'-Pproject'

Indicates the name of a project file. This project file will be parsed with the verbosity indicated by '`-vPx`', if any, and using the external references indicated by '`-X`' qualifiers, if any.

There must be only one '`-P`' qualifier on the command line.

Since the Project Manager parses the project file only after all the qualifiers on the command line are checked, the order of the qualifiers '`-P`', '`-Vpx`' or '`-X`' is not significant.

'-Xname=value'

Indicates that external variable *name* has the value *value*. The Project Manager will use this value for occurrences of `external(name)` when parsing the project file.

If *name* or *value* includes a space, then *name=value* should be put between quotes.

```
-XOS=NT
-X"user=John Doe"
```

Several '`-X`' qualifiers can be used simultaneously. If several '`-X`' qualifiers specify the same *name*, only the last one is used.

An external variable specified with a '`-X`' qualifier takes precedence over the value of the same name in the environment.

‘-vPx’ Indicates the verbosity of the parsing of GNAT project files. ‘-vP0’ means Default (no output for syntactically correct project files); ‘-vP1’ means Medium; ‘-vP2’ means High. The default is Default. If several ‘-vPx’ qualifiers are present, only the last one is used.

10.13 Tools Supporting Project Files

10.13.1 GNAT MAKE and Project Files

This section covers two topics related to GNAT MAKE and project files: defining qualifiers for GNAT MAKE and for the tools that it invokes; and the use of the **Main** attribute.

10.13.1.1 Qualifiers and Project Files

For each of the packages **Builder**, **Compiler**, **Binder**, and **Linker**, you can specify a **Default_Qualifiers** attribute, a **Qualifiers** attribute, or both; as their names imply, these qualifier-related attributes affect which qualifiers are used for which files when GNAT MAKE is invoked. As will be explained below, these package-contributed qualifiers precede the qualifiers passed on the GNAT MAKE command line.

The **Default_Qualifiers** attribute is an associative array indexed by language name (case insensitive) and returning a string list. For example:

```
package Compiler is
  for Default_Qualifiers ("Ada") use ("/STYLE=", "-v");
end Compiler;
```

The **Qualifiers** attribute is also an associative array, indexed by a file name (which may or may not be case sensitive, depending on the operating system) and returning a string list. For example:

```
package Builder is
  for Qualifiers ("MAIN1.ADB") use ("/OPTIMIZE=ALL");
  for Qualifiers ("MAIN2.ADB") use ("-g");
end Builder;
```

For the **Builder** package, the file names should designate source files for main subprograms. For the **Binder** and **Linker** packages, the file names should designate ‘**ALI**’ or source files for main subprograms. In each case just the file name (without explicit extension) is acceptable.

For each tool used in a program build (GNAT MAKE, the compiler, the binder, and the linker), its corresponding package *contributes* a set of qualifiers for each file on which the tool is invoked, based on the qualifier-related attributes defined in the package. In particular, the qualifiers that each of these packages contributes for a given file *f* comprise:

- the value of attribute **Qualifiers** (*f*), if it is specified in the package for the given file,
- otherwise, the value of **Default_Qualifiers** ("Ada"), if it is specified in the package.

If neither of these attributes is defined in the package, then the package does not contribute any qualifiers for the given file.

When **GNAT MAKE** is invoked on a file, the qualifiers comprise two sets, in the following order: those contributed for the file by the **Builder** package; and the qualifiers passed on the command line.

When **GNAT MAKE** invokes a tool (compiler, binder, linker) on a file, the qualifiers passed to the tool comprise three sets, in the following order:

1. the applicable qualifiers contributed for the file by the **Builder** package in the project file supplied on the command line;
2. those contributed for the file by the package (in the relevant project file – see below) corresponding to the tool; and
3. the applicable qualifiers passed on the command line.

The term *applicable qualifiers* reflects the fact that **GNAT MAKE** qualifiers may or may not be passed to individual tools, depending on the individual qualifier.

GNAT MAKE may invoke the compiler on source files from different projects. The Project Manager will use the appropriate project file to determine the **Compiler** package for each source file being compiled. Likewise for the **Binder** and **Linker** packages.

As an example, consider the following package in a project file:

```
project Proj1 is
  package Compiler is
    for Default_Qualifiers ("Ada") use ("-g");
    for Qualifiers ("A.ADB") use ("/OPTIMIZE=SOME");
    for Qualifiers ("B.ADB") use ("/OPTIMIZE=ALL", "/STYLE=");
  end Compiler;
end Proj1;
```

If **GNAT MAKE** is invoked with this project file, and it needs to compile, say, the files ‘A.ADB’, ‘B.ADB’, and ‘C.ADB’, then ‘A.ADB’ will be compiled with the qualifier ‘/OPTIMIZE=SOME’, ‘B.ADB’ with qualifiers ‘/OPTIMIZE=ALL’ and ‘/STYLE=’, and ‘C.ADB’ with ‘-g’.

Another example illustrates the ordering of the qualifiers contributed by different packages:

```
project Proj2 is
  package Builder is
    for Qualifiers ("MAIN.ADB") use ("-g", "/OPTIMIZE=SOME", "-f");
  end Builder;
  package Compiler is
    for Qualifiers ("MAIN.ADB") use ("/OPTIMIZE=ALL");
  end Compiler;
end Proj2;
```

If you issue the command:

```
GNAT MAKE -PProj2 /OPTIMIZE=NONE main
```

then the compiler will be invoked on ‘MAIN.ADB’ with the following sequence of qualifiers

```
-g /OPTIMIZE=SOME /OPTIMIZE=ALL /OPTIMIZE=NONE
```

with the last ‘-O’ qualifier having precedence over the earlier ones; several other qualifiers (such as ‘-c’) are added implicitly.

The qualifiers ‘-g’ and ‘/OPTIMIZE=SOME’ are contributed by package **Builder**, ‘/OPTIMIZE=ALL’ is contributed by the package **Compiler** and ‘/OPTIMIZE=NONE’ comes from the command line.

The ‘-g’ qualifier will also be passed in the invocation of **GNAT LINK**.

A final example illustrates qualifier contributions from packages in different project files:

```

project Proj3 is
  for Source_Files use ("PACK.ADS", "PACK.ADB");
  package Compiler is
    for Default_Qualifiers ("Ada") use ("/CHECKS=ASSERTIONS");
  end Compiler;
end Proj3;

with "Proj3";
project Proj4 is
  for Source_Files use ("FOO_MAIN.ADB", "BAR_MAIN.ADB");
  package Builder is
    for Qualifiers ("FOO_MAIN.ADB") use ("-s", "-g");
  end Builder;
end Proj4;

-- Ada source file:
with Pack;
procedure Foo_Main is
  ...
end Foo_Main;

```

If the command is

```
GNAT MAKE -PProj4 FOO_MAIN.ADB /COMPILER_QUALIFIERS /CHECKS=OVERFLOW
```

then the qualifiers passed to the compiler for 'FOO_MAIN.ADB' are '-g' (contributed by the package Proj4.Builder) and '/CHECKS=OVERFLOW' (passed on the command line). When the imported package Pack is compiled, the qualifiers used are '-g' from Proj4.Builder, '/CHECKS=ASSERTIONS' (contributed from package Proj3.Compiler, and '/CHECKS=OVERFLOW' from the command line.

10.13.1.2 Project Files and Main Subprograms

When using a project file, you can invoke **GNAT MAKE** with several main subprograms, by specifying their source files on the command line. Each of these needs to be an immediate source file of the project.

```
GNAT MAKE -Pprj main1 main2 main3
```

When using a project file, you can also invoke **GNAT MAKE** without explicitly specifying any main, and the effect depends on whether you have defined the **Main** attribute. This attribute has a string list value, where each element in the list is the name of a source file (the file extension is optional) containing a main subprogram.

If the **Main** attribute is defined in a project file as a non-empty string list and the qualifier '-u' is not used on the command line, then invoking **GNAT MAKE** with this project file but without any main on the command line is equivalent to invoking **GNAT MAKE** with all the file names in the **Main** attribute on the command line.

Example:

```

project Prj is
  for Main use ("main1", "main2", "main3");
end Prj;

```

With this project file, "GNAT MAKE -Pprj" is equivalent to "GNAT MAKE -Pprj main1 main2 main3".

When the project attribute **Main** is not specified, or is specified as an empty string list, or when the qualifier '-u' is used on the command line, then invoking **GNAT MAKE** with no main on the command line will result in all immediate sources of the project file being checked,

and potentially recompiled. Depending on the presence of the qualifier ‘-u’, sources from other project files on which the immediate sources of the main project file depend are also checked and potentially recompiled. In other words, the ‘-u’ qualifier is applied to all of the immediate sources of the main project file.

10.13.2 The GNAT Driver and Project Files

A number of GNAT tools, other than GNAT MAKE are project-aware: GNAT BIND, GNAT FIND, GNAT LINK, GNAT LIST and GNAT XREF. However, none of these tools can be invoked directly with a project file qualifier (-P). They need to be invoked through the **gnat** driver.

The **gnat** driver is a front-end that accepts a number of commands and call the corresponding tool. It has been designed initially for VMS to convert VMS style qualifiers to Unix style qualifiers, but it is now available to all the GNAT supported platforms.

On non VMS platforms, the **gnat** driver accepts the following commands (case insensitive):

- BIND to invoke GNAT BIND
- CHOP to invoke GNAT CHOP
- COMP or COMPILE to invoke the compiler
- ELIM to invoke GNAT ELIM
- FIND to invoke GNAT FIND
- KR or KRUNCH to invoke GNAT KRUNCH
- LINK to invoke GNAT LINK
- LS or LIST to invoke GNAT LIST
- MAKE to invoke GNAT MAKE
- NAME to invoke **gnatname**
- PREP or PREPROCESS to invoke GNAT PREPROCESS
- PSTA or STANDARD to invoke GNAT STANDARD
- STUB to invoke GNAT STUB
- XREF to invoke GNAT XREF

Note that the compiler is invoked using the command **GNAT MAKE -f -u**.

Following the command, you may put qualifiers and arguments for the invoked tool.

```
gnat bind -C MAIN.ALI
gnat ls -a main
gnat chop foo.txt
```

In addition, for command BIND, FIND, LS or LIST, LINK and XREF, the project file related qualifiers (-P, -X and -vPx) may be used in addition to the qualifiers of the invoking tool.

For each of these command, there is possibly a package in the main project that corresponds to the invoked tool.

- package **Binder** for command BIND (invoking GNAT BIND)
- package **Finder** for command FIND (invoking GNAT FIND)
- package **GNAT LIST** for command LS or LIST (invoking GNAT LIST)

- package `Linker` for command `LINK` (invoking `GNAT LINK`)
- package `Cross_Reference` for command `XREF` (invoking `GNAT LINK`)

Package `GNAT LIST` has a unique attribute `Qualifiers`, a simple variable with a string list value. It contains qualifiers for the invocation of `GNAT LIST`.

```
project Proj1 is
  package GNAT_LIST is
    for Qualifiers use ("-a", "-v");
  end GNAT_LIST;
end Proj1;
```

All other packages contains a qualifier `Default_Qualifiers`, an associative array, indexed by the programming language (case insensitive) and having a string list value. `Default_Qualifiers ("Ada")` contains the qualifiers for the invocation of the tool corresponding to the package.

```
project Proj is

  for Source_Dirs use ("./**");

  package GNAT_LIST is
    for Qualifiers use ("-a", "-v");
  end GNAT_LIST;

  package Binder is
    for Default_Qualifiers ("Ada") use ("-C", "-e");
  end Binder;

  package Linker is
    for Default_Qualifiers ("Ada") use ("-C");
  end Linker;

  package Finder is
    for Default_Qualifiers ("Ada") use ("-a", "-f");
  end Finder;

  package Cross_Reference is
    for Default_Qualifiers ("Ada") use ("-a", "-f", "-d", "-u");
  end Cross_Reference;
end Proj;
```

With the above project file, commands such as

```
gnat ls -Pproj main
gnat xref -Pproj main
gnat bind -Pproj MAIN.ALI
```

will set up the environment properly and invoke the tool with the qualifiers found in the package corresponding to the tool.

10.14 An Extended Example

Suppose that we have two programs, *prog1* and *prog2*, with the sources in the respective directories. We would like to build them with a single `GNAT MAKE` command, and we would like to place their object files into `‘.build’` subdirectories of the source directories. Furthermore, we would like to have two separate subdirectories in `‘.build’` – `‘release’`

and ‘debug’ – which will contain the object files compiled with different set of compilation flags.

In other words, we have the following structure:

```
main
|- prog1
|   |- .build
|       | debug
|       | release
|- prog2
|   |- .build
|       | debug
|       | release
```

Here are the project files that we need to create in a directory ‘main’ to maintain this structure:

1. We create a **Common** project with a package **Compiler** that specifies the compilation qualifiers:

```
File "common.gpr":
project Common is

    for Source_Dirs use (); -- No source files
    type Build_Type is ("release", "debug");
    Build : Build_Type := External ("BUILD", "debug");
    package Compiler is
        case Build is
            when "release" =>
                for Default_Qualifiers ("Ada") use ("/OPTIMIZE=ALL");
            when "debug" =>
                for Default_Qualifiers ("Ada") use ("-g");
        end case;
    end Compiler;

end Common;
```

2. We create separate projects for the two programs:

```
File "prog1.gpr":

with "common";
project Prog1 is

    for Source_Dirs use ("prog1");
    for Object_Dir use "prog1/.build/" & Common.Build;

    package Compiler renames Common.Compiler;

end Prog1;

File "prog2.gpr":

with "common";
project Prog2 is

    for Source_Dirs use ("prog2");
    for Object_Dir use "prog2/.build/" & Common.Build;

    package Compiler renames Common.Compiler;

end Prog2;
```

3. We create a wrapping project *Main*:

```
File "main.gpr":

with "common";
with "prog1";
with "prog2";
project Main is

    package Compiler renames Common.Compiler;

end Main;
```

4. Finally we need to create a dummy procedure that **withs** (either explicitly or implicitly) all the sources of our two programs.

Now we can build the programs using the command

```
GNAT MAKE -Pmain dummy
```

for the Debug mode, or

```
GNAT MAKE -Pmain -XBUILD=release
```

for the Release mode.

10.15 Project File Complete Syntax

```
project ::=
  context_clause project_declaration

context_clause ::=
  {with_clause}

with_clause ::=
  with literal_string { , literal_string } ;

project_declaration ::=
  project <project_>simple_name [ extends literal_string ] is
    {declarative_item}
  end <project_>simple_name;

declarative_item ::=
  package_declaration |
  typed_string_declaration |
  other_declarative_item

package_declaration ::=
  package <package_>simple_name package_completion

package_completion ::=
  package_body | package_renaming

package body ::=
  is
    {other_declarative_item}
  end <package_>simple_name ;

package_renaming ::=
  renames <project_>simple_name.<package_>simple_name ;
```

```

typed_string_declaration ::=
  type <typed_string>_simple_name is
    ( literal_string {, literal_string} );

other_declarative_item ::=
  attribute_declaration |
  typed_variable_declaration |
  variable_declaration |
  case_construction

attribute_declaration ::=
  for attribute use expression ;

attribute ::=
  <simple_attribute>_simple_name |
  <associative_array_attribute>_simple_name ( literal_string )

typed_variable_declaration ::=
  <typed_variable>_simple_name : <typed_string>name := string_expression ;

variable_declaration ::=
  <variable>_simple_name := expression;

expression ::=
  term {& term}

term ::=
  literal_string |
  string_list |
  <variable>_name |
  external_value |
  attribute_reference

literal_string ::=
  (same as Ada)

string_list ::=
  ( <string>_expression { , <string>_expression } )

external_value ::=
  external ( literal_string [, literal_string] )

attribute_reference ::=
  attribute_parent ' <simple_attribute>_simple_name [ ( literal_string ) ]

attribute_parent ::=
  project |
  <project_or_package>_simple_name |
  <project>_simple_name . <package>_simple_name

case_construction ::=
  case <typed_variable>_name is
    {case_item}
  end case ;

case_item ::=
  when discrete_choice_list => {case_construction | attribute_declaration}

```

```
discrete_choice_list ::=
  literal_string { | literal_string }

name ::=
  simple_name { . simple_name }

simple_name ::=
  identifier (same as Ada)
```

11 Elaboration Order Handling in GNAT

This chapter describes the handling of elaboration code in Ada 95 and in GNAT, and discusses how the order of elaboration of program units can be controlled in GNAT, either automatically or with explicit programming features.

11.1 Elaboration Code in Ada 95

Ada 95 provides rather general mechanisms for executing code at elaboration time, that is to say before the main program starts executing. Such code arises in three contexts:

Initializers for variables.

Variables declared at the library level, in package specs or bodies, can require initialization that is performed at elaboration time, as in:

```
Sqrt_Half : Float := Sqrt (0.5);
```

Package initialization code

Code in a **BEGIN-END** section at the outer level of a package body is executed as part of the package body elaboration code.

Library level task allocators

Tasks that are declared using task allocators at the library level start executing immediately and hence can execute at elaboration time.

Subprogram calls are possible in any of these contexts, which means that any arbitrary part of the program may be executed as part of the elaboration code. It is even possible to write a program which does all its work at elaboration time, with a null main program, although stylistically this would usually be considered an inappropriate way to structure a program.

An important concern arises in the context of elaboration code: we have to be sure that it is executed in an appropriate order. What we have is a series of elaboration code sections, potentially one section for each unit in the program. It is important that these execute in the correct order. Correctness here means that, taking the above example of the declaration of **Sqrt_Half**, if some other piece of elaboration code references **Sqrt_Half**, then it must run after the section of elaboration code that contains the declaration of **Sqrt_Half**.

There would never be any order of elaboration problem if we made a rule that whenever you **with** a unit, you must elaborate both the spec and body of that unit before elaborating the unit doing the **with**'ing:

```
with Unit_1;  
package Unit_2 is ...
```

would require that both the body and spec of **Unit_1** be elaborated before the spec of **Unit_2**. However, a rule like that would be far too restrictive. In particular, it would make it impossible to have routines in separate packages that were mutually recursive.

You might think that a clever enough compiler could look at the actual elaboration code and determine an appropriate correct order of elaboration, but in the general case, this is not possible. Consider the following example.

In the body of **Unit_1**, we have a procedure **Func_1** that references the variable **Sqrt_1**, which is declared in the elaboration code of the body of **Unit_1**:

```
Sqrt_1 : Float := Sqrt (0.1);
```

The elaboration code of the body of **Unit_1** also contains:

```
if expression_1 = 1 then
  Q := Unit_2.Func_2;
end if;
```

Unit_2 is exactly parallel, it has a procedure **Func_2** that references the variable **Sqrt_2**, which is declared in the elaboration code of the body **Unit_2**:

```
Sqrt_2 : Float := Sqrt (0.1);
```

The elaboration code of the body of **Unit_2** also contains:

```
if expression_2 = 2 then
  Q := Unit_1.Func_1;
end if;
```

Now the question is, which of the following orders of elaboration is acceptable:

```
Spec of Unit_1
Spec of Unit_2
Body of Unit_1
Body of Unit_2
```

or

```
Spec of Unit_2
Spec of Unit_1
Body of Unit_2
Body of Unit_1
```

If you carefully analyze the flow here, you will see that you cannot tell at compile time the answer to this question. If **expression_1** is not equal to 1, and **expression_2** is not equal to 2, then either order is acceptable, because neither of the function calls is executed. If both tests evaluate to true, then neither order is acceptable and in fact there is no correct order.

If one of the two expressions is true, and the other is false, then one of the above orders is correct, and the other is incorrect. For example, if **expression_1** = 1 and **expression_2**

`/= 2`, then the call to `Func_2` will occur, but not the call to `Func_1`. This means that it is essential to elaborate the body of `Unit_1` before the body of `Unit_2`, so the first order of elaboration is correct and the second is wrong.

By making `expression_1` and `expression_2` depend on input data, or perhaps the time of day, we can make it impossible for the compiler or binder to figure out which of these expressions will be true, and hence it is impossible to guarantee a safe order of elaboration at run time.

11.2 Checking the Elaboration Order in Ada 95

In some languages that involve the same kind of elaboration problems, e.g. Java and C++, the programmer is expected to worry about these ordering problems himself, and it is common to write a program in which an incorrect elaboration order gives surprising results, because it references variables before they are initialized. Ada 95 is designed to be a safe language, and a programmer-beware approach is clearly not sufficient. Consequently, the language provides three lines of defense:

Standard rules

Some standard rules restrict the possible choice of elaboration order. In particular, if you `with` a unit, then its spec is always elaborated before the unit doing the `with`. Similarly, a parent spec is always elaborated before the child spec, and finally a spec is always elaborated before its corresponding body.

Dynamic elaboration checks

Dynamic checks are made at run time, so that if some entity is accessed before it is elaborated (typically by means of a subprogram call) then the exception (`Program_Error`) is raised.

Elaboration control

Facilities are provided for the programmer to specify the desired order of elaboration.

Let's look at these facilities in more detail. First, the rules for dynamic checking. One possible rule would be simply to say that the exception is raised if you access a variable which has not yet been elaborated. The trouble with this approach is that it could require expensive checks on every variable reference. Instead Ada 95 has two rules which are a little more restrictive, but easier to check, and easier to state:

Restrictions on calls

A subprogram can only be called at elaboration time if its body has been elaborated. The rules for elaboration given above guarantee that the spec of the subprogram has been elaborated before the call, but not the body. If this rule is violated, then the exception `Program_Error` is raised.

Restrictions on instantiations

A generic unit can only be instantiated if the body of the generic unit has been elaborated. Again, the rules for elaboration given above guarantee that the spec of the generic unit has been elaborated before the instantiation, but not the body. If this rule is violated, then the exception `Program_Error` is raised.

The idea is that if the body has been elaborated, then any variables it references must have been elaborated; by checking for the body being elaborated we guarantee that none of its references causes any trouble. As we noted above, this is a little too restrictive, because a subprogram that has no non-local references in its body may in fact be safe to call. However, it really would be unsafe to rely on this, because it would mean that the caller was aware of details of the implementation in the body. This goes against the basic tenets of Ada.

A plausible implementation can be described as follows. A Boolean variable is associated with each subprogram and each generic unit. This variable is initialized to False, and is set to True at the point body is elaborated. Every call or instantiation checks the variable, and raises `Program_Error` if the variable is False.

Note that one might think that it would be good enough to have one Boolean variable for each package, but that would not deal with cases of trying to call a body in the same package as the call that has not been elaborated yet. Of course a compiler may be able to do enough analysis to optimize away some of the Boolean variables as unnecessary, and GNAT indeed does such optimizations, but still the easiest conceptual model is to think of there being one variable per subprogram.

11.3 Controlling the Elaboration Order in Ada 95

In the previous section we discussed the rules in Ada 95 which ensure that `Program_Error` is raised if an incorrect elaboration order is chosen. This prevents erroneous executions, but we need mechanisms to specify a correct execution and avoid the exception altogether. To achieve this, Ada 95 provides a number of features for controlling the order of elaboration. We discuss these features in this section.

First, there are several ways of indicating to the compiler that a given unit has no elaboration problems:

packages that do not require a body

In Ada 95, a library package that does not require a body does not permit a body. This means that if we have a such a package, as in:

```
package Definitions is
  generic
    type m is new integer;
  package Subp is
    type a is array (1 .. 10) of m;
    type b is array (1 .. 20) of m;
  end Subp;
end Definitions;
```

A package that `with's` `Definitions` may safely instantiate `Definitions.Subp` because the compiler can determine that there definitely is no package body to worry about in this case

pragma Pure

Places sufficient restrictions on a unit to guarantee that no call to any subprogram in the unit can result in an elaboration problem. This means that the

compiler does not need to worry about the point of elaboration of such units, and in particular, does not need to check any calls to any subprograms in this unit.

`pragma Preelaborate`

This pragma places slightly less stringent restrictions on a unit than does `pragma Pure`, but these restrictions are still sufficient to ensure that there are no elaboration problems with any calls to the unit.

`pragma Elaborate_Body`

This pragma requires that the body of a unit be elaborated immediately after its spec. Suppose a unit **A** has such a pragma, and unit **B** does a `with` of unit **A**. Recall that the standard rules require the spec of unit **A** to be elaborated before the `with`'ing unit; given the pragma in **A**, we also know that the body of **A** will be elaborated before **B**, so that calls to **A** are safe and do not need a check.

Note that, unlike `pragma Pure` and `pragma Preelaborate`, the use of `Elaborate_Body` does not guarantee that the program is free of elaboration problems, because it may not be possible to satisfy the requested elaboration order. Let's go back to the example with **Unit_1** and **Unit_2**. If a programmer marks **Unit_1** as `Elaborate_Body`, and not **Unit_2**, then the order of elaboration will be:

```
Spec of Unit_2
Spec of Unit_1
Body of Unit_1
Body of Unit_2
```

Now that means that the call to **Func_1** in **Unit_2** need not be checked, it must be safe. But the call to **Func_2** in **Unit_1** may still fail if **Expression_1** is equal to 1, and the programmer must still take responsibility for this not being the case.

If all units carry a `pragma Elaborate_Body`, then all problems are eliminated, except for calls entirely within a body, which are in any case fully under programmer control. However, using the pragma everywhere is not always possible. In particular, for our **Unit_1/Unit_2** example, if we marked both of them as having `pragma Elaborate_Body`, then clearly there would be no possible elaboration order.

The above pragmas allow a server to guarantee safe use by clients, and clearly this is the preferable approach. Consequently a good rule in Ada 95 is to mark units as `Pure` or `Preelaborate` if possible, and if this is not possible, mark them as `Elaborate_Body` if possible. As we have seen, there are situations where neither of these three pragmas can be used. So we also provide methods for clients to control the order of elaboration of the servers on which they depend:

`pragma Elaborate (unit)`

This pragma is placed in the context clause, after a `with` clause, and it requires that the body of the named unit be elaborated before the unit in which the pragma occurs. The idea is to use this pragma if the current unit calls at elaboration time, directly or indirectly, some subprogram in the named unit.

`pragma Elaborate_All (unit)`

This is a stronger version of the `Elaborate` pragma. Consider the following example:

```
Unit A with's unit B and calls B.Func in elab code
Unit B with's unit C, and B.Func calls C.Func
```

Now if we put a pragma `Elaborate (B)` in unit A, this ensures that the body of B is elaborated before the call, but not the body of C, so the call to `C.Func` could still cause `Program_Error` to be raised.

The effect of a pragma `Elaborate_All` is stronger, it requires not only that the body of the named unit be elaborated before the unit doing the `with`, but also the bodies of all units that the named unit uses, following `with` links transitively. For example, if we put a pragma `Elaborate_All (B)` in unit A, then it requires not only that the body of B be elaborated before A, but also the body of C, because B `with's` C.

We are now in a position to give a usage rule in Ada 95 for avoiding elaboration problems, at least if dynamic dispatching and access to subprogram values are not used. We will handle these cases separately later.

The rule is simple. If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a `with'ed` unit, or instantiate a generic unit in a `with'ed` unit, then if the `with'ed` unit does not have pragma `Pure` or `Preelaborate`, then the client should have a pragma `Elaborate_All` for the `with'ed` unit. By following this rule a client is assured that calls can be made without risk of an exception. If this rule is not followed, then a program may be in one of four states:

No order exists

No order of elaboration exists which follows the rules, taking into account any `Elaborate`, `Elaborate_All`, or `Elaborate_Body` pragmas. In this case, an Ada 95 compiler must diagnose the situation at bind time, and refuse to build an executable program.

One or more orders exist, all incorrect

One or more acceptable elaboration orders exists, and all of them generate an elaboration order problem. In this case, the binder can build an executable program, but `Program_Error` will be raised when the program is run.

Several orders exist, some right, some incorrect

One or more acceptable elaboration orders exists, and some of them work, and some do not. The programmer has not controlled the order of elaboration, so the binder may or may not pick one of the correct orders, and the program may or may not raise an exception when it is run. This is the worst case, because it means that the program may fail when moved to another compiler, or even another version of the same compiler.

One or more orders exists, all correct

One or more acceptable elaboration orders exist, and all of them work. In this case the program runs successfully. This state of affairs can be guaranteed by following the rule we gave above, but may be true even if the rule is not followed.

Note that one additional advantage of following our `Elaborate_All` rule is that the program continues to stay in the ideal (all orders OK) state even if maintenance changes some bodies of some subprograms. Conversely, if a program that does not follow this rule happens to be

safe at some point, this state of affairs may deteriorate silently as a result of maintenance changes.

You may have noticed that the above discussion did not mention the use of `Elaborate_Body`. This was a deliberate omission. If you with an `Elaborate_Body` unit, it still may be the case that code in the body makes calls to some other unit, so it is still necessary to use `Elaborate_All` on such units.

11.4 Controlling Elaboration in GNAT - Internal Calls

In the case of internal calls, i.e. calls within a single package, the programmer has full control over the order of elaboration, and it is up to the programmer to elaborate declarations in an appropriate order. For example writing:

```
function One return Float;

Q : Float := One;

function One return Float is
begin
    return 1.0;
end One;
```

will obviously raise `Program_Error` at run time, because function `One` will be called before its body is elaborated. In this case GNAT will generate a warning that the call will raise `Program_Error`:

```
1. procedure y is
2.     function One return Float;
3.
4.     Q : Float := One;
      |
      >>> warning: cannot call "One" before body is elaborated
      >>> warning: Program_Error will be raised at run time
5.
6.     function One return Float is
7.     begin
8.         return 1.0;
9.     end One;
10.
11. begin
12.     null;
13. end;
```

Note that in this particular case, it is likely that the call is safe, because the function `One` does not access any global variables. Nevertheless in Ada 95, we do not want the validity

of the check to depend on the contents of the body (think about the separate compilation case), so this is still wrong, as we discussed in the previous sections.

The error is easily corrected by rearranging the declarations so that the body of `One` appears before the declaration containing the call (note that in Ada 95, declarations can appear in any order, so there is no restriction that would prevent this reordering, and if we write:

```
function One return Float;  
  
function One return Float is  
begin  
    return 1.0;  
end One;  
  
Q : Float := One;
```

then all is well, no warning is generated, and no `Program_Error` exception will be raised. Things are more complicated when a chain of subprograms is executed:

```
function A return Integer;  
function B return Integer;  
function C return Integer;  
  
function B return Integer is begin return A; end;  
function C return Integer is begin return B; end;  
  
X : Integer := C;  
  
function A return Integer is begin return 1; end;
```

Now the call to `C` at elaboration time in the declaration of `X` is correct, because the body of `C` is already elaborated, and the call to `B` within the body of `C` is correct, but the call to `A` within the body of `B` is incorrect, because the body of `A` has not been elaborated, so `Program_Error` will be raised on the call to `A`. In this case GNAT will generate a warning that `Program_Error` may be raised at the point of the call. Let's look at the warning:

```

1. procedure x is
2.   function A return Integer;
3.   function B return Integer;
4.   function C return Integer;
5.
6.   function B return Integer is begin return A; end;
      |
      >>> warning: call to "A" before body is elaborated may
            raise Program_Error
      >>> warning: "B" called at line 7
      >>> warning: "C" called at line 9

7.   function C return Integer is begin return B; end;
8.
9.   X : Integer := C;
10.
11.  function A return Integer is begin return 1; end;
12.
13. begin
14.   null;
15. end;

```

Note that the message here says "may raise", instead of the direct case, where the message says "will be raised". That's because whether A is actually called depends in general on run-time flow of control. For example, if the body of B said

```

function B return Integer is
begin
  if some-condition-depending-on-input-data then
    return A;
  else
    return 1;
  end if;
end B;

```

then we could not know until run time whether the incorrect call to A would actually occur, so `Program_Error` might or might not be raised. It is possible for a compiler to do a better job of analyzing bodies, to determine whether or not `Program_Error` might be raised, but it certainly couldn't do a perfect job (that would require solving the halting problem and is provably impossible), and because this is a warning anyway, it does not seem worth the effort to do the analysis. Cases in which it would be relevant are rare.

In practice, warnings of either of the forms given above will usually correspond to real errors, and should be examined carefully and eliminated. In the rare case where a warning is bogus, it can be suppressed by any of the following methods:

- Compile with the `/WARNINGS=SUPPRESS` qualifier set
- Suppress `Elaboration_Checks` for the called subprogram
- Use pragma `Warnings_Off` to turn warnings off for the call

For the internal elaboration check case, GNAT by default generates the necessary run-time checks to ensure that `Program_Error` is raised if any call fails an elaboration check. Of course this can only happen if a warning has been issued as described above. The use of pragma `Suppress (Elaboration_Checks)` may (but is not guaranteed to) suppress some of these checks, meaning that it may be possible (but is not guaranteed) for a program to be able to call a subprogram whose body is not yet elaborated, without raising a `Program_Error` exception.

11.5 Controlling Elaboration in GNAT - External Calls

The previous section discussed the case in which the execution of a particular thread of elaboration code occurred entirely within a single unit. This is the easy case to handle, because a programmer has direct and total control over the order of elaboration, and furthermore, checks need only be generated in cases which are rare and which the compiler can easily detect. The situation is more complex when separate compilation is taken into account. Consider the following:

```
package Math is
  function Sqrt (Arg : Float) return Float;
end Math;

package body Math is
  function Sqrt (Arg : Float) return Float is
  begin
    ...
  end Sqrt;
end Math;
with Math;
package Stuff is
  X : Float := Math.Sqrt (0.5);
end Stuff;

with Stuff;
procedure Main is
begin
  ...
end Main;
```

where `Main` is the main program. When this program is executed, the elaboration code must first be executed, and one of the jobs of the binder is to determine the order in which the units of a program are to be elaborated. In this case we have four units: the spec and body of `Math`, the spec of `Stuff` and the body of `Main`). In what order should the four separate sections of elaboration code be executed?

There are some restrictions in the order of elaboration that the binder can choose. In particular, if unit `U` has a `with` for a package `X`, then you are assured that the spec of `X` is elaborated before `U`, but you are not assured that the body of `X` is elaborated before `U`. This means that in the above case, the binder is allowed to choose the order:

```
spec of Math
spec of Stuff
```



```

body of Math
body of Main

```

but that's not good, because now the call to `Math.Sqrt` that happens during the elaboration of the `Stuff` spec happens before the body of `Math.Sqrt` is elaborated, and hence causes `Program_Error` exception to be raised. At first glance, one might say that the binder is misbehaving, because obviously you want to elaborate the body of something you `with` first, but that is not a general rule that can be followed in all cases. Consider

```

package X is ...

package Y is ...

with X;
package body Y is ...

with Y;
package body X is ...

```

This is a common arrangement, and, apart from the order of elaboration problems that might arise in connection with elaboration code, this works fine. A rule that says that you must first elaborate the body of anything you `with` cannot work in this case: the body of `X` `with's` `Y`, which means you would have to elaborate the body of `Y` first, but that `with's` `X`, which means you have to elaborate the body of `X` first, but ... and we have a loop that cannot be broken.

It is true that the binder can in many cases guess an order of elaboration that is unlikely to cause a `Program_Error` exception to be raised, and it tries to do so (in the above example of `Math/Stuff/Spec`, the GNAT binder will by default elaborate the body of `Math` right after its spec, so all will be well).

However, a program that blindly relies on the binder to be helpful can get into trouble, as we discussed in the previous sections, so GNAT provides a number of facilities for assisting the programmer in developing programs that are robust with respect to elaboration order.

11.6 Default Behavior in GNAT - Ensuring Safety

The default behavior in GNAT ensures elaboration safety. In its default mode GNAT implements the rule we previously described as the right approach. Let's restate it:

- *If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a `with`'ed unit, or instantiate a generic unit in a `with`'ed unit, then if the `with`'ed unit does not have pragma `Pure` or `Preelaborate`, then the client should have an `Elaborate_All` for the `with`'ed unit.*

By following this rule a client is assured that calls and instantiations can be made without risk of an exception.

In this mode GNAT traces all calls that are potentially made from elaboration code, and puts in any missing implicit `Elaborate_All` pragmas. The advantage of this approach is that no elaboration problems are possible if the binder can find an elaboration order that is

consistent with these implicit `Elaborate_All` pragmas. The disadvantage of this approach is that no such order may exist.

If the binder does not generate any diagnostics, then it means that it has found an elaboration order that is guaranteed to be safe. However, the binder may still be relying on implicitly generated `Elaborate_All` pragmas so portability to other compilers than GNAT is not guaranteed.

If it is important to guarantee portability, then the compilations should use the `'/WARNINGS=ELABORATION'` (warn on elaboration problems) qualifier. This will cause warning messages to be generated indicating the missing `Elaborate_All` pragmas. Consider the following source program:

```
with k;
package j is
  m : integer := k.r;
end;
```

where it is clear that there should be a pragma `Elaborate_All` for unit `k`. An implicit pragma will be generated, and it is likely that the binder will be able to honor it. However, it is safer to include the pragma explicitly in the source. If this unit is compiled with the `'/WARNINGS=ELABORATION'` qualifier, then the compiler outputs a warning:

```
1. with k;
2. package j is
3.   m : integer := k.r;
   |
   >>> warning: call to "r" may raise Program_Error
   >>> warning: missing pragma Elaborate_All for "k"
4. end;
```

and these warnings can be used as a guide for supplying manually the missing pragmas.

This default mode is more restrictive than the Ada Reference Manual, and it is possible to construct programs which will compile using the dynamic model described there, but will run into a circularity using the safer static model we have described.

Of course any Ada compiler must be able to operate in a mode consistent with the requirements of the Ada Reference Manual, and in particular must have the capability of implementing the standard dynamic model of elaboration with run-time checks.

In GNAT, this standard mode can be achieved either by the use of the `'/CHECKS=ELABORATION'` qualifier on the compiler (`GNAT_COMPILE` or `GNAT_MAKE`) command, or by the use of the configuration pragma:

```
pragma Elaboration_Checks (RM);
```

Either approach will cause the unit affected to be compiled using the standard dynamic run-time elaboration checks described in the Ada Reference Manual. The static model is

generally preferable, since it is clearly safer to rely on compile and link time checks rather than run-time checks. However, in the case of legacy code, it may be difficult to meet the requirements of the static model. This issue is further discussed in Section 11.9 [What to Do If the Default Elaboration Behavior Fails], page 159.

Note that the static model provides a strict subset of the allowed behavior and programs of the Ada Reference Manual, so if you do adhere to the static model and no circularities exist, then you are assured that your program will work using the dynamic model.

11.7 Elaboration Issues for Library Tasks

In this section we examine special elaboration issues that arise for programs that declare library level tasks.

Generally the model of execution of an Ada program is that all units are elaborated, and then execution of the program starts. However, the declaration of library tasks definitely does not fit this model. The reason for this is that library tasks start as soon as they are declared (more precisely, as soon as the statement part of the enclosing package body is reached), that is to say before elaboration of the program is complete. This means that if such a task calls a subprogram, or an entry in another task, the callee may or may not be elaborated yet, and in the standard Reference Manual model of dynamic elaboration checks, you can even get timing dependent `Program_Error` exceptions, since there can be a race between the elaboration code and the task code.

The static model of elaboration in GNAT seeks to avoid all such dynamic behavior, by being conservative, and the conservative approach in this particular case is to assume that all the code in a task body is potentially executed at elaboration time if a task is declared at the library level.

This can definitely result in unexpected circularities. Consider the following example

```
package Decls is
  task Lib_Task is
    entry Start;
  end Lib_Task;

  type My_Int is new Integer;

  function Ident (M : My_Int) return My_Int;
end Decls;

with Utils;
package body Decls is
  task body Lib_Task is
  begin
    accept Start;
    Utils.Put_Val (2);
  end Lib_Task;

  function Ident (M : My_Int) return My_Int is
  begin
    return M;
  end Ident;
end Decls;
```

```

with Decls;
package Utils is
  procedure Put_Val (Arg : Decls.My_Int);
end Utils;

with Text_IO;
package body Utils is
  procedure Put_Val (Arg : Decls.My_Int) is
  begin
    Text_IO.Put_Line (Decls.My_Int'Image (Decls.Ident (Arg)));
  end Put_Val;
end Utils;

with Decls;
procedure Main is
begin
  Decls.Lib_Task.Start;
end;

```

If the above example is compiled in the default static elaboration mode, then a circularity occurs. The circularity comes from the call `Utils.Put_Val` in the task body of `Decls.Lib_Task`. Since this call occurs in elaboration code, we need an implicit pragma `Elaborate_All` for `Utils`. This means that not only must the spec and body of `Utils` be elaborated before the body of `Decls`, but also the spec and body of any unit that is `with`'ed by the body of `Utils` must also be elaborated before the body of `Decls`. This is the transitive implication of pragma `Elaborate_All` and it makes sense, because in general the body of `Put_Val` might have a call to something in a `with`'ed unit.

In this case, the body of `Utils` (actually its spec) `with`'s `Decls`. Unfortunately this means that the body of `Decls` must be elaborated before itself, in case there is a call from the body of `Utils`.

Here is the exact chain of events we are worrying about:

1. In the body of `Decls` a call is made from within the body of a library task to a subprogram in the package `Utils`. Since this call may occur at elaboration time (given that the task is activated at elaboration time), we have to assume the worst, i.e. that the call does happen at elaboration time.
2. This means that the body and spec of `Util` must be elaborated before the body of `Decls` so that this call does not cause an access before elaboration.
3. Within the body of `Util`, specifically within the body of `Util.Put_Val` there may be calls to any unit `with`'ed by this package.
4. One such `with`'ed package is package `Decls`, so there might be a call to a subprogram in `Decls` in `Put_Val`. In fact there is such a call in this example, but we would have to assume that there was such a call even if it were not there, since we are not supposed to write the body of `Decls` knowing what is in the body of `Utils`; certainly in the case of the static elaboration model, the compiler does not know what is in other bodies and must assume the worst.
5. This means that the spec and body of `Decls` must also be elaborated before we elaborate the unit containing the call, but that unit is `Decls`! This means that the body of `Decls` must be elaborated before itself, and that's a circularity.

Indeed, if you add an explicit pragma `Elaborate_All` for `Utils` in the body of `Decls` you will get a true Ada Reference Manual circularity that makes the program illegal.

In practice, we have found that problems with the static model of elaboration in existing code often arise from library tasks, so we must address this particular situation.

Note that if we compile and run the program above, using the dynamic model of elaboration (that is to say use the `/CHECKS=ELABORATION` qualifier), then it compiles, binds, links, and runs, printing the expected result of 2. Therefore in some sense the circularity here is only apparent, and we need to capture the properties of this program that distinguish it from other library-level tasks that have real elaboration problems.

We have four possible answers to this question:

- Use the dynamic model of elaboration.

If we use the `/CHECKS=ELABORATION` qualifier, then as noted above, the program works. Why is this? If we examine the task body, it is apparent that the task cannot proceed past the `accept` statement until after elaboration has been completed, because the corresponding entry call comes from the main program, not earlier. This is why the dynamic model works here. But that's really giving up on a precise analysis, and we prefer to take this approach only if we cannot solve the problem in any other manner. So let us examine two ways to reorganize the program to avoid the potential elaboration problem.

- Split library tasks into separate packages.

Write separate packages, so that library tasks are isolated from other declarations as much as possible. Let us look at a variation on the above program.

```

package Decls1 is
  task Lib_Task is
    entry Start;
  end Lib_Task;
end Decls1;

with Utils;
package body Decls1 is
  task body Lib_Task is
  begin
    accept Start;
    Utils.Put_Val (2);
  end Lib_Task;
end Decls1;

package Decls2 is
  type My_Int is new Integer;
  function Ident (M : My_Int) return My_Int;
end Decls2;

with Utils;
package body Decls2 is
  function Ident (M : My_Int) return My_Int is
  begin
    return M;
  end Ident;
end Decls2;

with Decls2;
```

```

package Utils is
  procedure Put_Val (Arg : Decls2.My_Int);
end Utils;

with Text_IO;
package body Utils is
  procedure Put_Val (Arg : Decls2.My_Int) is
  begin
    Text_IO.Put_Line (Decls2.My_Int'Image (Decls2.Ident (Arg)));
  end Put_Val;
end Utils;

with Decls1;
procedure Main is
begin
  Decls1.Lib_Task.Start;
end;

```

All we have done is to split `Decls` into two packages, one containing the library task, and one containing everything else. Now there is no cycle, and the program compiles, binds, links and executes using the default static model of elaboration.

- Declare separate task types.

A significant part of the problem arises because of the use of the single task declaration form. This means that the elaboration of the task type, and the elaboration of the task itself (i.e. the creation of the task) happen at the same time. A good rule of style in Ada 95 is to always create explicit task types. By following the additional step of placing task objects in separate packages from the task type declaration, many elaboration problems are avoided. Here is another modified example of the example program:

```

package Decls is
  task type Lib_Task_Type is
    entry Start;
  end Lib_Task_Type;

  type My_Int is new Integer;

  function Ident (M : My_Int) return My_Int;
end Decls;

with Utils;
package body Decls is
  task body Lib_Task_Type is
  begin
    accept Start;
    Utils.Put_Val (2);
  end Lib_Task_Type;

  function Ident (M : My_Int) return My_Int is
  begin
    return M;
  end Ident;
end Decls;

with Decls;
package Utils is
  procedure Put_Val (Arg : Decls.My_Int);

```

```

end Utils;

with Text_IO;
package body Utils is
  procedure Put_Val (Arg : Decls.My_Int) is
  begin
    Text_IO.Put_Line (Decls.My_Int'Image (Decls.Ident (Arg)));
  end Put_Val;
end Utils;

with Decls;
package Declst is
  Lib_Task : Decls.Lib_Task_Type;
end Declst;

with Declst;
procedure Main is
begin
  Declst.Lib_Task.Start;
end;
```

What we have done here is to replace the **task** declaration in package **Decls** with a **task type** declaration. Then we introduce a separate package **Declst** to contain the actual task object. This separates the elaboration issues for the **task type** declaration, which causes no trouble, from the elaboration issues of the task object, which is also unproblematic, since it is now independent of the elaboration of **Utils**. This separation of concerns also corresponds to a generally sound engineering principle of separating declarations from instances. This version of the program also compiles, binds, links, and executes, generating the expected output.

- Use **No_Entry_Calls_In_Elaboration_Code** restriction.

The previous two approaches described how a program can be restructured to avoid the special problems caused by library task bodies. In practice, however, such restructuring may be difficult to apply to existing legacy code, so we must consider solutions that do not require massive rewriting.

Let us consider more carefully why our original sample program works under the dynamic model of elaboration. The reason is that the code in the task body blocks immediately on the **accept** statement. Now of course there is nothing to prohibit elaboration code from making entry calls (for example from another library level task), so we cannot tell in isolation that the task will not execute the **accept** statement during elaboration.

However, in practice it is very unusual to see elaboration code make any entry calls, and the pattern of tasks starting at elaboration time and then immediately blocking on **accept** or **select** statements is very common. What this means is that the compiler is being too pessimistic when it analyzes the whole package body as though it might be executed at elaboration time.

If we know that the elaboration code contains no entry calls, (a very safe assumption most of the time, that could almost be made the default behavior), then we can compile all units of the program under control of the following configuration pragma:

```
pragma Restrictions (No_Entry_Calls_In_Elaboration_Code);
```

This pragma can be placed in the 'GNAT.ADC' file in the usual manner. If we take our original unmodified program and compile it in the presence of a 'GNAT.ADC' containing the above pragma, then once again, we can compile, bind, link, and execute, obtaining the expected result. In the presence of this pragma, the compiler does not trace calls in a task body, that appear after the first **accept** or **select** statement, and therefore does not report a potential circularity in the original program.

The compiler will check to the extent it can that the above restriction is not violated, but it is not always possible to do a complete check at compile time, so it is important to use this pragma only if the stated restriction is in fact met, that is to say no task receives an entry call before elaboration of all units is completed.

11.8 Mixing Elaboration Models

So far, we have assumed that the entire program is either compiled using the dynamic model or static model, ensuring consistency. It is possible to mix the two models, but rules have to be followed if this mixing is done to ensure that elaboration checks are not omitted.

The basic rule is that *a unit compiled with the static model cannot be with'ed by a unit compiled with the dynamic model*. The reason for this is that in the static model, a unit assumes that its clients guarantee to use (the equivalent of) pragma **Elaborate_All** so that no elaboration checks are required in inner subprograms, and this assumption is violated if the client is compiled with dynamic checks.

The precise rule is as follows. A unit that is compiled with dynamic checks can only **with** a unit that meets at least one of the following criteria:

- The **with'ed** unit is itself compiled with dynamic elaboration checks (that is with the '/CHECKS=ELABORATION' qualifier).
- The **with'ed** unit is an internal GNAT implementation unit from the System, Interfaces, Ada, or GNAT hierarchies.
- The **with'ed** unit has pragma **Preelaborate** or pragma **Pure**.
- The **with'ing** unit (that is the client) has an explicit pragma **Elaborate_All** for the **with'ed** unit.

If this rule is violated, that is if a unit with dynamic elaboration checks **with's** a unit that does not meet one of the above four criteria, then the binder (GNAT BIND) will issue a warning similar to that in the following example:

```
warning: "X.ADS" has dynamic elaboration checks and with's
warning:  "Y.ADS" which has static elaboration checks
```

These warnings indicate that the rule has been violated, and that as a result elaboration checks may be missed in the resulting executable file. This warning may be suppressed using the **-ws** binder qualifier in the usual manner.

One useful application of this mixing rule is in the case of a subsystem which does not itself **with** units from the remainder of the application. In this case, the entire subsystem can be compiled with dynamic checks to resolve a circularity in the subsystem, while allowing the main application that uses this subsystem to be compiled using the more reliable default static model.

11.9 What to Do If the Default Elaboration Behavior Fails

If the binder cannot find an acceptable order, it outputs detailed diagnostics. For example:

```
error: elaboration circularity detected
info:  "proc (body)" must be elaborated before "pack (body)"
info:    reason: Elaborate_All probably needed in unit "pack (body)"
info:    recompile "pack (body)" with /WARNINGS=ELABORATION
info:                                     for full details
info:    "proc (body)"
info:      is needed by its spec:
info:    "proc (spec)"
info:      which is withed by:
info:    "pack (body)"
info:  "pack (body)" must be elaborated before "proc (body)"
info:    reason: pragma Elaborate in unit "proc (body)"
```

In this case we have a cycle that the binder cannot break. On the one hand, there is an explicit pragma `Elaborate` in `proc` for `pack`. This means that the body of `pack` must be elaborated before the body of `proc`. On the other hand, there is elaboration code in `pack` that calls a subprogram in `proc`. This means that for maximum safety, there should really be a pragma `Elaborate_All` in `pack` for `proc` which would require that the body of `proc` be elaborated before the body of `pack`. Clearly both requirements cannot be satisfied. Faced with a circularity of this kind, you have three different options.

Fix the program

The most desirable option from the point of view of long-term maintenance is to rearrange the program so that the elaboration problems are avoided. One useful technique is to place the elaboration code into separate child packages. Another is to move some of the initialization code to explicitly called subprograms, where the program controls the order of initialization explicitly. Although this is the most desirable option, it may be impractical and involve too much modification, especially in the case of complex legacy code.

Perform dynamic checks

If the compilations are done using the `/CHECKS=ELABORATION` (dynamic elaboration check) qualifier, then GNAT behaves in a quite different manner. Dynamic checks are generated for all calls that could possibly result in raising an exception. With this qualifier, the compiler does not generate implicit `Elaborate_All` pragmas. The behavior then is exactly as specified in the Ada 95 Reference Manual. The binder will generate an executable program that may or may not raise `Program_Error`, and then it is the programmer's job to ensure that it does not raise an exception. Note that it is important to compile all units with the qualifier, it cannot be used selectively.

Suppress checks

The drawback of dynamic checks is that they generate a significant overhead at run time, both in space and time. If you are absolutely sure that your program cannot raise any elaboration exceptions, and you still want to use the dynamic elaboration model, then you can use the configuration pragma

`Suppress (Elaboration_Checks)` to suppress all such checks. For example this pragma could be placed in the 'GNAT.ADC' file.

Suppress checks selectively

When you know that certain calls in elaboration code cannot possibly lead to an elaboration error, and the binder nevertheless generates warnings on those calls and inserts `Elaborate_All` pragmas that lead to elaboration circularities, it is possible to remove those warnings locally and obtain a program that will bind. Clearly this can be unsafe, and it is the responsibility of the programmer to make sure that the resulting program has no elaboration anomalies. The pragma `Suppress (Elaboration_Check)` can be used with different granularity to suppress warnings and break elaboration circularities:

- Place the pragma that names the called subprogram in the declarative part that contains the call.
- Place the pragma in the declarative part, without naming an entity. This disables warnings on all calls in the corresponding declarative region.
- Place the pragma in the package spec that declares the called subprogram, and name the subprogram. This disables warnings on all elaboration calls to that subprogram.
- Place the pragma in the package spec that declares the called subprogram, without naming any entity. This disables warnings on all elaboration calls to all subprograms declared in this spec.

These four cases are listed in order of decreasing safety, and therefore require increasing programmer care in their application. Consider the following program:

```

package Pack1 is
  function F1 return Integer;
  X1 : Integer;
end Pack1;

package Pack2 is
  function F2 return Integer;
  function Pure (x : integer) return integer;
  -- pragma Suppress (Elaboration_Check, On => Pure); -- (3)
  -- pragma Suppress (Elaboration_Check);             -- (4)
end Pack2;

with Pack2;
package body Pack1 is
  function F1 return Integer is
  begin
    return 100;
  end F1;
  Val : integer := Pack2.Pure (11);    -- Elab. call (1)
begin
  declare
    -- pragma Suppress(Elaboration_Check, Pack2.F2); -- (1)
    -- pragma Suppress(Elaboration_Check);           -- (2)
  begin
    X1 := Pack2.F2 + 1;                  -- Elab. call (2)
  end;
end;
```

```

end Pack1;

with Pack1;
package body Pack2 is
  function F2 return Integer is
  begin
    return Pack1.F1;
  end F2;
  function Pure (x : integer) return integer is
  begin
    return x ** 3 - 3 * x;
  end;
end Pack2;

with Pack1, Ada.Text_IO;
procedure Proc3 is
begin
  Ada.Text_IO.Put_Line(Pack1.X1'Img); -- 101
end Proc3;

```

In the absence of any pragmas, an attempt to bind this program produces the following diagnostics:

```

error: elaboration circularity detected
info:   "pack1 (body)" must be elaborated before "pack1 (body)"
info:     reason: Elaborate_All probably needed in unit "pack1 (body)"
info:     recompile "pack1 (body)" with /WARNINGS=ELABORATION for full details
info:       "pack1 (body)"
info:         must be elaborated along with its spec:
info:       "pack1 (spec)"
info:         which is withed by:
info:       "pack2 (body)"
info:         which must be elaborated along with its spec:
info:       "pack2 (spec)"
info:         which is withed by:
info:       "pack1 (body)"

```

The sources of the circularity are the two calls to `Pack2.Pure` and `Pack2.F2` in the body of `Pack1`. We can see that the call to `F2` is safe, even though `F2` calls `F1`, because the call appears after the elaboration of the body of `F1`. Therefore the pragma (1) is safe, and will remove the warning on the call. It is also possible to use pragma (2) because there are no other potentially unsafe calls in the block.

The call to `Pure` is safe because this function does not depend on the state of `Pack2`. Therefore any call to this function is safe, and it is correct to place pragma (3) in the corresponding package spec.

Finally, we could place pragma (4) in the spec of `Pack2` to disable warnings on all calls to functions declared therein. Note that this is not necessarily safe, and requires more detailed examination of the subprogram bodies involved. In particular, a call to `F2` requires that `F1` be already elaborated.

It is hard to generalize on which of these four approaches should be taken. Obviously if it is possible to fix the program so that the default treatment works, this is preferable, but this may not always be practical. It is certainly simple enough to use `/CHECKS=ELABORATION` but the danger in this case is that, even if the GNAT binder finds a correct elaboration

order, it may not always do so, and certainly a binder from another Ada compiler might not. A combination of testing and analysis (for which the warnings generated with the `/WARNINGS=ELABORATION` qualifier can be useful) must be used to ensure that the program is free of errors. One qualifier that is useful in this testing is the `/PESSIMISTIC_ELABORATION_ORDER` qualifier for GNAT BIND. Normally the binder tries to find an order that has the best chance of avoiding elaboration problems. With this qualifier, the binder plays a devil's advocate role, and tries to choose the order that has the best chance of failing. If your program works even with this qualifier, then it has a better chance of being error free, but this is still not a guarantee.

For an example of this approach in action, consider the C-tests (executable tests) from the ACVC suite. If these are compiled and run with the default treatment, then all but one of them succeed without generating any error diagnostics from the binder. However, there is one test that fails, and this is not surprising, because the whole point of this test is to ensure that the compiler can handle cases where it is impossible to determine a correct order statically, and it checks that an exception is indeed raised at run time.

This one test must be compiled and run using the `/CHECKS=ELABORATION` qualifier, and then it passes. Alternatively, the entire suite can be run using this qualifier. It is never wrong to run with the dynamic elaboration qualifier if your code is correct, and we assume that the C-tests are indeed correct (it is less efficient, but efficiency is not a factor in running the ACVC tests.)

11.10 Elaboration for Access-to-Subprogram Values

The introduction of access-to-subprogram types in Ada 95 complicates the handling of elaboration. The trouble is that it becomes impossible to tell at compile time which procedure is being called. This means that it is not possible for the binder to analyze the elaboration requirements in this case.

If at the point at which the access value is created (i.e., the evaluation of `P'Access` for a subprogram `P`), the body of the subprogram is known to have been elaborated, then the access value is safe, and its use does not require a check. This may be achieved by appropriate arrangement of the order of declarations if the subprogram is in the current unit, or, if the subprogram is in another unit, by using pragma `Pure`, `Preelaborate`, or `Elaborate_Body` on the referenced unit.

If the referenced body is not known to have been elaborated at the point the access value is created, then any use of the access value must do a dynamic check, and this dynamic check will fail and raise a `Program_Error` exception if the body has not been elaborated yet. GNAT will generate the necessary checks, and in addition, if the `/WARNINGS=ELABORATION` qualifier is set, will generate warnings that such checks are required.

The use of dynamic dispatching for tagged types similarly generates a requirement for dynamic checks, and premature calls to any primitive operation of a tagged type before the body of the operation has been elaborated, will result in the raising of `Program_Error`.

11.11 Summary of Procedures for Elaboration Control

First, compile your program with the default options, using none of the special elaboration control qualifiers. If the binder successfully binds your program, then you can be confident that, apart from issues raised by the use of access-to-subprogram types and dynamic dispatching, the program is free of elaboration errors. If it is important that the program be portable, then use the `'/WARNINGS=ELABORATION'` qualifier to generate warnings about missing `Elaborate_All` pragmas, and supply the missing pragmas.

If the program fails to bind using the default static elaboration handling, then you can fix the program to eliminate the binder message, or recompile the entire program with the `'/CHECKS=ELABORATION'` qualifier to generate dynamic elaboration checks, and, if you are sure there really are no elaboration problems, use a global pragma `Suppress (Elaboration_Checks)`.

11.12 Other Elaboration Order Considerations

This section has been entirely concerned with the issue of finding a valid elaboration order, as defined by the Ada Reference Manual. In a case where several elaboration orders are valid, the task is to find one of the possible valid elaboration orders (and the static model in GNAT will ensure that this is achieved).

The purpose of the elaboration rules in the Ada Reference Manual is to make sure that no entity is accessed before it has been elaborated. For a subprogram, this means that the spec and body must have been elaborated before the subprogram is called. For an object, this means that the object must have been elaborated before its value is read or written. A violation of either of these two requirements is an access before elaboration order, and this section has been all about avoiding such errors.

In the case where more than one order of elaboration is possible, in the sense that access before elaboration errors are avoided, then any one of the orders is "correct" in the sense that it meets the requirements of the Ada Reference Manual, and no such error occurs.

However, it may be the case for a given program, that there are constraints on the order of elaboration that come not from consideration of avoiding elaboration errors, but rather from extra-lingual logic requirements. Consider this example:

```
with Init_Constants;
package Constants is
  X : Integer := 0;
  Y : Integer := 0;
end Constants;

package Init_Constants is
  procedure Calc;
end Init_Constants;

with Constants;
package body Init_Constants is
  procedure Calc is begin null; end;
begin
  Constants.X := 3;
  Constants.Y := 4;
```

```

end Init_Constants;

with Constants;
package Calc is
  Z : Integer := Constants.X + Constants.Y;
end Calc;

with Calc;
with Text_IO; use Text_IO;
procedure Main is
begin
  Put_Line (Calc.Z'Img);
end Main;

```

In this example, there is more than one valid order of elaboration. For example both the following are correct orders:

```

Init_Constants spec
Constants spec
Calc spec
Main body
Init_Constants body

and

Init_Constants spec
Init_Constants body
Constants spec
Calc spec
Main body

```

There is no language rule to prefer one or the other, both are correct from an order of elaboration point of view. But the programmatic effects of the two orders are very different. In the first, the elaboration routine of **Calc** initializes **Z** to zero, and then the main program runs with this value of zero. But in the second order, the elaboration routine of **Calc** runs after the body of **Init_Constants** has set **X** and **Y** and thus **Z** is set to 7 before **Main** runs.

One could perhaps by applying pretty clever non-artificial intelligence to the situation guess that it is more likely that the second order of elaboration is the one desired, but there is no formal linguistic reason to prefer one over the other. In fact in this particular case, GNAT will prefer the second order, because of the rule that bodies are elaborated as soon as possible, but it's just luck that this is what was wanted (if indeed the second order was preferred).

If the program cares about the order of elaboration routines in a case like this, it is important to specify the order required. In this particular case, that could have been achieved by adding to the spec of **Calc**:

```
pragma Elaborate_All (Constants);
```

which requires that the body (if any) and spec of **Constants**, as well as the body and spec of any unit with'ed by **Constants** be elaborated before **Calc** is elaborated.

Clearly no automatic method can always guess which alternative you require, and if you are working with legacy code that had constraints of this kind which were not properly specified by adding **Elaborate** or **Elaborate_All** pragmas, then indeed it is possible that two different compilers can choose different orders.

The GNAT `BIND /PESSIMISTIC_ELABORATION` qualifier may be useful in smoking out problems. This qualifier causes bodies to be elaborated as late as possible instead of as early as possible. In the example above, it would have forced the choice of the first elaboration order. If you get different results when using this qualifier, and particularly if one set of results is right, and one is wrong as far as you are concerned, it shows that you have some missing `Elaborate` pragmas. For the example above, we have the following output:

```
GNAT MAKE -f -q main
main
7
GNAT MAKE -f -q main /BINDER_QUALIFIERS -p
main
0
```

It is of course quite unlikely that both these results are correct, so it is up to you in a case like this to investigate the source of the difference, by looking at the two elaboration orders that are chosen, and figuring out which is correct, and then adding the necessary `Elaborate_All` pragmas to ensure the desired order.

12 The Cross-Referencing Tools GNAT XREF and GNAT FIND

The compiler generates cross-referencing information (unless you set the `/XREF=SUPPRESS` qualifier), which are saved in the `.ALI` files. This information indicates where in the source each entity is declared and referenced. Note that entities in package Standard are not included, but entities in all other predefined units are included in the output.

Before using any of these two tools, you need to compile successfully your application, so that GNAT gets a chance to generate the cross-referencing information.

The two tools GNAT XREF and GNAT FIND take advantage of this information to provide the user with the capability to easily locate the declaration and references to an entity. These tools are quite similar, the difference being that GNAT FIND is intended for locating definitions and/or references to a specified entity or entities, whereas GNAT XREF is oriented to generating a full report of all cross-references.

To use these tools, you must not compile your application using the `/XREF=SUPPRESS` qualifier on the `GNAT MAKE` command line (See Info file `gnat_ug`, node `The GNAT Make Program GNAT MAKE`). Otherwise, cross-referencing information will not be generated.

12.1 GNAT XREF Qualifiers

The command lines for GNAT XREF is:

```
$ GNAT XREF [qualifiers] sourcefile1 [sourcefile2 ...]
```

where

`sourcefile1, sourcefile2`

identifies the source files for which a report is to be generated. The `'with'`ed units will be processed too. You must provide at least one file.

These file names are considered to be regular expressions, so for instance specifying `'source*.ADB'` is the same as giving every file in the current directory whose name starts with `'source'` and whose extension is `'adb'`.

The qualifiers can be :

`/ALL_FILES`

If this qualifier is present, GNAT FIND and GNAT XREF will parse the read-only files found in the library search path. Otherwise, these files will be ignored. This option can be used to protect Gnat sources or your own libraries from being parsed, thus making GNAT FIND and GNAT XREF much faster, and their output much smaller.

`/SOURCE_SEARCH=dir`

When looking for source files also look in directory DIR. The order in which source file search is undertaken is the same as for `'GNAT MAKE'`.

`/OBJECT_SEARCH=dir`

When searching for library and object files, look in directory DIR. The order in which library files are searched is the same as for `'GNAT MAKE'`.

`/NOSTD_INCLUDES`

Do not look for sources in the system default directory.

/NOSTD_LIBRARIES

Do not look for library files in the system default directory.

/RUNTIME_SYSTEM=*rts-path*

Specifies the default location of the runtime library. Same meaning as the equivalent **GNAT MAKE** flag (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).

-d

If this qualifier is set **GNAT XREF** will output the parent type reference for each matching derived types.

/FULL_PATHNAME

If this qualifier is set, the output file names will be preceded by their directory (if the file was found in the search path). If this qualifier is not set, the directory will not be printed.

/IGNORE_LOCALS

If this qualifier is set, information is output only for library-level entities, ignoring local entities. The use of this qualifier may accelerate **GNAT FIND** and **GNAT XREF**.

/SEARCH=direc

Equivalent to '**/OBJECT_SEARCH=direc /SOURCE_SEARCH=direc**'.

/PROJECT=file

Specify a project file to use See Section 10.1.1 [Project Files], page 109. By default, **GNAT XREF** and **GNAT FIND** will try to locate a project file in the current directory.

If a project file is either specified or found by the tools, then the content of the source directory and object directory lines are added as if they had been specified respectively by '**/SOURCE_SEARCH**' and '**/OBJECT_SEARCH**'.

/UNUSED

Output only unused symbols. This may be really useful if you give your main compilation unit on the command line, as **GNAT XREF** will then display every unused entity and 'with'ed package.

All these qualifiers may be in any order on the command line, and may even appear after the file names. They need not be separated by spaces, thus you can say '**GNAT XREF /ALL_FILES/IGNORE_LOCALS**' instead of '**GNAT XREF /ALL_FILES /IGNORE_LOCALS**'.

12.2 GNAT FIND Qualifiers

The command line for **GNAT FIND** is:

```
$ GNAT FIND [qualifiers] pattern[:sourcefile[:line[:column]]]
           [file1 file2 ...]
```

where

pattern An entity will be output only if it matches the regular expression found in '**pattern**', see See Section 12.4 [Regular Expressions in GNAT FIND and GNAT XREF], page 172.

Omitting the pattern is equivalent to specifying '*', which will match any entity. Note that if you do not provide a pattern, you have to provide both a sourcefile and a line.

Entity names are given in Latin-1, with uppercase/lowercase equivalence for matching purposes. At the current time there is no support for 8-bit codes other than Latin-1, or for wide characters in identifiers.

sourcefile

GNAT FIND will look for references, bodies or declarations of symbols referenced in 'sourcefile', at line 'line' and column 'column'. See Section 12.6 [Examples of GNAT FIND Usage], page 174 for syntax examples.

line is a decimal integer identifying the line number containing the reference to the entity (or entities) to be located.

column is a decimal integer identifying the exact location on the line of the first character of the identifier for the entity reference. Columns are numbered from 1.

file1 file2 ...

The search will be restricted to these files. If none are given, then the search will be done for every library file in the search path. These file must appear only after the pattern or sourcefile.

These file names are considered to be regular expressions, so for instance specifying 'source*.ADB' is the same as giving every file in the current directory whose name starts with 'source' and whose extension is 'adb'.

Not that if you specify at least one file in this part, GNAT FIND may sometimes not be able to find the body of the subprograms...

At least one of 'sourcefile' or 'pattern' has to be present on the command line.

The following qualifiers are available:

/ALL_FILES

If this qualifier is present, GNAT FIND and GNAT XREF will parse the read-only files found in the library search path. Otherwise, these files will be ignored. This option can be used to protect Gnat sources or your own libraries from being parsed, thus making GNAT FIND and GNAT XREF much faster, and their output much smaller.

/SOURCE_SEARCH=dir

When looking for source files also look in directory DIR. The order in which source file search is undertaken is the same as for 'GNAT MAKE'.

/OBJECT_SEARCH=dir

When searching for library and object files, look in directory DIR. The order in which library files are searched is the same as for 'GNAT MAKE'.

/NOSTD_INCLUDES

Do not look for sources in the system default directory.

/NOSTD_LIBRARIES

Do not look for library files in the system default directory.

/RUNTIME_SYSTEM=runtime-path

Specifies the default location of the runtime library. Same meaning as the equivalent GNAT MAKE flag (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).

-d If this qualifier is set, then **GNAT FIND** will output the parent type reference for each matching derived types.

/EXPRESSIONS

By default, **GNAT FIND** accept the simple regular expression set for **'pattern'**. If this qualifier is set, then the pattern will be considered as full Unix-style regular expression.

/FULL_PATHNAME

If this qualifier is set, the output file names will be preceded by their directory (if the file was found in the search path). If this qualifier is not set, the directory will not be printed.

/IGNORE_LOCALS

If this qualifier is set, information is output only for library-level entities, ignoring local entities. The use of this qualifier may accelerate **GNAT FIND** and **GNAT XREF**.

/SEARCH=direc

Equivalent to **'/OBJECT_SEARCH=direc /SOURCE_SEARCH=direc'**.

/PROJECT=file

Specify a project file (see Section 10.1.1 [Project Files], page 109) to use. By default, **GNAT XREF** and **GNAT FIND** will try to locate a project file in the current directory.

If a project file is either specified or found by the tools, then the content of the source directory and object directory lines are added as if they had been specified respectively by **'/SOURCE_SEARCH'** and **'/OBJECT_SEARCH'**.

/REFERENCES

By default, **GNAT FIND** will output only the information about the declaration, body or type completion of the entities. If this qualifier is set, the **GNAT FIND** will locate every reference to the entities in the files specified on the command line (or in every file in the search path if no file is given on the command line).

/PRINT_LINES

If this qualifier is set, then **GNAT FIND** will output the content of the Ada source file lines were the entity was found.

-t If this qualifier is set, then **GNAT FIND** will output the type hierarchy for the specified type. It act like **-d** option but recursively from parent type to parent type. When this qualifier is set it is not possible to specify more than one file.

All these qualifiers may be in any order on the command line, and may even appear after the file names. They need not be separated by spaces, thus you can say **'GNAT XREF /ALL_FILES/IGNORE_LOCALS'** instead of **'GNAT XREF /ALL_FILES /IGNORE_LOCALS'**.

As stated previously, **GNAT FIND** will search in every directory in the search path. You can force it to look only in the current directory if you specify ***** at the end of the command line.

12.3 Project Files for GNAT XREF and GNAT FIND

Project files allow a programmer to specify how to compile its application, where to find sources,... These files are used primarily by the Glide Ada mode, but they can also be used by the two tools GNAT XREF and GNAT FIND.

A project file name must end with `‘.adp’`. If a single one is present in the current directory, then GNAT XREF and GNAT FIND will extract the information from it. If multiple project files are found, none of them is read, and you have to use the `‘-p’` qualifier to specify the one you want to use.

The following lines can be included, even though most of them have default values which can be used in most cases. The lines can be entered in any order in the file. Except for `‘src_dir’` and `‘obj_dir’`, you can only have one instance of each line. If you have multiple instances, only the last one is taken into account.

`src_dir=DIR [default: "[]"]`

specifies a directory where to look for source files. Multiple `src_dir` lines can be specified and they will be searched in the order they are specified.

`obj_dir=DIR [default: "[]"]`

specifies a directory where to look for object and library files. Multiple `obj_dir` lines can be specified and they will be searched in the order they are specified

`comp_opt=SWITCHES [default: ""]`

creates a variable which can be referred to subsequently by using the `‘${comp_opt}’` notation. This is intended to store the default qualifiers given to `‘GNAT MAKE’` and `‘GNAT COMPILE’`.

`bind_opt=SWITCHES [default: ""]`

creates a variable which can be referred to subsequently by using the `‘${bind_opt}’` notation. This is intended to store the default qualifiers given to `‘GNAT BIND’`.

`link_opt=SWITCHES [default: ""]`

creates a variable which can be referred to subsequently by using the `‘${link_opt}’` notation. This is intended to store the default qualifiers given to `‘GNAT LINK’`.

`main=EXECUTABLE [default: ""]`

specifies the name of the executable for the application. This variable can be referred to in the following lines by using the `‘${main}’` notation.

`comp_cmd=COMMAND [default: "GNAT COMPILE /SEARCH=${src_dir} /DEBUG /TRY_SEMANTICS"]`

specifies the command used to compile a single file in the application.

`make_cmd=COMMAND [default: "GNAT MAKE ${main} /SOURCE_SEARCH=${src_dir} /OBJECT_SEARCH=${obj_dir} /DEBUG /TRY_SEMANTICS /COMPILER_QUALIFIERS ${comp_opt} /BINDER_QUALIFIERS ${bind_opt} /LINKER_QUALIFIERS ${link_opt}"]`

specifies the command used to recompile the whole application.

`run_cmd=COMMAND [default: "${main}"]`

specifies the command used to run the application.

`debug_cmd=COMMAND [default: "GDB ${main}"]`
 specifies the command used to debug the application

GNAT XREF and GNAT FIND only take into account the 'src_dir' and 'obj_dir' lines, and ignore the others.

12.4 Regular Expressions in GNAT FIND and GNAT XREF

As specified in the section about GNAT FIND, the pattern can be a regular expression. Actually, there are two set of regular expressions which are recognized by the program :

globbing patterns

These are the most usual regular expression. They are the same that you generally used in a Unix shell command line, or in a DOS session.
 Here is a more formal grammar :

```

regexp ::= term
term    ::= elmt                -- matches elmt
term    ::= elmt elmt           -- concatenation (elmt then elmt)
term    ::= *                   -- any string of 0 or more characters
term    ::= ?                   -- matches any character
term    ::= [char {char}]       -- matches any character listed
term    ::= [char - char]       -- matches any character in range

```

full regular expression

The second set of regular expressions is much more powerful. This is the type of regular expressions recognized by utilities such a 'grep'.

The following is the form of a regular expression, expressed in Ada reference manual style BNF is as follows

```

regexp ::= term { | term } -- alternation (term or term ...)

term ::= item { item }      -- concatenation (item then item)

item ::= elmt              -- match elmt
item ::= elmt *            -- zero or more elmt's
item ::= elmt +            -- one or more elmt's
item ::= elmt ?            -- matches elmt or nothing
elmt ::= nschar            -- matches given character
elmt ::= [nschar {nschar}] -- matches any character listed
elmt ::= [^ nschar {nschar}] -- matches any character not listed
elmt ::= [char - char]     -- matches chars in given range
elmt ::= \ char            -- matches given character
elmt ::= .                -- matches any single character
elmt ::= ( regexp )        -- parens used for grouping

char ::= any character, including special characters
nschar ::= any character except () [].*+?^

```

Following are a few examples :

'abcde|fghi'

will match any of the two strings 'abcde' and 'fghi'.

- ‘abc*d’ will match any string like ‘abd’, ‘abcd’, ‘abccd’, ‘abcccd’, and so on
- ‘[a-z]+’ will match any string which has only lowercase characters in it (and at least one character)

12.5 Examples of GNAT XREF Usage

12.5.1 General Usage

For the following examples, we will consider the following units :

```

MAIN.ADS:
1: with Bar;
2: package Main is
3:   procedure Foo (B : in Integer);
4:   C : Integer;
5: private
6:   D : Integer;
7: end Main;

MAIN.ADB:
1: package body Main is
2:   procedure Foo (B : in Integer) is
3:   begin
4:     C := B;
5:     D := B;
6:     Bar.Print (B);
7:     Bar.Print (C);
8:   end Foo;
9: end Main;

BAR.ADS:
1: package Bar is
2:   procedure Print (B : Integer);
3: end bar;

```

The first thing to do is to recompile your application (for instance, in that case just by doing a ‘GNAT MAKE main’, so that GNAT generates the cross-referencing information. You can then issue any of the following commands:

GNAT XREF MAIN.ADB

GNAT XREF generates cross-reference information for MAIN.ADB and every unit ‘with’ed by MAIN.ADB.

The output would be:

B		Type: Integer
Decl: BAR.ADS	2:22	
B		Type: Integer

Decl:	MAIN.ADS	3:20				
Body:	MAIN.ADB	2:20				
Ref:	MAIN.ADB	4:13	5:13	6:19		
Bar					Type: Unit	
Decl:	BAR.ADS	1:9				
Ref:	MAIN.ADB	6:8	7:8			
	MAIN.ADS	1:6				
C					Type: Integer	
Decl:	MAIN.ADS	4:5				
Modi:	MAIN.ADB	4:8				
Ref:	MAIN.ADB	7:19				
D					Type: Integer	
Decl:	MAIN.ADS	6:5				
Modi:	MAIN.ADB	5:8				
Foo					Type: Unit	
Decl:	MAIN.ADS	3:15				
Body:	MAIN.ADB	2:15				
Main					Type: Unit	
Decl:	MAIN.ADS	2:9				
Body:	MAIN.ADB	1:14				
Print					Type: Unit	
Decl:	BAR.ADS	2:15				
Ref:	MAIN.ADB	6:12	7:12			

that is the entity `Main` is declared in `MAIN.ADS`, line 2, column 9, its body is in `MAIN.ADB`, line 1, column 14 and is not referenced any where.

The entity `Print` is declared in `BAR.ADS`, line 2, column 15 and it is referenced in `MAIN.ADB`, line 6 column 12 and line 7 column 12.

GNAT XREF PACKAGE1.ADB PACKAGE2.ADS

GNAT XREF will generate cross-reference information for `PACKAGE1.ADB`, `PACKAGE2.ADS` and any other package 'with'ed by any of these.

12.6 Examples of GNAT FIND Usage

GNAT FIND /FULL_PATHNAME xyz:MAIN.ADB

Find declarations for all entities `xyz` referenced at least once in `MAIN.ADB`. The references are searched in every library file in the search path.

The directories will be printed as well (as the `'/FULL_PATHNAME'` qualifier is set)

The output will look like:

```
[directory]MAIN.ADS:106:14: xyz <= declaration
[directory]MAIN.ADB:24:10: xyz <= body
[directory]FOO.ADS:45:23: xyz <= declaration
```

that is to say, one of the entities `xyz` found in `MAIN.ADB` is declared at line 12 of `MAIN.ADS` (and its body is in `MAIN.ADB`), and another one is declared at line 45 of `FOO.ADS`

GNAT FIND /FULL_PATHNAME/SOURCE_LINE xyz:MAIN.ADB

This is the same command as the previous one, instead GNAT FIND will display the content of the Ada source file lines.

The output will look like:

```
[directory]MAIN.ADS:106:14: xyz <= declaration
```



```
    procedure xyz;  
[directory]MAIN.ADB:24:10: xyz <= body  
    procedure xyz is  
[directory]FOO.ADS:45:23: xyz <= declaration  
    xyz : Integer;
```

This can make it easier to find exactly the location you are looking for.

GNAT FIND /REFERENCES "*x*":MAIN.ADS:123 FOO.ADB

Find references to all entities containing an x that are referenced on line 123 of MAIN.ADS. The references will be searched only in MAIN.ADB and FOO.ADB.

GNAT FIND MAIN.ADS:123

Find declarations and bodies for all entities that are referenced on line 123 of MAIN.ADS.

This is the same as **GNAT FIND "*" :MAIN.ADB:123**.

GNAT FIND [mydir]MAIN.ADB:123:45

Find the declaration for the entity referenced at column 45 in line 123 of file MAIN.ADB in directory mydir. Note that it is usual to omit the identifier name when the column is given, since the column position identifies a unique reference.

The column has to be the beginning of the identifier, and should not point to any character in the middle of the identifier.

13 File Name Krunching Using GNAT KRUNCH

This chapter discusses the method used by the compiler to shorten the default file names chosen for Ada units so that they do not exceed the maximum length permitted. It also describes the GNAT KRUNCH utility that can be used to determine the result of applying this shortening.

13.1 About GNAT KRUNCH

The default file naming rule in GNAT is that the file name must be derived from the unit name. The exact default rule is as follows:

- Take the unit name and replace all dots by hyphens.
- If such a replacement occurs in the second character position of a name, and the first character is A, G, S, or I then replace the dot by the character \$ (dollar sign) instead of a minus.

The reason for this exception is to avoid clashes with the standard names for children of System, Ada, Interfaces, and GNAT, which use the prefixes S- A- I- and G- respectively.

The `/FILE_NAME_MAX_LENGTH=nn` qualifier of the compiler activates a "krunching" circuit that limits file names to *nn* characters (where *nn* is a decimal integer). For example, using OpenVMS, where the maximum file name length is 39, the value of *nn* is usually set to 39, but if you want to generate a set of files that would be usable if ported to a system with some different maximum file length, then a different value can be specified. The default value of 39 for OpenVMS need not be specified.

The GNAT KRUNCH utility can be used to determine the krunched name for a given file, when krunched to a specified maximum length.

13.2 Using GNAT KRUNCH

The GNAT KRUNCH command has the form

```
$ GNAT KRUNCH name /COUNT=nn
```

name can be an Ada name with dots or the GNAT name of the unit, where the dots representing child units or subunit are replaced by hyphens. The only confusion arises if a name ends in `.ADS` or `.ADB`. GNAT KRUNCH takes this to be an extension if there are no other dots in the name.

length represents the length of the krunched name. The default when no argument is given is 39 characters. A length of zero stands for unlimited, in other words do not chop except for system files which are always 39.

The output is the krunched name. The output has an extension only if the original argument was a file name with an extension.

13.3 Krunching Method

The initial file name is determined by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit and replacing the separating dots with hyphens and using uppercase for all letters, except that a hyphen in the second character position is replaced by a dollar sign if the first character is A, I, G, or S. The extension is .ADS for a specification and .ADB for a body. Krunching does not affect the extension, but the file name is shortened to the specified length by following these rules:

- The name is divided into segments separated by hyphens, tildes or underscores and all hyphens, tildes, and underscores are eliminated. If this leaves the name short enough, we are done.
- If the name is too long, the longest segment is located (left-most if there are two of equal length), and shortened by dropping its last character. This is repeated until the name is short enough.

As an example, consider the krunching of

'OUR-STRINGS-WIDE_FIXED.ADB' to fit the name into 8 characters as required by some operating systems.

```
our-strings-wide_fixed 22
our strings wide fixed 19
our string  wide fixed 18
our strin   wide fixed 17
our stri    wide fixed 16
our stri    wide fixe  15
our str     wide fixe  14
our str     wid  fixe  13
our str     wid  fix   12
ou str      wid  fix   11
ou st       wid  fix   10
ou st       wi   fix   9
ou st       wi   fi    8
Final file name: OUSTWIFI.ADB
```

- The file names for all predefined units are always krunched to eight characters. The krunching of these predefined units uses the following special prefix replacements:

'ada-' replaced by 'A-'

'gnat-' replaced by 'G-'

'interfaces-'
replaced by 'I-'

'system-' replaced by 'S-'

These system files have a hyphen in the second character position. That is why normal user files replace such a character with a dollar sign, to avoid confusion with system file names.

As an example of this special rule, consider

'ADA-STRINGS-WIDE_FIXED.ADB', which gets krunched as follows:

```
ada-strings-wide_fixed 22
a- strings wide fixed 18
a- string  wide fixed 17
```

```

a- strin  wide fixed 16
a- stri   wide fixed 15
a- stri   wide fixe 14
a- str    wide fixe 13
a- str    wid  fixe 12
a- str    wid  fix  11
a- st     wid  fix  10
a- st     wi   fix   9
a- st     wi   fi    8
Final file name: A-STWIFI.ADB

```

Of course no file shortening algorithm can guarantee uniqueness over all possible unit names, and if file name krunching is used then it is your responsibility to ensure that no name clashes occur. The utility program GNAT KRUNCH is supplied for conveniently determining the krunched name of a file.

13.4 Examples of GNAT KRUNCH Usage

```

$ GNAT KRUNCH VERY_LONG_UNIT_NAME.ADS/count=6 --> VLUNNA.ADS
$ GNAT KRUNCH VERY_LONG_UNIT_NAME.ADS/count=0 --> VERY_LONG_UNIT_NAME.ADS

```


14 Preprocessing Using GNAT PREPROCESS

The GNAT PREPROCESS utility provides a simple preprocessing capability for Ada programs. It is designed for use with GNAT, but is not dependent on any special features of GNAT.

14.1 Using GNAT PREPROCESS

To call GNAT PREPROCESS use

```
$ GNAT PREPROCESS [-bcrsu] [-Dsymbol=value] infile outfile [deffile]
```

where

- infile** is the full name of the input file, which is an Ada source file containing preprocessor directives.
- outfile** is the full name of the output file, which is an Ada source in standard Ada form. When used with GNAT, this file name will normally have an ads or adb suffix.
- deffile** is the full name of a text file containing definitions of symbols to be referenced by the preprocessor. This argument is optional, and can be replaced by the use of the `-D` qualifier.

qualifiers

is an optional sequence of qualifiers as described in the next section.

14.2 Qualifiers for GNAT PREPROCESS

/BLANK_LINES

Causes both preprocessor lines and the lines deleted by preprocessing to be replaced by blank lines in the output source file, preserving line numbers in the output file.

/COMMENTS

Causes both preprocessor lines and the lines deleted by preprocessing to be retained in the output source as comments marked with the special string `"-!"`. This option will result in line numbers being preserved in the output file.

-Dsymbol=value

Defines a new symbol, associated with value. If no value is given on the command line, then symbol is considered to be `True`. This qualifier can be used in place of a definition file.

/REMOVE (default)

This is the default setting which causes lines deleted by preprocessing to be entirely removed from the output file.

/REFERENCE

Causes a `Source_Reference` pragma to be generated that references the original input file, so that error messages will use the file name of this original file. The use of this qualifier implies that preprocessor lines are not to be removed from

the file, so its use will force `/BLANK_LINES` mode if `/COMMENTS` has not been specified explicitly.

Note that if the file to be preprocessed contains multiple units, then it will be necessary to `GNAT CHOP` the output file from `GNAT PREPROCESS`. If a `Source_Reference` pragma is present in the preprocessed file, it will be respected by `GNAT CHOP /REFERENCE` so that the final chopped files will correctly refer to the original input source file for `GNAT PREPROCESS`.

/SYMBOLS Causes a sorted list of symbol names and values to be listed on the standard output file.

/UNDEFINED

Causes undefined symbols to be treated as having the value `FALSE` in the context of a preprocessor test. In the absence of this option, an undefined symbol in a `#if` or `#elsif` test will be treated as an error.

14.3 Form of Definitions File

The definitions file contains lines of the form

```
symbol := value
```

where `symbol` is an identifier, following normal Ada (case-insensitive) rules for its syntax, and `value` is one of the following:

- Empty, corresponding to a null substitution
- A string literal using normal Ada syntax
- Any sequence of characters from the set (letters, digits, period, underline).

Comment lines may also appear in the definitions file, starting with the usual `--`, and comments may be added to the definitions lines.

14.4 Form of Input Text for GNAT PREPROCESS

The input text may contain preprocessor conditional inclusion lines, as well as general symbol substitution sequences.

The preprocessor conditional inclusion commands have the form

```
#if expression [then]
  lines
#elsif expression [then]
  lines
#elsif expression [then]
  lines
...
#else
  lines
#end if;
```

In this example, *expression* is defined by the following grammar:


```

expression ::= <symbol>
expression ::= <symbol> = "<value>"
expression ::= <symbol> = <symbol>
expression ::= <symbol> 'Defined
expression ::= not expression
expression ::= expression and expression
expression ::= expression or expression
expression ::= expression and then expression
expression ::= expression or else expression
expression ::= ( expression )

```

For the first test (*expression* ::= <symbol>) the symbol must have either the value true or false, that is to say the right-hand of the symbol definition must be one of the (case-insensitive) literals **True** or **False**. If the value is true, then the corresponding lines are included, and if the value is false, they are excluded.

The test (*expression* ::= <symbol> 'Defined) is true only if the symbol has been defined in the definition file or by a **-D** qualifier on the command line. Otherwise, the test is false.

The equality tests are case insensitive, as are all the preprocessor lines.

If the symbol referenced is not defined in the symbol definitions file, then the effect depends on whether or not qualifier **-u** is specified. If so, then the symbol is treated as if it had the value false and the test fails. If this qualifier is not specified, then it is an error to reference an undefined symbol. It is also an error to reference a symbol that is defined with a value other than **True** or **False**.

The use of the **not** operator inverts the sense of this logical test, so that the lines are included only if the symbol is not defined. The **then** keyword is optional as shown

The **#** must be the first non-blank character on a line, but otherwise the format is free form. Spaces or tabs may appear between the **#** and the keyword. The keywords and the symbols are case insensitive as in normal Ada code. Comments may be used on a preprocessor line, but other than that, no other tokens may appear on a preprocessor line. Any number of **elsif** clauses can be present, including none at all. The **else** is optional, as in Ada.

The **#** marking the start of a preprocessor line must be the first non-blank character on the line, i.e. it must be preceded only by spaces or horizontal tabs.

Symbol substitution outside of preprocessor lines is obtained by using the sequence

```
$symbol
```

anywhere within a source line, except in a comment or within a string literal. The identifier following the **\$** must match one of the symbols defined in the symbol definition file, and the result is to substitute the value of the symbol in place of **\$symbol** in the output file.

Note that although the substitution of strings within a string literal is not possible, it is possible to have a symbol whose defined value is a string literal. So instead of setting XYZ to **hello** and writing:

```
Header : String := "$XYZ";
```

you should set XYZ to **"hello"** and write:

```
Header : String := $XYZ;
```

and then the substitution will occur as desired.

15 The GNAT Run-Time Library Builder GNAT LIBRARY

GNAT LIBRARY is a tool for rebuilding the GNAT run time with user supplied configuration pragmas.

15.1 Running GNAT LIBRARY

The GNAT LIBRARY command has the form

```
$ GNAT LIBRARY /[CREATE | SET | DELETE]=directory [/CONFIG=file]
```

15.2 Qualifiers for GNAT LIBRARY

GNAT LIBRARY recognizes the following qualifiers:

/CREATE=directory

Create the new run-time library in the specified directory.

/SET=directory

Make the library in the specified directory the current run-time library.

/DELETE=directory

Delete the run-time library in the specified directory.

/CONFIG=file

With **/CREATE**: Use the configuration pragmas in the specified file when building the library.

With **/SET**: Use the configuration pragmas in the specified file when compiling.

15.3 Example of GNAT LIBRARY Usage

Contents of VAXFLOAT.ADC:

```
pragma Float_Representation (VAX_Float);
```

```
$ GNAT LIBRARY /CREATE=[.VAXFLOAT] /CONFIG=VAXFLOAT.ADC
```

GNAT LIBRARY rebuilds the run-time library in directory `[.VAXFLOAT]`

16 The GNAT Library Browser GNAT LIST

GNAT LIST is a tool that outputs information about compiled units. It gives the relationship between objects, unit names and source files. It can also be used to check the source dependencies of a unit as well as various characteristics.

16.1 Running GNAT LIST

The GNAT LIST command has the form

```
$ GNAT LIST qualifiers object_or_alias_file
```

The main argument is the list of object or ‘ali’ files (see Section 2.8 [The Ada Library Information Files], page 19) for which information is requested.

In normal mode, without additional option, GNAT LIST produces a four-column listing. Each line represents information for a specific object. The first column gives the full path of the object, the second column gives the name of the principal unit in this object, the third column gives the status of the source and the fourth column gives the full path of the source representing this unit. Here is a simple example of use:

```
$ GNAT LIST *.OBJ
[] DEMO1.OBJ          demo1          DIF DEMO1.ADB
[] DEMO2.OBJ          demo2          OK DEMO2.ADB
[] HELLO.OBJ          h1            OK HELLO.ADB
[] INSTR-CHILD.OBJ    instr.child    MOK INSTR-CHILD.ADB
[] INSTR.OBJ          instr         OK INSTR.ADB
[] TEF.OBJ            tef           DIF TEF.ADB
[] TEXT_IO_EXAMPLE.OBJ text_io_example OK TEXT_IO_EXAMPLE.ADB
[] TGEF.OBJ           tgef          DIF TGEF.ADB
```

The first line can be interpreted as follows: the main unit which is contained in object file ‘DEMO1.OBJ’ is demo1, whose main source is in ‘DEMO1.ADB’. Furthermore, the version of the source used for the compilation of demo1 has been modified (DIF). Each source file has a status qualifier which can be:

OK (unchanged)

The version of the source file used for the compilation of the specified unit corresponds exactly to the actual source file.

MOK (slightly modified)

The version of the source file used for the compilation of the specified unit differs from the actual source file but not enough to require recompilation. If you use GNAT MAKE with the qualifier /MINIMAL_RECOMPILATION, a file marked MOK will not be recompiled.

DIF (modified)

No version of the source found on the path corresponds to the source used to build this object.

??? (file not found)

No source file was found for this unit.

HID (hidden, unchanged version not first on PATH)

The version of the source that corresponds exactly to the source used for compilation has been found on the path but it is hidden by another version of the same source that has been modified.

16.2 Qualifiers for GNAT LIST

GNAT LIST recognizes the following qualifiers:

/ALL_UNITS

Consider all units, including those of the predefined Ada library. Especially useful with /DEPENDENCIES.

/DEPENDENCIES

List sources from which specified units depend on.

/OUTPUT=OPTIONS

Output the list of options.

/OUTPUT=OBJECTS

Only output information about object files.

/OUTPUT=SOURCES

Only output information about source files.

/OUTPUT=UNITS

Only output information about compilation units.

/OBJECT_SEARCH=*dir***/SOURCE_SEARCH=*dir*****/SEARCH=*dir*****/NOCURRENT_DIRECTORY****/NOSTD_INCLUDES**

Source path manipulation. Same meaning as the equivalent GNAT MAKE flags (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).

/RUNTIME_SYSTEM=*rts-path*

Specifies the default location of the runtime library. Same meaning as the equivalent GNAT MAKE flag (see Section 6.2 [Qualifiers for GNAT MAKE], page 92).

/OUTPUT=VERBOSE

Verbose mode. Output the complete source and object paths. Do not use the default column layout but instead use long format giving as much as information possible on each requested units, including special characteristics such as:

Preelaborable

The unit is preelaborable in the Ada 95 sense.

No_Elab_Code

No elaboration code has been produced by the compiler for this unit.

Pure

The unit is pure in the Ada 95 sense.

Elaborate_Body

The unit contains a pragma `Elaborate_Body`.

Remote_Types

The unit contains a pragma `Remote_Types`.

Shared_Passive

The unit contains a pragma `Shared_Passive`.

Predefined

This unit is part of the predefined environment and cannot be modified by the user.

Remote_Call_Interface

The unit contains a pragma `Remote_Call_Interface`.

16.3 Example of GNAT LIST Usage

```
GNAT LIST /DEPENDENCIES /OUTPUT=SOURCES /ALL_UNITS DEMO1.ADB
```

```
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] ADA.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] A-FINALI.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] A-FILICO.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] A-STREAM.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] A-TAGS.ADS
DEMO1.ADB
GEN_LIST.ADS
GEN_LIST.ADB
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] GNAT.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] G-IO.ADS
INSTR.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] SYSTEM.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-EXCTAB.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-FINIMP.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-FINROO.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-SECSTA.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-STALIB.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-STOELE.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-STRATT.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-TASOLI.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] S-UNSTYP.ADS
GNU: [LIB.OPENVMS7_1.2_8_1.ADALIB] UNCHCONV.ADS
```


17 Finding Memory Problems with GNAT Debug Pool

The use of unchecked deallocation and unchecked conversion can easily lead to incorrect memory references. The problems generated by such references are usually difficult to tackle because the symptoms can be very remote from the origin of the problem. In such cases, it is very helpful to detect the problem as early as possible. This is the purpose of the Storage Pool provided by `GNAT.Debug_Pools`.

In order to use the GNAT specific debugging pool, the user must associate a debug pool object with each of the access types that may be related to suspected memory problems. See Ada Reference Manual 13.11.

```
type Ptr is access Some_Type;
Pool : GNAT.Debug_Pools.Debug_Pool;
for Ptr'Storage_Pool use Pool;
```

`GNAT.Debug_Pools` is derived from of a GNAT-specific kind of pool: the `Checked_Pool`. Such pools, like standard Ada storage pools, allow the user to redefine allocation and deallocation strategies. They also provide a checkpoint for each dereference, through the use of the primitive operation `Dereference` which is implicitly called at each dereference of an access value.

Once an access type has been associated with a debug pool, operations on values of the type may raise four distinct exceptions, which correspond to four potential kinds of memory corruption:

- `GNAT.Debug_Pools.Accessing_Not_Allocated_Storage`
- `GNAT.Debug_Pools.Accessing_Deallocated_Storage`
- `GNAT.Debug_Pools.Freeing_Not_Allocated_Storage`
- `GNAT.Debug_Pools.Freeing_Deallocated_Storage`

For types associated with a `Debug_Pool`, dynamic allocation is performed using the standard GNAT allocation routine. References to all allocated chunks of memory are kept in an internal dictionary. The deallocation strategy consists in not releasing the memory to the underlying system but rather to fill it with a memory pattern easily recognizable during debugging sessions: The memory pattern is the old IBM hexadecimal convention: `16#DEADBEEF#`. Upon each dereference, a check is made that the access value denotes a properly allocated memory location. Here is a complete example of use of `Debug_Pools`, that includes typical instances of memory corruption:

```
with Gnat.Io; use Gnat.Io;
with Unchecked_Deallocation;
with Unchecked_Conversion;
with GNAT.Debug_Pools;
with System.Storage_Elements;
with Ada.Exceptions; use Ada.Exceptions;
procedure Debug_Pool_Test is

  type T is access Integer;
  type U is access all T;

  P : GNAT.Debug_Pools.Debug_Pool;
  for T'Storage_Pool use P;

  procedure Free is new Unchecked_Deallocation (Integer, T);
  function UC is new Unchecked_Conversion (U, T);
```

```

A, B : aliased T;

procedure Info is new GNAT.Debug_Pools.Print_Info(Put_Line);

begin
  Info (P);
  A := new Integer;
  B := new Integer;
  B := A;
  Info (P);
  Free (A);
  begin
    Put_Line (Integer'Image(B.all));
  exception
    when E : others => Put_Line ("raised: " & Exception_Name (E));
  end;
  begin
    Free (B);
  exception
    when E : others => Put_Line ("raised: " & Exception_Name (E));
  end;
  B := UC(A'Access);
  begin
    Put_Line (Integer'Image(B.all));
  exception
    when E : others => Put_Line ("raised: " & Exception_Name (E));
  end;
  begin
    Free (B);
  exception
    when E : others => Put_Line ("raised: " & Exception_Name (E));
  end;
  Info (P);
end Debug_Pool_Test;

```

The debug pool mechanism provides the following precise diagnostics on the execution of this erroneous program:

```

Debug Pool info:
Total allocated bytes : 0
Total deallocated bytes : 0
Current Water Mark: 0
High Water Mark: 0

Debug Pool info:
Total allocated bytes : 8
Total deallocated bytes : 0
Current Water Mark: 8
High Water Mark: 8

raised: GNAT.DEBUG_POOLS.ACCESSING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_DEALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.ACCESSING_NOT_ALLOCATED_STORAGE
raised: GNAT.DEBUG_POOLS.FREEING_NOT_ALLOCATED_STORAGE
Debug Pool info:
Total allocated bytes : 8
Total deallocated bytes : 4
Current Water Mark: 4
High Water Mark: 8

```

18 Creating Sample Bodies Using GNAT STUB

GNAT STUB creates body stubs, that is, empty but compilable bodies for library unit declarations.

To create a body stub, GNAT STUB has to compile the library unit declaration. Therefore, bodies can be created only for legal library units. Moreover, if a library unit depends semantically upon units located outside the current directory, you have to provide the source search path when calling GNAT STUB, see the description of GNAT STUB qualifiers below.

18.1 Running GNAT STUB

GNAT STUB has the command-line interface of the form

```
$ GNAT STUB [qualifiers] filename [directory]
```

where

filename is the name of the source file that contains a library unit declaration for which a body must be created. This name should follow the GNAT file name conventions. No crunching is allowed for this file name. The file name may contain the path information.

directory

indicates the directory to place a body stub (default is the current directory)

qualifiers

is an optional sequence of qualifiers as described in the next section

18.2 Qualifiers for GNAT STUB

/FULL If the destination directory already contains a file with a name of the body file for the argument spec file, replace it with the generated body stub.

/HEADER=SPEC

Put the comment header (i.e. all the comments preceding the compilation unit) from the source of the library unit declaration into the body stub.

/HEADER=GENERAL

Put a sample comment header into the body stub.

/SEARCH=dir

/NOCURRENT_DIRECTORY

These qualifiers have the same meaning as in calls to GNAT COMPILE. They define the source search path in the call to GNAT COMPILE issued by GNAT STUB to compile an argument source file.

/INDENTATION=n

(*n* is a decimal natural number). Set the indentation level in the generated body sample to *n*, '/INDENTATION=0' means "no indentation", the default indentation is 3.

/TREE_FILE=SAVE

Do not remove the tree file (i.e. the snapshot of the compiler internal structures used by GNAT STUB) after creating the body stub.

/LINE_LENGTH=*n*

(*n* is a decimal positive number) Set the maximum line length in the body stub to *n*, the default is 78.

/QUIET

Quiet mode: do not generate a confirmation when a body is successfully created or a message when a body is not required for an argument unit.

/TREE_FILE=REUSE

Reuse the tree file (if it exists) instead of creating it: instead of creating the tree file for the library unit declaration, GNAT STUB tries to find it in the current directory and use it for creating a body. If the tree file is not found, no body is created. /REUSE also implies /SAVE, whether or not /SAVE is set explicitly.

/TREE_FILE=OVERWRITE

Overwrite the existing tree file: if the current directory already contains the file which, according to the GNAT file name rules should be considered as a tree file for the argument source file, GNAT STUB will refuse to create the tree file needed to create a body sampler, unless **-t** option is set

/VERBOSE Verbose mode: generate version information.

19 Reducing the Size of Ada Executables with GNAT ELIM

19.1 About GNAT ELIM

When a program shares a set of Ada packages with other programs, it may happen that this program uses only a fraction of the subprograms defined in these packages. The code created for these unused subprograms increases the size of the executable.

GNAT ELIM tracks unused subprograms in an Ada program and outputs a list of GNAT-specific **Eliminate** pragmas (see next section) marking all the subprograms that are declared but never called. By placing the list of **Eliminate** pragmas in the GNAT configuration file ‘GNAT.ADC’ and recompiling your program, you may decrease the size of its executable, because the compiler will not generate the code for ‘eliminated’ subprograms.

GNAT ELIM needs as its input data a set of tree files (see Section 19.3 [Tree Files], page 195) representing all the components of a program to process and a bind file for a main subprogram (see Section 19.4 [Preparing Tree and Bind Files for GNAT ELIM], page 196).

19.2 Eliminate Pragma

The simplified syntax of the Eliminate pragma used by GNAT ELIM is:

```
pragma Eliminate (Library_Unit_Name, Subprogram_Name);
```

where

Library_Unit_Name

full expanded Ada name of a library unit

Subprogram_Name

a simple or expanded name of a subprogram declared within this compilation unit

The effect of an **Eliminate** pragma placed in the GNAT configuration file ‘GNAT.ADC’ is:

- If the subprogram **Subprogram_Name** is declared within the library unit **Library_Unit_Name**, the compiler will not generate code for this subprogram. This applies to all overloaded subprograms denoted by **Subprogram_Name**.
- If a subprogram marked by the pragma **Eliminate** is used (called) in a program, the compiler will produce an error message in the place where it is called.

19.3 Tree Files

A tree file stores a snapshot of the compiler internal data structures at the very end of a successful compilation. It contains all the syntactic and semantic information for the compiled unit and all the units upon which it depends semantically. To use tools that make use of tree files, you need to first produce the right set of tree files.

GNAT produces correct tree files when `/TREE_OUTPUT /NOLOAD` options are set in a GNAT COMPILE call. The tree files have an `.adt` extension. Therefore, to produce a tree file for the compilation unit contained in a file named `'FOO.ADB'`, you must use the command

```
$ GNAT COMPILE /NOLOAD /TREE_OUTPUT FOO.ADB
```

and you will get the tree file `'foo.adt'`. compilation.

19.4 Preparing Tree and Bind Files for GNAT ELIM

A set of tree files covering the program to be analyzed with GNAT ELIM and the bind file for the main subprogram does not have to be in the current directory. `'-T'` GNAT ELIM option may be used to provide the search path for tree files, and `'-b'` option may be used to point to the bind file to process (see Section 19.5 [Running GNAT ELIM], page 196)

If you do not have the appropriate set of tree files and the right bind file, you may create them in the current directory using the following procedure.

Let `Main_Prog` be the name of a main subprogram, and suppose this subprogram is in a file named `'MAIN_PROG.ADB'`.

To create a bind file for GNAT ELIM, run GNAT BIND for the main subprogram. GNAT ELIM can work with both Ada and C bind files; when both are present, it uses the Ada bind file. The following commands will build the program and create the bind file:

```
$ GNAT MAKE /ACTIONS=COMPILE MAIN_PROG
$ GNAT BIND main_prog
```

To create a minimal set of tree files covering the whole program, call GNAT MAKE for this program as follows:

```
$ GNAT MAKE /FORCE_COMPILE /ACTIONS=COMPILE /NOLOAD /TREE_OUTPUT MAIN_PROG
```

The `/ACTIONS=COMPILE` GNAT MAKE option turns off the bind and link steps, that are useless anyway because the sources are compiled with `'/NOLOAD'` option which turns off code generation.

The `/FORCE_COMPILE` GNAT MAKE option forces recompilation of all the needed sources.

This sequence of actions will create all the data needed by GNAT ELIM from scratch and therefore guarantee its consistency. If you would like to use some existing set of files as GNAT ELIM output, you must make sure that the set of files is complete and consistent. You can use the `-m` qualifier to check if there are missed tree files

Note, that GNAT ELIM needs neither object nor ALI files.

19.5 Running GNAT ELIM

GNAT ELIM has the following command-line interface:

```
$ GNAT ELIM [options] name
```

`name` should be a full expanded Ada name of a main subprogram of a program (partition).

GNAT ELIM options:

- `/QUIET` Quiet mode: by default GNAT ELIM generates to the standard error stream a trace of the source file names of the compilation units being processed. This option turns this trace off.
- `/VERBOSE` Verbose mode: GNAT ELIM version information is printed as Ada comments to the standard output stream.
- `/ALL` Also look for subprograms from the GNAT run time that can be eliminated.
- `/MISSED` Check if any tree files are missing for an accurate result.
- `/TREE_DIRS=dir`
 When looking for tree files also look in directory *dir*
- `/BIND_FILE=bind_file`
 Specifies *bind_file* as the bind file to process. If not set, the name of the bind file is computed from the full expanded Ada name of a main subprogram.
- `-dx` Activate internal debugging qualifiers. *x* is a letter or digit, or string of letters or digits, which specifies the type of debugging mode desired. Normally these are used only for internal development or system debugging purposes. You can find full documentation for these qualifiers in the body of the GNAT ELIM.Options unit in the compiler source file ‘GNATELIM-OPTIONS.ADB’.

GNAT ELIM sends its output to the standard output stream, and all the tracing and debug information is sent to the standard error stream. In order to produce a proper GNAT configuration file ‘GNAT.ADC’, redirection must be used:

```
$ PIPE GNAT ELIM MAIN_PROG > GNAT.ADC
```

In order to append the GNAT ELIM output to the existing contents of ‘GNAT.ADC’.

19.6 Correcting the List of Eliminate Pragmas

In some rare cases it may happen that GNAT ELIM will try to eliminate subprograms which are actually called in the program. In this case, the compiler will generate an error message of the form:

```
FILE.ADB:106:07: cannot call eliminated subprogram "My_Prog"
```

You will need to manually remove the wrong `Eliminate` pragmas from the ‘GNAT.ADC’ file. It is advised that you recompile your program from scratch after that because you need a consistent ‘GNAT.ADC’ file during the entire compilation.

19.7 Making Your Executables Smaller

In order to get a smaller executable for your program you now have to recompile the program completely with the new ‘GNAT.ADC’ file created by GNAT ELIM in your current directory:

```
$ GNAT MAKE /FORCE_COMPILE MAIN_PROG
```

(you will need `/FORCE_COMPILE` option for GNAT MAKE to recompile everything with the set of pragmas `Eliminate` you have obtained with GNAT ELIM).

Be aware that the set of `Eliminate` pragmas is specific to each program. It is not recommended to merge sets of `Eliminate` pragmas created for different programs in one ‘GNAT.ADC’ file.

19.8 Summary of the GNAT ELIM Usage Cycle

Here is a quick summary of the steps to be taken in order to reduce the size of your executables with GNAT ELIM. You may use other GNAT options to control the optimization level, to produce the debugging information, to set search path, etc.

1. Produce a bind file and a set of tree files

```
$ GNAT MAKE /ACTIONS=COMPILE MAIN_PROG
$ GNAT BIND main_prog
$ GNAT MAKE /FORCE_COMPILE /NO_LINK /NOLOAD /TREE_OUTPUT MAIN_PROG
```

2. Generate a list of `Eliminate` pragmas

```
$ PIPE GNAT ELIM MAIN_PROG > GNAT.ADC
```

3. Recompile the application

```
$ GNAT MAKE /FORCE_COMPILE MAIN_PROG
```


20 Other Utility Programs

This chapter discusses some other utility programs available in the Ada environment.

20.1 Using Other Utility Programs with GNAT

The object files generated by GNAT are in standard system format and in particular the debugging information uses this format. This means programs generated by GNAT can be used with existing utilities that depend on these formats.

20.2 The GNAT STANDARD Utility Program

Many of the definitions in package Standard are implementation-dependent. However, the source of this package does not exist as an Ada source file, so these values cannot be determined by inspecting the source. They can be determined by examining in detail the coding of ‘CSTAND.ADB’ which creates the image of Standard in the compiler, but this is awkward and requires a great deal of internal knowledge about the system.

The GNAT STANDARD utility is designed to deal with this situation. It is an Ada program that dynamically determines the values of all the relevant parameters in Standard, and prints them out in the form of an Ada source listing for Standard, displaying all the values of interest. This output is generated to ‘SYS\$OUTPUT’.

To determine the value of any parameter in package Standard, simply run GNAT STANDARD with no qualifiers or arguments, and examine the output. This is preferable to consulting documentation, because you know that the values you are getting are the actual ones provided by the executing system.

20.3 The External Symbol Naming Scheme of GNAT

In order to interpret the output from GNAT, when using tools that are originally intended for use with other languages, it is useful to understand the conventions used to generate link names from the Ada entity names.

All link names are in all lowercase letters. With the exception of library procedure names, the mechanism used is simply to use the full expanded Ada name with dots replaced by double underscores. For example, suppose we have the following package spec:

```
package QRS is
  MN : Integer;
end QRS;
```

The variable MN has a full expanded Ada name of QRS.MN, so the corresponding link name is qrs__mn. Of course if a pragma Export is used this may be overridden:

```

package Exports is
  Var1 : Integer;
  pragma Export (Var1, C, External_Name => "var1_name");
  Var2 : Integer;
  pragma Export (Var2, C, Link_Name => "var2_link_name");
end Exports;

```

In this case, the link name for *Var1* is whatever link name the C compiler would assign for the C function *var1_name*. This typically would be either *var1_name* or *_var1_name*, depending on operating system conventions, but other possibilities exist. The link name for *Var2* is *var2.link_name*, and this is not operating system dependent.

One exception occurs for library level procedures. A potential ambiguity arises between the required name *_main* for the C main program, and the name we would otherwise assign to an Ada library level procedure called *Main* (which might well not be the main program).

To avoid this ambiguity, we attach the prefix *_ada_* to such names. So if we have a library level procedure such as

```

procedure Hello (S : String);

```

the external name of this procedure will be *_ada.hello*.

20.4 Ada Mode for Glide

The Glide mode for programming in Ada (both, Ada83 and Ada95) helps the user in understanding existing code and facilitates writing new code. It furthermore provides some utility functions for easier integration of standard EMACS features when programming in Ada.

20.4.1 General Features:

- Full Integrated Development Environment :
 - support of 'project files' for the configuration (directories, compilation options,...)
 - compiling and stepping through error messages.
 - running and debugging your applications within Glide.
- easy to use for beginners by pull-down menus,
- user configurable by many user-option variables.

20.4.2 Ada Mode Features That Help Understanding Code:

- functions for easy and quick stepping through Ada code,
- getting cross reference information for identifiers (e.g. find the defining place by a keystroke),

- displaying an index menu of types and subprograms and move point to the chosen one,
- automatic color highlighting of the various entities in Ada code.

20.4.3 Glide Support for Writing Ada Code:

- switching between spec and body files with possible autogeneration of body files,
- automatic formatting of subprograms parameter lists.
- automatic smart indentation according to Ada syntax,
- automatic completion of identifiers,
- automatic casing of identifiers, keywords, and attributes,
- insertion of statement templates,
- filling comment paragraphs like filling normal text,

For more information, please refer to the online Glide documentation available in the Glide → Help Menu.

20.5 Converting Ada Files to html with gnathtml

This Perl script allows Ada source files to be browsed using standard Web browsers. For installation procedure, see the section See Section 20.6 [Installing gnathtml], page 202.

Ada reserved keywords are highlighted in a bold font and Ada comments in a blue font. Unless your program was compiled with the GNAT COMPILE `‘/XREF=SUPPRESS’` qualifier to suppress the generation of cross-referencing information, user defined variables and types will appear in a different color; you will be able to click on any identifier and go to its declaration.

The command line is as follow:

```
$ perl gnathtml.pl [qualifiers] ada-files
```

You can pass it as many Ada files as you want. **gnathtml** will generate an html file for every ada file, and a global file called `‘index.htm’`. This file is an index of every identifier defined in the files.

The available qualifiers are the following ones :

- | | |
|------------------|---|
| -83 | Only the subset on the Ada 83 keywords will be highlighted, not the full Ada 95 keywords set. |
| -cc <i>color</i> | This option allows you to change the color used for comments. The default value is green. The color argument can be any name accepted by html. |
| -d | If the ada files depend on some other files (using for instance the <code>with</code> command, the latter will also be converted to html. Only the files in the user project will be converted to html, not the files in the run-time library itself. |
| -D | This command is the same as -d above, but gnathtml will also look for files in the run-time library, and generate html files for them. |

- f** By default, gnathtml will generate html links only for global entities ('with'ed units, global variables and types,...). If you specify the **-f** on the command line, then links will be generated for local entities too.
- l *number*** If this qualifier is provided and *number* is not 0, then **gnathtml** will number the html files every *number* line.
- I *dir*** Specify a directory to search for library files ('.ALI' files) and source files. You can provide several **-I** qualifiers on the command line, and the directories will be parsed in the order of the command line.
- o *dir*** Specify the output directory for html files. By default, gnathtml will saved the generated html files in a subdirectory named 'html/'.
- p *file*** If you are using EMACS and the most recent EMACS Ada mode, which provides a full Integrated Development Environment for compiling, checking, running and debugging applications, you may be using '.adp' files to give the directories where EMACS can find sources and object files.

Using this qualifier, you can tell gnathtml to use these files. This allows you to get an html version of your application, even if it is spread over multiple directories.
- sc *color*** This option allows you to change the color used for symbol definitions. The default value is red. The color argument can be any name accepted by html.
- t *file*** This qualifier provides the name of a file. This file contains a list of file names to be converted, and the effect is exactly as though they had appeared explicitly on the command line. This is the recommended way to work around the command line length limit on some systems.

20.6 Installing gnathtml

Perl needs to be installed on your machine to run this script. Perl is freely available for almost every architecture and Operating System via the Internet.

On Unix systems, you may want to modify the first line of the script **gnathtml**, to explicitly tell the Operating system where Perl is. The syntax of this line is :

```
#!/full_path_name_to_perl
```

Alternatively, you may run the script using the following command line:

```
$ perl gnathtml.pl [qualifiers] files
```

20.7 LSE

The GNAT distribution provides an Ada 95 template for the Digital Language Sensitive Editor (LSE), a component of DECset. In order to access it, invoke LSE with the qualifier `/ENVIRONMENT=GNU:[LIB]ADA95.ENV`.

20.8 Profiling

GNAT supports The Digital Performance Coverage Analyzer (PCA), a component of DECset.

To use it proceed as outlined under "HELP PCA", except for running the collection phase with the /DEBUG qualifier.

```
$ GNAT MAKE /DEBUG <PROGRAM_NAME>
$ DEFINE LIB$DEBUG PCA$COLLECTOR
$ RUN/DEBUG <PROGRAM_NAME>
```


21 Running and Debugging Ada Programs

This chapter discusses how to debug Ada programs. An incorrect Ada program may be handled in three ways by the GNAT compiler:

1. The illegality may be a violation of the static semantics of Ada. In that case GNAT diagnoses the constructs in the program that are illegal. It is then a straightforward matter for the user to modify those parts of the program.
2. The illegality may be a violation of the dynamic semantics of Ada. In that case the program compiles and executes, but may generate incorrect results, or may terminate abnormally with some exception.
3. When presented with a program that contains convoluted errors, GNAT itself may terminate abnormally without providing full diagnostics on the incorrect user program.

21.1 The GNAT Debugger GDB

GDB is a general purpose, platform-independent debugger that can be used to debug mixed-language programs compiled with GCC, and in particular is capable of debugging Ada programs compiled with GNAT. The latest versions of GDB are Ada-aware and can handle complex Ada data structures.

The manual *Debugging with GDB*, located in the GNU:[DOCS] directory, contains full details on the usage of GDB, including a section on its usage on programs. This manual should be consulted for full details. The section that follows is a brief introduction to the philosophy and use of GDB.

When GNAT programs are compiled, the compiler optionally writes debugging information into the generated object file, including information on line numbers, and on declared types and variables. This information is separate from the generated code. It makes the object files considerably larger, but it does not add to the size of the actual executable that will be loaded into memory, and has no impact on run-time performance. The generation of debug information is triggered by the use of the `/DEBUG` qualifier in the GNAT `COMPILE` or `GNAT MAKE` command used to carry out the compilations. It is important to emphasize that the use of these options does not change the generated code.

The debugging information is written in standard system formats that are used by many tools, including debuggers and profilers. The format of the information is typically designed to describe C types and semantics, but GNAT implements a translation scheme which allows full details about Ada types and variables to be encoded into these standard C formats. Details of this encoding scheme may be found in the file `EXP_DEBUG.ADS` in the GNAT source distribution. However, the details of this encoding are, in general, of no interest to a user, since GDB automatically performs the necessary decoding.

When a program is bound and linked, the debugging information is collected from the object files, and stored in the executable image of the program. Again, this process significantly increases the size of the generated executable file, but it does not increase the size of the executable program itself. Furthermore, if this program is run in the normal manner, it runs exactly as if the debug information were not present, and takes no more actual memory.

However, if the program is run under control of GDB, the debugger is activated. The image of the program is loaded, at which point it is ready to run. If a run command is given, then the program will run exactly as it would have if GDB were not present. This is a crucial part of the GDB design philosophy. GDB is entirely non-intrusive until a breakpoint is encountered. If no breakpoint is ever hit, the program will run exactly as it would if no debugger were present. When a breakpoint is hit, GDB accesses the debugging information and can respond to user commands to inspect variables, and more generally to report on the state of execution.

21.2 Running GDB

The debugger can be launched directly and simply from **glide** or through its graphical interface: **gvd**. It can also be used directly in text mode. Here is described the basic use of GDB in text mode. All the commands described below can be used in the **gvd** console window eventhough there is usually other more graphical ways to achieve the same goals.

The command to run GDB in text mode is

```
$ $ GDB PROGRAM
```

where **PROGRAM** is the name of the executable file. This activates the debugger and results in a prompt for debugger commands. The simplest command is simply **run**, which causes the program to run exactly as if the debugger were not present. The following section describes some of the additional commands that can be given to GDB.

21.3 Introduction to GDB Commands

GDB contains a large repertoire of commands. The manual *Debugging with GDB*, located in the GNU:[DOCS] directory, includes extensive documentation on the use of these commands, together with examples of their use. Furthermore, the command *help* invoked from within GDB activates a simple help facility which summarizes the available commands and their options. In this section we summarize a few of the most commonly used commands to give an idea of what GDB is about. You should create a simple program with debugging information and experiment with the use of these GDB commands on the program as you read through the following section.

set args arguments

The *arguments* list above is a list of arguments to be passed to the program on a subsequent run command, just as though the arguments had been entered on a normal invocation of the program. The **set args** command is not needed if the program does not require arguments.

run

The **run** command causes execution of the program to start from the beginning. If the program is already running, that is to say if you are currently positioned at a breakpoint, then a prompt will ask for confirmation that you want to abandon the current execution and restart.

breakpoint location

The breakpoint command sets a breakpoint, that is to say a point at which execution will halt and GDB will await further commands. *location* is either a

line number within a file, given in the format `file:linenumber`, or it is the name of a subprogram. If you request that a breakpoint be set on a subprogram that is overloaded, a prompt will ask you to specify on which of those subprograms you want to breakpoint. You can also specify that all of them should be breakpointed. If the program is run and execution encounters the breakpoint, then the program stops and GDB signals that the breakpoint was encountered by printing the line of code before which the program is halted.

breakpoint exception name

A special form of the breakpoint command which breakpoints whenever exception *name* is raised. If *name* is omitted, then a breakpoint will occur when any exception is raised.

print expression

This will print the value of the given expression. Most simple Ada expression formats are properly handled by GDB, so the expression can contain function calls, variables, operators, and attribute references.

continue Continues execution following a breakpoint, until the next breakpoint or the termination of the program.

step Executes a single line after a breakpoint. If the next statement is a subprogram call, execution continues into (the first statement of) the called subprogram.

next Executes a single line. If this line is a subprogram call, executes and returns from the call.

list Lists a few lines around the current source location. In practice, it is usually more convenient to have a separate edit window open with the relevant source file displayed. Successive applications of this command print subsequent lines. The command can be given an argument which is a line number, in which case it displays a few lines around the specified one.

backtrace

Displays a backtrace of the call chain. This command is typically used after a breakpoint has occurred, to examine the sequence of calls that leads to the current breakpoint. The display includes one line for each activation record (frame) corresponding to an active subprogram.

up At a breakpoint, GDB can display the values of variables local to the current frame. The command `up` can be used to examine the contents of other active frames, by moving the focus up the stack, that is to say from callee to caller, one frame at a time.

down Moves the focus of GDB down from the frame currently being examined to the frame of its callee (the reverse of the previous command),

frame n Inspect the frame with the given number. The value 0 denotes the frame of the current breakpoint, that is to say the top of the call stack.

The above list is a very short introduction to the commands that GDB provides. Important additional capabilities, including conditional breakpoints, the ability to execute command sequences on a breakpoint, the ability to debug at the machine instruction level and many

other features are described in detail in *Debugging with GDB*. Note that most commands can be abbreviated (for example, `c` for continue, `bt` for backtrace).

21.4 Using Ada Expressions

GDB supports a fairly large subset of Ada expression syntax, with some extensions. The philosophy behind the design of this subset is

- That GDB should provide basic literals and access to operations for arithmetic, dereferencing, field selection, indexing, and subprogram calls, leaving more sophisticated computations to subprograms written into the program (which therefore may be called from GDB).
- That type safety and strict adherence to Ada language restrictions are not particularly important to the GDB user.
- That brevity is important to the GDB user.

Thus, for brevity, the debugger acts as if there were implicit `with` and `use` clauses in effect for all user-written packages, thus making it unnecessary to fully qualify most names with their packages, regardless of context. Where this causes ambiguity, GDB asks the user's intent.

For details on the supported Ada syntax, see *Debugging with GDB*.

21.5 Calling User-Defined Subprograms

An important capability of GDB is the ability to call user-defined subprograms while debugging. This is achieved simply by entering a subprogram call statement in the form:

```
call subprogram-name (parameters)
```

The keyword `call` can be omitted in the normal case where the `subprogram-name` does not coincide with any of the predefined GDB commands.

The effect is to invoke the given subprogram, passing it the list of parameters that is supplied. The parameters can be expressions and can include variables from the program being debugged. The subprogram must be defined at the library level within your program, and GDB will call the subprogram within the environment of your program execution (which means that the subprogram is free to access or even modify variables within your program).

The most important use of this facility is in allowing the inclusion of debugging routines that are tailored to particular data structures in your program. Such debugging routines can be written to provide a suitably high-level description of an abstract type, rather than a low-level dump of its physical layout. After all, the standard GDB `print` command only knows the physical layout of your types, not their abstract meaning. Debugging routines can provide information at the desired semantic level and are thus enormously useful.

For example, when debugging GNAT itself, it is crucial to have access to the contents of the tree nodes used to represent the program internally. But tree nodes are represented simply by an integer value (which in turn is an index into a table of nodes). Using the `print` command on a tree node would simply print this integer value, which is not very useful. But the `PN` routine (defined in file `TREEPR.ADB` in the GNAT sources) takes a

tree node as input, and displays a useful high level representation of the tree node, which includes the syntactic category of the node, its position in the source, the integers that denote descendant nodes and parent node, as well as varied semantic information. To study this example in more detail, you might want to look at the body of the PN procedure in the stated file.

21.6 Using the Next Command in a Function

When you use the **next** command in a function, the current source location will advance to the next statement as usual. A special case arises in the case of a **return** statement.

Part of the code for a return statement is the "epilog" of the function. This is the code that returns to the caller. There is only one copy of this epilog code, and it is typically associated with the last return statement in the function if there is more than one return. In some implementations, this epilog is associated with the first statement of the function.

The result is that if you use the **next** command from a return statement that is not the last return statement of the function you may see a strange apparent jump to the last return statement or to the start of the function. You should simply ignore this odd jump. The value returned is always that from the first return statement that was stepped through.

21.7 Breaking on Ada Exceptions

You can set breakpoints that trip when your program raises selected exceptions.

break exception

Set a breakpoint that trips whenever (any task in the) program raises any exception.

break exception *name*

Set a breakpoint that trips whenever (any task in the) program raises the exception *name*.

break exception unhandled

Set a breakpoint that trips whenever (any task in the) program raises an exception for which there is no handler.

info exceptions

info exceptions *regexp*

The **info exceptions** command permits the user to examine all defined exceptions within Ada programs. With a regular expression, *regexp*, as argument, prints out only those exceptions whose name matches *regexp*.

21.8 Ada Tasks

GDB allows the following task-related commands:

info tasks

This command shows a list of current Ada tasks, as in the following example:

```
(GDB) info tasks
  ID      TID P-ID  Thread Pri State      Name
  1      8088000  0   807e000  15 Child Activation Wait main_task
  2      80a4000  1   80ae000  15 Accept/Select Wait    b
  3      809a800  1   80a4800  15 Child Activation Wait a
  * 4      80ae800  3   80b8000  15 Running      c
```

In this listing, the asterisk before the first task indicates it to be the currently running task. The first column lists the task ID that is used to refer to tasks in the following commands.

```
break linespec task taskid
break linespec task taskid if ...
```

These commands are like the `break ... thread linespec` specifies source lines.

Use the qualifier ‘`task taskid`’ with a breakpoint command to specify that you only want GDB to stop the program when a particular Ada task reaches this breakpoint. *taskid* is one of the numeric task identifiers assigned by GDB, shown in the first column of the ‘`info tasks`’ display.

If you do not specify ‘`task taskid`’ when you set a breakpoint, the breakpoint applies to *all* tasks of your program.

You can use the `task` qualifier on conditional breakpoints as well; in this case, place ‘`task taskid`’ before the breakpoint condition (before the `if`).

```
task taskno
```

This command allows to qualifier to the task referred by *taskno*. In particular, This allows to browse the backtrace of the specified task. It is advised to qualifier back to the original task before continuing execution otherwise the scheduling of the program may be perturbed.

For more detailed information on the tasking support, see *Debugging with GDB*.

21.9 Debugging Generic Units

GNAT always uses code expansion for generic instantiation. This means that each time an instantiation occurs, a complete copy of the original code is made, with appropriate substitutions of formals by actuals.

It is not possible to refer to the original generic entities in GDB, but it is always possible to debug a particular instance of a generic, by using the appropriate expanded names. For example, if we have

```

procedure g is

  generic package k is
    procedure kp (v1 : in out integer);
  end k;

  package body k is
    procedure kp (v1 : in out integer) is
      begin
        v1 := v1 + 1;
      end kp;
    end k;

  package k1 is new k;
  package k2 is new k;

  var : integer := 1;

begin
  k1.kp (var);
  k2.kp (var);
  k1.kp (var);
  k2.kp (var);
end;

```

Then to break on a call to procedure kp in the k2 instance, simply use the command:

```
(GDB) break g.k2.kp
```

When the breakpoint occurs, you can step through the code of the instance in the normal manner and examine the values of local variables, as for other units.

21.10 GNAT Abnormal Termination or Failure to Terminate

When presented with programs that contain serious errors in syntax or semantics, GNAT may on rare occasions experience problems in operation, such as aborting with a segmentation fault or illegal memory access, raising an internal exception, terminating abnormally, or failing to terminate at all. In such cases, you can activate various features of GNAT that can help you pinpoint the construct in your program that is the likely source of the problem.

The following strategies are presented in increasing order of difficulty, corresponding to your experience in using GNAT and your familiarity with compiler internals.

1. Run **GNAT COMPILER** with the `‘/REPORT_ERRORS=FULL’`. This first qualifier causes all errors on a given line to be reported. In its absence, only the first error on a line is displayed.

The `‘/REPORT_ERRORS=IMMEDIATE’` qualifier causes errors to be displayed as soon as they are encountered, rather than after compilation is terminated. If GNAT terminates prematurely or goes into an infinite loop, the last error message displayed may help to pinpoint the culprit.

2. Run **GNAT_COMPILE** with the **/VERBOSE** qualifier. In this mode, **GNAT_COMPILE** produces ongoing information about the progress of the compilation and provides the name of each procedure as code is generated. This qualifier allows you to find which Ada procedure was being compiled when it encountered a code generation problem.
3. Run **GNAT_COMPILE** with the **'/TRACE_UNITS'** qualifier. This is a GNAT specific qualifier that does for the front-end what **VERBOSE** does for the back end. The system prints the name of each unit, either a compilation unit or nested unit, as it is being analyzed.
4. Finally, you can start **GDB** directly on the **GNAT1** executable. **GNAT1** is the front-end of GNAT, and can be run independently (normally it is just called from **GNAT_COMPILE**). You can use **GDB** on **GNAT1** as you would on a C program (but see Section 21.1 [The GNAT Debugger **GDB**], page 205 for caveats). The **where** command is the first line of attack; the variable **lineno** (seen by **print lineno**), used by the second phase of **GNAT1** and by the **GNAT_COMPILE** backend, indicates the source line at which the execution stopped, and **input_file name** indicates the name of the source file.

21.11 Naming Conventions for GNAT Source Files

In order to examine the workings of the GNAT system, the following brief description of its organization may be helpful:

- Files with prefix **'SC'** contain the lexical scanner.
- All files prefixed with **'PAR'** are components of the parser. The numbers correspond to chapters of the Ada 95 Reference Manual. For example, parsing of select statements can be found in **'PAR-CH9.ADB'**.
- All files prefixed with **'SEM'** perform semantic analysis. The numbers correspond to chapters of the Ada standard. For example, all issues involving context clauses can be found in **'SEM-CH10.ADB'**. In addition, some features of the language require sufficient special processing to justify their own semantic files: **sem_aggr** for aggregates, **sem_disp** for dynamic dispatching, etc.
- All files prefixed with **'EXP'** perform normalization and expansion of the intermediate representation (abstract syntax tree, or AST). these files use the same numbering scheme as the parser and semantics files. For example, the construction of record initialization procedures is done in **'EXP-CH3.ADB'**.
- The files prefixed with **'BIND'** implement the binder, which verifies the consistency of the compilation, determines an order of elaboration, and generates the bind file.
- The files **'ATREE.ADS'** and **'ATREE.ADB'** detail the low-level data structures used by the front-end.
- The files **'SINFO.ADS'** and **'SINFO.ADB'** detail the structure of the abstract syntax tree as produced by the parser.
- The files **'EINFO.ADS'** and **'EINFO.ADB'** detail the attributes of all entities, computed during semantic analysis.
- Library management issues are dealt with in files with prefix **'LIB'**.
- Ada files with the prefix **'A-'** are children of **Ada**, as defined in Annex A.
- Files with prefix **'I-'** are children of **Interfaces**, as defined in Annex B.

- Files with prefix ‘S-’ are children of **System**. This includes both language-defined children and GNAT run-time routines.
- Files with prefix ‘G-’ are children of **GNAT**. These are useful general-purpose packages, fully documented in their specifications. All the other ‘.C’ files are modifications of common GNAT **COMPILE** files.

21.12 Getting Internal Debugging Information

Most compilers have internal debugging qualifiers and modes. GNAT does also, except GNAT internal debugging qualifiers and modes are not secret. A summary and full description of all the compiler and binder debug flags are in the file ‘**DEBUG.ADB**’. You must obtain the sources of the compiler to see the full detailed effects of these flags.

The qualifiers that print the source of the program (reconstructed from the internal tree) are of general interest for user programs, as are the options to print the full internal tree, and the entity table (the symbol table information). The reconstructed source provides a readable version of the program after the front-end has completed analysis and expansion, and is useful when studying the performance of specific constructs. For example, constraint checks are indicated, complex aggregates are replaced with loops and assignments, and tasking primitives are replaced with run-time calls.

21.13 Stack Traceback

Traceback is a mechanism to display the sequence of subprogram calls that leads to a specified execution point in a program. Often (but not always) the execution point is an instruction at which an exception has been raised. This mechanism is also known as *stack unwinding* because it obtains its information by scanning the run-time stack and recovering the activation records of all active subprograms. Stack unwinding is one of the most important tools for program debugging.

The first entry stored in traceback corresponds to the deepest calling level, that is to say the subprogram currently executing the instruction from which we want to obtain the traceback.

Note that there is no runtime performance penalty when stack traceback is enabled and no exception are raised during program execution.

21.13.1 Non-Symbolic Traceback

Note: this feature is not supported on all platforms. See ‘**GNAT.Traceback spec in G-TRACEB.ADS**’ for a complete list of supported platforms.

21.13.1.1 Tracebacks From an Unhandled Exception

A runtime non-symbolic traceback is a list of addresses of call instructions. To enable this feature you must use the **-E GNAT BIND**’s option. With this option a stack traceback is stored as part of exception information. It is possible to retrieve this information using the standard **Ada.Exception.Exception_Information** routine.

Let’s have a look at a simple example:

```

procedure STB is

  procedure P1 is
  begin
    raise Constraint_Error;
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  P2;
end STB;

```

```

$ GNAT MAKE stb /BINDER_QUALIFIERS -E
$ stb

```

```

Execution terminated by unhandled exception
Exception name: CONSTRAINT_ERROR
Message: STB.ADB:5
Call stack traceback locations:
0x401373 0x40138b 0x40139c 0x401335 0x4011c4 0x4011f1 0x77e892a4

```

As we see the traceback lists a sequence of addresses for the unhandled exception `CONSTRAINT_ERROR` raised in procedure P1. It is easy to guess that this exception come from procedure P1. To translate these addresses into the source lines where the calls appear, the `addr2line` tool, described below, is invaluable. The use of this tool requires the program to be compiled with debug information.

```

$ GNAT MAKE -g stb /BINDER_QUALIFIERS -E
$ stb

Execution terminated by unhandled exception
Exception name: CONSTRAINT_ERROR
Message: STB.ADB:5
Call stack traceback locations:
0x401373 0x40138b 0x40139c 0x401335 0x4011c4 0x4011f1 0x77e892a4

$ addr2line --exe=stb 0x401373 0x40138b 0x40139c 0x401335 0x4011c4
0x4011f1 0x77e892a4

00401373 at d:/stb/STB.ADB:5
0040138B at d:/stb/STB.ADB:10
0040139C at d:/stb/STB.ADB:14
00401335 at d:/stb/B~STB.ADB:104
004011C4 at /build/.../CRT1.C:200
004011F1 at /build/.../CRT1.C:222
77E892A4 in ?? at ??:0

```

`addr2line` has a number of other useful options:

`--functions`

to get the function name corresponding to any location

`--demangle=gnat`

to use the **gnat** decoding mode for the function names. Note that for binutils version 2.9.x the option is simply `--demangle`.

```
$ addr2line --exe=stb --functions --demangle=gnat 0x401373 0x40138b
0x40139c 0x401335 0x4011c4 0x4011f1

00401373 in stb.p1 at d:/stb/STB.ADB:5
0040138B in stb.p2 at d:/stb/STB.ADB:10
0040139C in stb at d:/stb/STB.ADB:14
00401335 in main at d:/stb/B~STB.ADB:104
004011C4 in <__mingw_CRTStartup> at /build/.../CRT1.C:200
004011F1 in <mainCRTStartup> at /build/.../CRT1.C:222
```

From this traceback we can see that the exception was raised in ‘STB.ADB’ at line 5, which was reached from a procedure call in ‘STB.ADB’ at line 10, and so on. The ‘B~STD.ADB’ is the binder file, which contains the call to the main program. see Section 4.1 [Running GNAT BIND], page 59. The remaining entries are assorted runtime routines, and the output will vary from platform to platform.

It is also possible to use GDB with these traceback addresses to debug the program. For example, we can break at a given code location, as reported in the stack traceback:

```
$ GDB -nw stb

(GDB) break *0x401373
Breakpoint 1 at 0x401373: file STB.ADB, line 5.
```

It is important to note that the stack traceback addresses do not change when debug information is included. This is particularly useful because it makes it possible to release software without debug information (to minimize object size), get a field report that includes a stack traceback whenever an internal bug occurs, and then be able to retrieve the sequence of calls with the same program compiled with debug information.

21.13.1.2 Tracebacks From Exception Occurrences

Non-symbolic tracebacks are obtained by using the `-E` binder argument. The stack traceback is attached to the exception information string, and can be retrieved in an exception handler within the Ada program, by means of the Ada95 facilities defined in `Ada.Exceptions`. Here is a simple example:

```

with Ada.Text_IO;
with Ada.Exceptions;

procedure STB is

  use Ada;
  use Ada.Exceptions;

  procedure P1 is
    K : Positive := 1;
  begin
    K := K - 1;
  exception
    when E : others =>
      Text_IO.Put_Line (Exception_Information (E));
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  P2;
end STB;

```

This program will output:

```

$ stb

Exception name: CONSTRAINT_ERROR
Message: STB.ADB:12
Call stack traceback locations:
0x4015e4 0x401633 0x401644 0x401461 0x4011c4 0x4011f1 0x77e892a4

```

21.13.1.3 Tracebacks From Anywhere in a Program

It is also possible to retrieve a stack traceback from anywhere in a program. For this you need to use the `GNAT.Traceback` API. This package includes a procedure called `Call_Chain` that computes a complete stack traceback, as well as useful display procedures described below. It is not necessary to use the `-E GNAT BIND` option in this case, because the stack traceback mechanism is invoked explicitly.

In the following example we compute a traceback at a specific location in the program, and we display it using `GNAT.Debug_Uutilities.Image` to convert addresses to strings:

```

with Ada.Text_IO;
with GNAT.Traceback;
with GNAT.Debug_Uilities;

procedure STB is

  use Ada;
  use GNAT;
  use GNAT.Traceback;

  procedure P1 is
    TB : Tracebacks_Array (1 .. 10);
    -- We are asking for a maximum of 10 stack frames.
    Len : Natural;
    -- Len will receive the actual number of stack frames returned.
  begin
    Call_Chain (TB, Len);

    Text_IO.Put ("In STB.P1 : ");

    for K in 1 .. Len loop
      Text_IO.Put (Debug_Uilities.Image (TB (K)));
      Text_IO.Put (' ');
    end loop;

    Text_IO.New_Line;
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  P2;
end STB;

```

```

$ GNAT MAKE stb
$ stb

```

```

In STB.P1 : 16#0040_F1E4# 16#0040_14F2# 16#0040_170B# 16#0040_171C#
16#0040_1461# 16#0040_11C4# 16#0040_11F1# 16#77E8_92A4#

```

21.13.2 Symbolic Traceback

A symbolic traceback is a stack traceback in which procedure names are associated with each code location.

Note that this feature is not supported on all platforms. See ‘GNAT.Traceback.Symbolic spec in G-TRASYM.ADS’ for a complete list of currently supported platforms.

Note that the symbolic traceback requires that the program be compiled with debug information. If it is not compiled with debug information only the non-symbolic information will be valid.

21.13.2.1 Tracebacks From Exception Occurrences

```

with Ada.Text_IO;
with GNAT.Traceback.Symbolic;

procedure STB is

  procedure P1 is
  begin
    raise Constraint_Error;
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

  procedure P3 is
  begin
    P2;
  end P3;

begin
  P3;
exception
  when E : others =>
    Ada.Text_IO.Put_Line (GNAT.Traceback.Symbolic.Symbolic_Traceback (E));
end STB;

```

```

$ GNAT MAKE -g stb /BINDER_QUALIFIERS -E /LINKER_QUALIFIERS -lgnat -laddr2line -lintl
$ stb

```

```

0040149F in stb.p1 at STB.ADB:8
004014B7 in stb.p2 at STB.ADB:13
004014CF in stb.p3 at STB.ADB:18
004015DD in ada.stb at STB.ADB:22
00401461 in main at B~STB.ADB:168
004011C4 in __mingw_CRTStartup at CRT1.C:200
004011F1 in mainCRTStartup at CRT1.C:222
77E892A4 in ?? at ??:0

```

The exact sequence of linker options may vary from platform to platform. The above /LINKER_QUALIFIERS section is for Windows platforms. By contrast, under Unix there is no need for the /LINKER_QUALIFIERS section. Differences across platforms are due to details of linker implementation.

21.13.2.2 Tracebacks From Anywhere in a Program

It is possible to get a symbolic stack traceback from anywhere in a program, just as for non-symbolic tracebacks. The first step is to obtain a non-symbolic traceback, and then call `Symbolic_Traceback` to compute the symbolic information. Here is an example:

```
with Ada.Text_IO;
with GNAT.Traceback;
with GNAT.Traceback.Symbolic;

procedure STB is

  use Ada;
  use GNAT.Traceback;
  use GNAT.Traceback.Symbolic;

  procedure P1 is
    TB : Tracebacks_Array (1 .. 10);
    -- We are asking for a maximum of 10 stack frames.
    Len : Natural;
    -- Len will receive the actual number of stack frames returned.
  begin
    Call_Chain (TB, Len);
    Text_IO.Put_Line (Symbolic_Traceback (TB (1 .. Len)));
  end P1;

  procedure P2 is
  begin
    P1;
  end P2;

begin
  P2;
end STB;
```


22 Compatibility with DEC Ada

This section of the manual compares DEC Ada for OpenVMS Alpha and GNAT OpenVMS Alpha. GNAT achieves a high level of compatibility with DEC Ada, and it should generally be straightforward to port code from the DEC Ada environment to GNAT. However, there are a few language and implementation differences of which the user must be aware. These differences are discussed in this section. In addition, the operating environment and command structure for the compiler are different, and these differences are also discussed.

Note that this discussion addresses specifically the implementation of Ada 83 for DIGITAL OpenVMS Alpha Systems. In cases where the implementation of DEC Ada differs between OpenVMS Alpha Systems and OpenVMS VAX Systems, GNAT always follows the Alpha implementation.

22.1 Ada 95 Compatibility

GNAT is an Ada 95 compiler, and DEC Ada is an Ada 83 compiler. Ada 95 is almost completely upwards compatible with Ada 83, and therefore Ada 83 programs will compile and run under GNAT with no changes or only minor changes. The Ada 95 Reference Manual (ANSI/ISO/IEC-8652:1995) provides details on specific incompatibilities.

GNAT provides the qualifier `/83` on the GNAT `COMPILE` command, as well as the pragma `ADA_83`, to force the compiler to operate in Ada 83 mode. This mode does not guarantee complete conformance to Ada 83, but in practice is sufficient to eliminate most sources of incompatibilities. In particular, it eliminates the recognition of the additional Ada 95 keywords, so that their use as identifiers in Ada83 program is legal, and handles the cases of packages with optional bodies, and generics that instantiate unconstrained types without the use of `(<>)`.

22.2 Differences in the Definition of Package System

Both the Ada 95 and Ada 83 reference manuals permit a compiler to add implementation-dependent declarations to package `System`. In normal mode, GNAT does not take advantage of this permission, and the version of `System` provided by GNAT exactly matches that in the Ada 95 Reference Manual.

However, DEC Ada adds an extensive set of declarations to package `System`, as fully documented in the DEC Ada manuals. To minimize changes required for programs that make use of these extensions, GNAT provides the pragma `Extend_System` for extending the definition of package `System`. By using:

```
pragma Extend_System (Aux_DEC);
```

The set of definitions in `System` is extended to include those in package `System.Aux_DEC`. These definitions are incorporated directly into package `System`, as though they had been declared there in the first place. For a list of the declarations added, see the specification

of this package, which can be found in the file `S-AUXDEC.ADS` in the GNAT library. The pragma `Extend_System` is a configuration pragma, which means that it can be placed in the file `'GNAT.ADC'`, so that it will automatically apply to all subsequent compilations. See the section on Configuration Pragas for further details.

An alternative approach that avoids the use of the non-standard `Extend_System` pragma is to add a context clause to the unit that references these facilities:

```
with System.Aux_DEC;  
use  System.Aux_DEC;
```

The effect is not quite semantically identical to incorporating the declarations directly into package `System`, but most programs will not notice a difference unless they use prefix notation (e.g. `System.Integer_8`) to reference the entities directly in package `System`. For units containing such references, the prefixes must either be removed, or the pragma `Extend_System` must be used.

22.3 Language-Related Features

The following sections highlight differences in types, representations of types, operations, alignment, and related topics.

22.3.1 Integer Types and Representations

The set of predefined integer types is identical in DEC Ada and GNAT. Furthermore the representation of these integer types is also identical, including the capability of size clauses forcing biased representation.

In addition, DEC Ada for OpenVMS Alpha systems has defined the following additional integer types in package `System`:

- `INTEGER_8`
- `INTEGER_16`
- `INTEGER_32`
- `INTEGER_64`
- `LARGEST_INTEGER`

When using GNAT, the first four of these types may be obtained from the standard Ada 95 package `Interfaces`. Alternatively, by use of the pragma `Extend_System`, identical declarations can be referenced directly in package `System`. On both GNAT and DEC Ada, the maximum integer size is 64 bits.

22.3.2 Floating-Point Types and Representations

The set of predefined floating-point types is identical in DEC Ada and GNAT. Furthermore the representation of these floating-point types is also identical. One important difference is

that the default representation for DEC Ada is `VAX_Float`, but the default representation for GNAT is IEEE.

Specific types may be declared to be `VAX_Float` or IEEE, using the pragma `Float_Representation` as described in the DEC Ada documentation. For example, the declarations:

```
type F_Float is digits 6;  
pragma Float_Representation (VAX_Float, F_Float);
```

declare a type `F_Float` that will be represented in `VAX_Float` format. This set of declarations actually appears in `System.Aux_DEC`, which provides the full set of additional floating-point declarations provided in the DEC Ada version of package `System`. This and similar declarations may be accessed in a user program by using pragma `Extend_System`. The use of this pragma, and the related pragma `Long_Float` is described in further detail in the following section.

22.3.3 Pragas `Float_Representation` and `Long_Float`

DEC Ada provides the pragma `Float_Representation`, which acts as a program library qualifier to allow control over the internal representation chosen for the predefined floating-point types declared in the package `Standard`. The format of this pragma is as follows:

```
pragma Float_Representation(VAX_Float | IEEE_Float);
```

This pragma controls the representation of floating-point types as follows:

- `VAX_Float` specifies that floating-point types are represented by default with the VAX hardware types F-floating, D-floating, G-floating. Note that the H-floating type is available only on DIGITAL Vax systems, and is not available in either DEC Ada or GNAT for Alpha systems.
- `IEEE_Float` specifies that floating-point types are represented by default with the IEEE single and double floating-point types.

GNAT provides an identical implementation of the pragma `Float_Representation`, except that it functions as a configuration pragma, as defined by Ada 95. Note that the notion of configuration pragma corresponds closely to the DEC Ada notion of a program library qualifier.

When no pragma is used in GNAT, the default is `IEEE_Float`, which is different from DEC Ada 83, where the default is `VAX_Float`. In addition, the predefined libraries in GNAT are built using `IEEE_Float`, so it is not advisable to change the format of numbers passed to standard library routines, and if necessary explicit type conversions may be needed.

The use of `IEEE_Float` is recommended in GNAT since it is more efficient, and (given that it conforms to an international standard) potentially more portable. The situation in

which `VAX_Float` may be useful is in interfacing to existing code and data that expects the use of `VAX_Float`. There are two possibilities here. If the requirement for the use of `VAX_Float` is localized, then the best approach is to use the predefined `VAX_Float` types in package `System`, as extended by `Extend_System`. For example, use `System.F_Float` to specify the 32-bit `F_Float` format.

Alternatively, if an entire program depends heavily on the use of the `VAX_Float` and in particular assumes that the types in package `Standard` are in `Vax_Float` format, then it may be desirable to reconfigure GNAT to assume `Vax_Float` by default. This is done by using the GNAT `LIBRARY` command to rebuild the library, and then using the general form of the `Float_Representation` pragma to ensure that this default format is used throughout. The form of the GNAT `LIBRARY` command is:

```
GNAT LIBRARY /CONFIG=file /CREATE=directory
```

where *file* contains the new configuration pragmas and *directory* is the directory to be created to contain the new library.

On OpenVMS systems, DEC Ada provides the pragma `Long_Float` to allow control over the internal representation chosen for the predefined type `Long_Float` and for floating-point type declarations with digits specified in the range 7 .. 15. The format of this pragma is as follows:

```
pragma Long_Float (D_FLOAT | G_FLOAT);
```

22.3.4 Fixed-Point Types and Representations

On DEC Ada for OpenVMS Alpha systems, rounding is away from zero for both positive and negative numbers. Therefore, +0.5 rounds to 1 and -0.5 rounds to -1.

On GNAT for OpenVMS Alpha, the results of operations on fixed-point types are in accordance with the Ada 95 rules. In particular, results of operations on decimal fixed-point types are truncated.

22.3.5 Record and Array Component Alignment

On DEC Ada for OpenVMS Alpha, all non composite components are aligned on natural boundaries. For example, 1-byte components are aligned on byte boundaries, 2-byte components on 2-byte boundaries, 4-byte components on 4-byte boundaries, and so on. The OpenVMS Alpha hardware runs more efficiently with naturally aligned data.

ON GNAT for OpenVMS Alpha, alignment rules are compatible with DEC Ada for OpenVMS Alpha.

22.3.6 Address Clauses

In DEC Ada and GNAT, address clauses are supported for objects and imported subprograms. The predefined type `System.Address` is a private type in both compilers, with the same representation (it is simply a machine pointer). Addition, subtraction, and comparison

operations are available in the standard Ada 95 package `System.Storage_Elements`, or in package `System` if it is extended to include `System.Aux_DEC` using a pragma `Extend_System` as previously described.

Note that code that with's both this extended package `System` and the package `System.Storage_Elements` should not use both packages, or ambiguities will result. In general it is better not to mix these two sets of facilities. The Ada 95 package was designed specifically to provide the kind of features that DEC Ada adds directly to package `System`.

GNAT is compatible with DEC Ada in its handling of address clauses, except for some limitations in the form of address clauses for composite objects with initialization. Such address clauses are easily replaced by the use of an explicitly-defined constant as described in the Ada 95 Reference Manual (13.1(22)). For example, the sequence of declarations:

```
X, Y : Integer := Init_Func;
Q : String (X .. Y) := "abc";
...
for Q'Address use Compute_Address;
```

will be rejected by GNAT, since the address cannot be computed at the time that `Q` is declared. To achieve the intended effect, write instead:

```
X, Y : Integer := Init_Func;
Q_Address : constant Address := Compute_Address;
Q : String (X .. Y) := "abc";
...
for Q'Address use Q_Address;
```

which will be accepted by GNAT (and other Ada 95 compilers), and is also backwards compatible with Ada 83. A fuller description of the restrictions on address specifications is found in the GNAT Reference Manual.

22.3.7 Other Representation Clauses

GNAT supports in a compatible manner all the representation clauses supported by DEC Ada. In addition, it supports representation clause forms that are new in Ada 95 including `COMPONENT_SIZE` and `SIZE` clauses for objects.

22.4 The Package STANDARD

The package `STANDARD`, as implemented by DEC Ada, is fully described in the Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A-1983) and in the DEC Ada Language Reference Manual. As implemented by GNAT, the package `STANDARD` is described in the Ada 95 Reference Manual.

In addition, DEC Ada supports the Latin-1 character set in the type `CHARACTER`. GNAT supports the Latin-1 character set in the type `CHARACTER` and also Unicode (ISO 10646 BMP) in the type `WIDE_CHARACTER`.

The floating-point types supported by GNAT are those supported by DEC Ada, but defaults are different, and are controlled by pragmas. See Section 22.3.2 [Floating-Point Types and Representations], page 222 for details.

22.5 The Package `SYSTEM`

DEC Ada provides a system-specific version of the package `SYSTEM` for each platform on which the language ships. For the complete specification of the package `SYSTEM`, see Appendix F of the DEC Ada Language Reference Manual.

On DEC Ada, the package `SYSTEM` includes the following conversion functions:

- `TO_ADDRESS(INTEGER)`
- `TO_ADDRESS(UNSIGNED_LONGWORD)`
- `TO_ADDRESS(universal_integer)`
- `TO_INTEGER(ADDRESS)`
- `TO_UNSIGNED_LONGWORD(ADDRESS)`
- Function `IMPORT_VALUE` return `UNSIGNED_LONGWORD` and the functions `IMPORT_ADDRESS` and `IMPORT_LARGEST_VALUE`

By default, GNAT supplies a version of `SYSTEM` that matches the definition given in the Ada 95 Reference Manual. This is a subset of the `DIGITAL` system definitions, which is as close as possible to the original definitions. The only difference is that the definition of `SYSTEM_NAME` is different:

```
type Name is (SYSTEM_NAME_GNAT);
System_Name : constant Name := SYSTEM_NAME_GNAT;
```

Also, GNAT adds the new Ada 95 declarations for `BIT_ORDER` and `DEFAULT_BIT_ORDER`.

However, the use of the following pragma causes GNAT to extend the definition of package `SYSTEM` so that it encompasses the full set of `DIGITAL`-specific extensions, including the functions listed above:

```
pragma Extend_System (Aux_DEC);
```

The pragma `Extend_System` is a configuration pragma that is most conveniently placed in the '`GNAT.ADC`' file. See the GNAT Reference Manual for further details.

DEC Ada does not allow the recompilation of the package `SYSTEM`. Instead DEC Ada provides several pragmas (`SYSTEM_NAME`, `STORAGE_UNIT`, and `MEMORY_SIZE`) to

modify values in the package `SYSTEM`. On OpenVMS Alpha systems, the pragma `SYSTEM_NAME` takes the enumeration literal `OPENVMS_AXP` as its single argument.

GNAT does permit the recompilation of package `SYSTEM` using a special qualifier (`/STYLE=GNAT`) and this qualifier can be used if it is necessary to change constants in `SYSTEM`. GNAT does not permit the specification of `SYSTEM_NAME`, `STORAGE_UNIT` or `MEMORY_SIZE` by any other means.

On GNAT systems, the pragma `SYSTEM_NAME` takes the enumeration literal `SYSTEM_NAME_GNAT`.

The definitions provided by the use of

```
pragma Extend_System (AUX_Dec);
```

are virtually identical to those provided by the DEC Ada 83 package `System`. One important difference is that the name of the `TO_ADDRESS` function for type `UNSIGNED_LONGWORD` is changed to `TO_ADDRESS_LONG`. See the GNAT Reference manual for a discussion of why this change was necessary.

The version of `TO_ADDRESS` taking a universal integer argument is in fact an extension to Ada 83 not strictly compatible with the reference manual. In GNAT, we are constrained to be exactly compatible with the standard, and this means we cannot provide this capability. In DEC Ada 83, the point of this definition is to deal with a call like:

```
TO_ADDRESS (16#12777#);
```

Normally, according to the Ada 83 standard, one would expect this to be ambiguous, since it matches both the `INTEGER` and `UNSIGNED_LONGWORD` forms of `TO_ADDRESS`. However, in DEC Ada 83, there is no ambiguity, since the definition using `universal_integer` takes precedence.

In GNAT, since the version with `universal_integer` cannot be supplied, it is not possible to be 100% compatible. Since there are many programs using numeric constants for the argument to `TO_ADDRESS`, the decision in GNAT was to change the name of the function in the `UNSIGNED_LONGWORD` case, so the declarations provided in the GNAT version of `AUX_Dec` are:

```
function To_Address (X : Integer) return Address;
pragma Pure_Function (To_Address);

function To_Address_Long (X : Unsigned_Longword) return Address;
pragma Pure_Function (To_Address_Long);
```

This means that programs using `TO_ADDRESS` for `UNSIGNED_LONGWORD` must change the name to `TO_ADDRESS_LONG`.

22.6 Tasking and Task-Related Features

The concepts relevant to a comparison of tasking on GNAT and on DEC Ada for OpenVMS Alpha systems are discussed in the following sections.

For detailed information on concepts related to tasking in DEC Ada, see the DEC Ada Language Reference Manual and the relevant run-time reference manual.

22.7 Implementation of Tasks in DEC Ada for OpenVMS Alpha Systems

On OpenVMS Alpha systems, each Ada task (except a passive task) is implemented as a single stream of execution that is created and managed by the kernel. On these systems, DEC Ada tasking support is based on DECthreads, an implementation of the POSIX standard for threads.

Although tasks are implemented as threads, all tasks in an Ada program are part of the same process. As a result, resources such as open files and virtual memory can be shared easily among tasks. Having all tasks in one process allows better integration with the programming environment (the shell and the debugger, for example).

Also, on OpenVMS Alpha systems, DEC Ada tasks and foreign code that calls DECthreads routines can be used together. The interaction between Ada tasks and DECthreads routines can have some benefits. For example when on OpenVMS Alpha, DEC Ada can call C code that is already threaded. GNAT on OpenVMS Alpha uses the facilities of DECthreads, and Ada tasks are mapped to threads.

22.7.1 Assigning Task IDs

The DEC Ada Run-Time Library always assigns %TASK 1 to the environment task that executes the main program. On OpenVMS Alpha systems, %TASK 0 is often used for tasks that have been created but are not yet activated.

On OpenVMS Alpha systems, task IDs are assigned at activation. On GNAT systems, task IDs are also assigned at task creation but do not have the same form or values as task ID values in DEC Ada. There is no null task, and the environment task does not have a specific task ID value.

22.7.2 Task IDs and Delays

On OpenVMS Alpha systems, tasking delays are implemented using Timer System Services. The Task ID is used for the identification of the timer request (the REQIDT parameter). If Timers are used in the application take care not to use 0 for the identification, because cancelling such a timer will cancel all timers and may lead to unpredictable results.

22.7.3 Task-Related Pragmas

Ada supplies the pragma TASK_STORAGE, which allows specification of the size of the guard area for a task stack. (The guard area forms an area of memory that has no read or write access and thus helps in the detection of stack overflow.) On OpenVMS Alpha systems, if the pragma TASK_STORAGE specifies a value of zero, a minimal guard area is created. In the absence of a pragma TASK_STORAGE, a default guard area is created.

GNAT supplies the following task-related pragmas:

- TASK_INFO

This pragma appears within a task definition and applies to the task in which it appears. The argument must be of type SYSTEM.TASK_INFO.TASK_INFO_TYPE.

- **TASK_STORAGE**

GNAT implements pragma `TASK_STORAGE` in the same way as DEC Ada. Both DEC Ada and GNAT supply the pragmas `PASSIVE`, `SUPPRESS`, and `VOLATILE`.

22.7.4 Scheduling and Task Priority

DEC Ada implements the Ada language requirement that when two tasks are eligible for execution and they have different priorities, the lower priority task does not execute while the higher priority task is waiting. The DEC Ada Run-Time Library keeps a task running until either the task is suspended or a higher priority task becomes ready.

On OpenVMS Alpha systems, the default strategy is round-robin with preemption. Tasks of equal priority take turns at the processor. A task is run for a certain period of time and then placed at the rear of the ready queue for its priority level.

DEC Ada provides the implementation-defined pragma `TIME_SLICE`, which can be used to enable or disable round-robin scheduling of tasks with the same priority. See the relevant DEC Ada run-time reference manual for information on using the pragmas to control DEC Ada task scheduling.

GNAT follows the scheduling rules of Annex D (real-time Annex) of the Ada 95 Reference Manual. In general, this scheduling strategy is fully compatible with DEC Ada although it provides some additional constraints (as fully documented in Annex D). GNAT implements time slicing control in a manner compatible with DEC Ada 83, by means of the pragma `Time_Slice`, whose semantics are identical to the DEC Ada 83 pragma of the same name. Note that it is not possible to mix GNAT tasking and DEC Ada 83 tasking in the same program, since the two run times are not compatible.

22.7.5 The Task Stack

In DEC Ada, a task stack is allocated each time a non passive task is activated. As soon as the task is terminated, the storage for the task stack is deallocated. If you specify a size of zero (bytes) with `T'SORAGE_SIZE`, a default stack size is used. Also, regardless of the size specified, some additional space is allocated for task management purposes. On OpenVMS Alpha systems, at least one page is allocated.

GNAT handles task stacks in a similar manner. According to the Ada 95 rules, it provides the pragma `STORAGE_SIZE` as an alternative method for controlling the task stack size. The specification of the attribute `T'SORAGE_SIZE` is also supported in a manner compatible with DEC Ada.

22.7.6 External Interrupts

On DEC Ada, external interrupts can be associated with task entries. GNAT is compatible with DEC Ada in its handling of external interrupts.

22.8 Pragmas and Pragma-Related Features

Both DEC Ada and GNAT supply all language-defined pragmas as specified by the Ada 83 standard. GNAT also supplies all language-defined pragmas specified in the Ada 95 Reference Manual. In addition, GNAT implements the implementation-defined pragmas from DEC Ada 83.

- `AST_ENTRY`
- `COMMON_OBJECT`
- `COMPONENT_ALIGNMENT`
- `EXPORT_EXCEPTION`
- `EXPORT_FUNCTION`
- `EXPORT_OBJECT`
- `EXPORT_PROCEDURE`
- `EXPORT_VALUED_PROCEDURE`
- `FLOAT_REPRESENTATION`
- `IDENT`
- `IMPORT_EXCEPTION`
- `IMPORT_FUNCTION`
- `IMPORT_OBJECT`
- `IMPORT_PROCEDURE`
- `IMPORT_VALUED_PROCEDURE`
- `INLINE_GENERIC`
- `INTERFACE_NAME`
- `LONG_FLOAT`
- `MAIN_STORAGE`
- `PASSIVE`
- `PSET_OBJECT`
- `SHARE_GENERIC`
- `SUPPRESS_ALL`
- `TASK_STORAGE`
- `TIME_SLICE`
- `TITLE`

These pragmas are all fully implemented, with the exception of `Title`, `Passive`, and `Share_Generic`, which are recognized, but which have no effect in GNAT. The effect of `Passive` may be obtained by the use of protected objects in Ada 95. In GNAT, all generics are inlined.

Unlike DEC Ada, the GNAT '`EXPORT_subprogram`' pragmas require a separate subprogram specification which must appear before the subprogram body.

GNAT also supplies a number of implementation-defined pragmas as follows:

- `C_PASS_BY_COPY`

- `EXTEND_SYSTEM`
- `SOURCE_FILE_NAME`
- `UNSUPPRESS`
- `WARNINGS`
- `ABORT_DEFER`
- `ADA_83`
- `ADA_95`
- `ANNOTATE`
- `ASSERT`
- `CPP_CLASS`
- `CPP_CONSTRUCTOR`
- `CPP_DESTRUCTOR`
- `CPP_VIRTUAL`
- `CP_VTABLE`
- `DEBUG`
- `LINKER_ALIAS`
- `LINKER_SECTION`
- `MACHINE_ATTRIBUTE`
- `NO_RETURN`
- `PURE_FUNCTION`
- `SOURCE_REFERENCE`
- `TASK_INFO`
- `UNCHECKED_UNION`
- `UNIMPLEMENTED_UNIT`
- `WEAK_EXTERNAL`

For full details on these GNAT implementation-defined pragmas, see the GNAT Reference Manual.

22.8.1 Restrictions on the Pragma `INLINE`

DEC Ada applies the following restrictions to the pragma `INLINE`:

- Parameters cannot be a task type.
- Function results cannot be task types, unconstrained array types, or unconstrained types with discriminants.
- Bodies cannot declare the following:
 - Subprogram body or stub (imported subprogram is allowed)
 - Tasks
 - Generic declarations
 - Instantiations

- Exceptions
- Access types (types derived from access types allowed)
- Array or record types
- Dependent tasks
- Direct recursive calls of subprogram or containing subprogram, directly or via a renaming

In GNAT, the only restriction on pragma `INLINE` is that the body must occur before the call if both are in the same unit, and the size must be appropriately small. There are no other specific restrictions which cause subprograms to be incapable of being inlined.

22.8.2 Restrictions on the Pragma `INTERFACE`

The following lists and describes the restrictions on the pragma `INTERFACE` on DEC Ada and GNAT:

- Languages accepted: Ada, Bliss, C, Fortran, Default. Default is the default on OpenVMS Alpha systems.
- Parameter passing: Language specifies default mechanisms but can be overridden with an `EXPORT` pragma.
 - Ada: Use internal Ada rules.
 - Bliss, C: Parameters must be mode `in`; cannot be record or task type. Result cannot be a string, an array, or a record.
 - Fortran: Parameters cannot be a task. Result cannot be a string, an array, or a record.

GNAT is entirely upwards compatible with DEC Ada, and in addition allows record parameters for all languages.

22.8.3 Restrictions on the Pragma `SYSTEM_NAME`

For DEC Ada for OpenVMS Alpha, the enumeration literal for the type `NAME` is `OPENVMS_AXP`. In GNAT, the enumeration literal for the type `NAME` is `SYSTEM_NAME_GNAT`.

22.9 Library of Predefined Units

A library of predefined units is provided as part of the DEC Ada and GNAT implementations. DEC Ada does not provide the package `MACHINE_CODE` but instead recommends importing assembler code.

The GNAT versions of the DEC Ada Run-Time Library (`ADA$PREDEFINED:`) units are taken from the OpenVMS Alpha version, not the OpenVMS VAX version. During GNAT installation, the DEC Ada Predefined Library units are copied into the `GNU:[LIB.OPENVMS7_x.2_8_x.DECLIB]` (aka `DECLIB`) directory and patched to remove Ada 95 incompatibilities and to make them interoperable with GNAT, see Section 22.9.1 [Changes to `DECLIB`], page 233 for details.

The GNAT RTL is contained in the GNU:[LIB.OPENVMS7_x.2.8_x.ADALIB] (aka ADALIB) directory and the default search path is set up to find DECLIB units in preference to ADALIB units with the same name (TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO, for example).

However, it is possible to change the default so that the reverse is true, or even to mix them using child package notation. The DEC Ada 83 units are available as DEC.xxx where xxx is the package name, and the Ada units are available in the standard manner defined for Ada 95, that is to say as Ada.xxx. To change the default, set ADA_INCLUDE_PATH and ADA_OBJECTS_PATH appropriately. For example, to change the default to use the Ada95 versions do:

```
$ DEFINE ADA_INCLUDE_PATH GNU:[LIB.OPENVMS7_1.2.8_1.ADAINCLUDE],-
                                GNU:[LIB.OPENVMS7_1.2.8_1.DECLIB]
$ DEFINE ADA_OBJECTS_PATH GNU:[LIB.OPENVMS7_1.2.8_1.ADALIB],-
                                GNU:[LIB.OPENVMS7_1.2.8_1.DECLIB]
```

22.9.1 Changes to DECLIB

The changes made to the DEC Ada predefined library for GNAT and Ada 95 compatibility are minor and include the following:

- Adjusting the location of pragmas and record representation clauses to obey Ada 95 rules
- Adding the proper notation to generic formal parameters that take unconstrained types in instantiation
- Adding pragma ELABORATE_BODY to package specifications that have package bodies not otherwise allowed
- Occurrences of the identifier "PROTECTED" are renamed to "PROTECTD". Currently these are found only in the STARLET package spec.

None of the above changes is visible to users.

22.10 Bindings

On OpenVMS Alpha, DEC Ada provides the following strongly-typed bindings:

- Command Language Interpreter (CLI interface)
- DECtalk Run-Time Library (DTK interface)
- Librarian utility routines (LBR interface)
- General Purpose Run-Time Library (LIB interface)
- Math Run-Time Library (MTH interface)
- National Character Set Run-Time Library (NCS interface)
- Compiled Code Support Run-Time Library (OTS interface)
- Parallel Processing Run-Time Library (PPL interface)
- Screen Management Run-Time Library (SMG interface)
- Sort Run-Time Library (SOR interface)
- String Run-Time Library (STR interface)

- STARLET System Library
- X Window System Version 11R4 and 11R5 (X, XLIB interface)
- X Windows Toolkit (XT interface)
- X/Motif Version 1.1.3 and 1.2 (XM interface)

GNAT provides implementations of these DEC bindings in the DECLIB directory.

The X/Motif bindings used to build DECLIB are whatever versions are in the DEC Ada ADA\$PREDEFINED directory with extension .ADC. The build script will automatically add a pragma Linker_Options to packages Xm, Xt, and X.Lib causing the default X/Motif shareable image libraries to be linked in. This is done via options files named xm.opt, xt.opt, and x_lib.opt (also located in the DECLIB directory).

It may be necessary to edit these options files to update or correct the library names if, for example, the newer X/Motif bindings from ADA\$EXAMPLES had been (previous to installing GNAT) copied and renamed to superseded the default ADA\$PREDEFINED versions.

22.10.1 Shared Libraries and Options Files

When using the DEC Ada predefined X and Motif bindings, the linking with their shareable images is done automatically by GNAT LINK. When using other X and Motif bindings, it is necessary to add the corresponding shareable images to the command line for GNAT LINK. When linking with shared libraries, or with .OPT files, it is also necessary to add them to the command line for GNAT LINK.

A shared library to be used with GNAT is built in the same way as other libraries under VMS. The VMS Link command can be used in standard fashion.

22.10.2 Interfaces to C

DEC Ada provides the following Ada types and operations:

- C types package (C_TYPES)
- C strings (C_TYPES.NULL_TERMINATED)
- Other_types (SHORT_INT)

Interfacing to C with GNAT, one can use the above approach described for DEC Ada or the facilities of Annex B of the Ada 95 Reference Manual (packages INTERFACES.C, INTERFACES.C.STRINGS and INTERFACES.C.POINTERS). For more information, see the section "Interfacing to C" in the GNAT Reference Manual.

The `'/UPPERCASE_EXTERNALS'` qualifier forces default and explicit `External_Name` parameters in pragmas Import and Export to be uppercased for compatibility with the default behavior of DEC C. The qualifier has no effect on `Link_Name` parameters.

22.11 Main Program Definition

The following section discusses differences in the definition of main programs on DEC Ada and GNAT. On DEC Ada, main programs are defined to meet the following conditions:

- Procedure with no formal parameters (returns 0 upon normal completion)
- Procedure with no formal parameters (returns 42 when unhandled exceptions are raised)
- Function with no formal parameters whose returned value is of a discrete type
- Procedure with one OUT formal of a discrete type for which a specification of pragma `EXPORT_VALUED_PROCEDURE` is given.

When declared with the pragma `EXPORT_VALUED_PROCEDURE`, a main function or main procedure returns a discrete value whose size is less than 64 bits (32 on VAX systems), the value is zero- or sign-extended as appropriate. On GNAT, main programs are defined as follows:

- Must be a non-generic, parameter-less subprogram that is either a procedure or function returning an Ada `STANDARD.INTEGER` (the predefined type)
- Cannot be a generic subprogram or an instantiation of a generic subprogram

22.12 Implementation-Defined Attributes

GNAT provides all DEC Ada implementation-defined attributes.

22.13 Compiler and Run-Time Interfacing

DEC Ada provides the following ways to pass options to the linker (ACS LINK):

- `/WAIT` and `/SUBMIT` qualifiers
- `/COMMAND` qualifier
- `/[NO]MAP` qualifier
- `/OUTPUT=file-spec`
- `/[NO]DEBUG` and `/[NO]TRACEBACK` qualifiers

To pass options to the linker, GNAT provides the following qualifiers:

- `/EXECUTABLE=exec-name`
- `/VERBOSE` qualifier
- `/[NO]DEBUG` and `/[NO]TRACEBACK` qualifiers

For more information on these qualifiers, see the section "Qualifiers for GNAT LINK" in the corresponding section of this Guide. In DEC Ada, the command-line qualifier `/OPTIMIZE` is available to control optimization. DEC Ada also supplies the following pragmas:

- `OPTIMIZE`
- `INLINE`
- `INLINE_GENERIC`

- SUPPRESS_ALL
- PASSIVE

In GNAT, optimization is controlled strictly by command line parameters, as described in the corresponding section of this guide. The DIGITAL pragmas for control of optimization are recognized but ignored.

Note that in GNAT, the default is optimization off, whereas in DEC Ada 83, the default is that optimization is turned on.

22.14 Program Compilation and Library Management

DEC Ada and GNAT provide a comparable set of commands to build programs. DEC Ada also provides a program library, which is a concept that does not exist on GNAT. Instead, GNAT provides directories of sources that are compiled as needed.

The following table summarizes the DEC Ada commands and provides equivalent GNAT commands. In this table, some GNAT equivalents reflect the fact that GNAT does not use the concept of a program library. Instead, it uses a model in which collections of source and object files are used in a manner consistent with other languages like C and Fortran. Therefore, standard system file commands are used to manipulate these elements. Those GNAT commands are marked with an asterisk in the table that follows. Note that, unlike DEC Ada, none of the GNAT commands accepts wild cards.

DEC_Ada_Command	GNAT_Equivalent	Description
ADA	GNAT COMPILE	Invokes the compiler to compile one or more Ada source files.
ACS ATTACH	No equivalent	Qualifiers control of terminal from current process running the program library manager.
ACS CHECK	GNAT MAKE /DEPENDENCY_LIST	Forms the execution closure of one or more compiled units and checks completeness and currency.
ACS COMPILE	GNAT MAKE /ACTIONS=COMPILE	Forms the execution closure of one or more specified units, checks completeness and currency, identifies units that have revised source files, compiles same, and recompiles units that are or will become obsolete. Also completes incomplete generic instantiations.

ACS COPY FOREIGN	Copy (*)	Copies a foreign object file into the program library as a library unit body.
ACS COPY UNIT	Copy (*)	Copies a compiled unit from one program library to another.
ACS CREATE LIBRARY	Create /directory (*)	Creates a program library.
ACS SUBLIBRARY ACS DELETE LIBRARY	Create /directory (*)	Creates a program sublibrary.
ACS SUBLIBRARY	DELETE	Deletes a program library and its contents.
ACS DELETE UNIT	Delete <i>file</i> (*)	Deletes a program sublibrary and its contents.
ACS DIRECTORY	Directory (*)	On OpenVMS systems, deletes one or more compiled units from the current program library.
ACS ENTER FOREIGN	Copy (*)	On OpenVMS systems, lists units contained in the current program library.
ACS ENTER UNIT	Copy (*)	Allows the import of a foreign body as an Ada library specification and enters a reference to a pointer.
ACS EXIT	No equivalent	Enters a reference (pointer) from the current program library to a unit compiled into another program library.
ACS EXPORT	Copy (*)	Exits from the program library manager.
ACS EXTRACT SOURCE	Copy (*)	Creates an object file that contains system-specific object code for one or more units. With GNAT, object files can simply be copied into the desired directory.
ACS HELP	HELP GNAT	Allows access to the copied source file for each Ada compilation unit
ACS LINK	GNAT LINK	Provides online help.
		Links an object file containing Ada units into an executable file.

ACS LOAD	Copy (*)	Loads (partially compiles) Ada units into the program library. Allows loading a program from a collection of files into a library without knowing the relationship among units.
ACS MERGE	Copy (*)	Merges into the current program library, one or more units from another library where they were modified.
ACS RECOMPILE	GNAT MAKE /ACTIONS=COMPILE	Recompiles from external or copied source files any obsolete unit in the closure. Also, completes any incomplete generic instantiations.
ACS REENTER	GNAT MAKE	Reenters current references to units compiled after last entered with the ACS ENTER UNIT command.
ACS SET LIBRARY	Set default (*)	Defines a program library to be the compilation context as well as the target library for compiler output and commands in general.
ACS SET PRAGMA	Edit GNAT.ADC (*)	Redefines specified values of the library characteristics LONG_FLOAT, MEMORY_SIZE, SYSTEM_NAME, and Float_Representation.
ACS SET SOURCE	define ADA_INCLUDE_PATH <i>path</i> (*)	Defines the source file search list for the ACS COMPILE command.
ACS SHOW LIBRARY	Directory (*)	Lists information about one or more program libraries.
ACS SHOW PROGRAM	No equivalent	Lists information about the execution closure of one or more units in the program library.
ACS SHOW SOURCE	Show logical ADA_INCLUDE_PATH	Shows the source file search used when compiling units.

ACS SHOW VERSION	Compile VERBOSE option	with	Displays the version number of the compiler and program library manager used.
ACS SPAWN	No equivalent		Creates a subprocess of the current process (same as DCL SPAWN command).
ACS VERIFY	No equivalent		Performs a series of consistency checks on a program library to determine whether the library structure and library files are in valid form.

22.15 Input-Output

On OpenVMS Alpha systems, DEC Ada uses OpenVMS Record Management Services (RMS) to perform operations on external files.

DEC Ada and GNAT predefine an identical set of input- output packages. To make the use of the generic TEXT_IO operations more convenient, DEC Ada provides predefined library packages that instantiate the integer and floating-point operations for the predefined integer and floating-point types as shown in the following table.

Package_Name	Instantiation
INTEGER_TEXT_IO	INTEGER_IO(INTEGER)
SHORT_INTEGER_TEXT_IO	INTEGER_IO(SHORT_INTEGER)
SHORT_SHORT_INTEGER_TEXT_IO	INTEGER_IO(SHORT_SHORT_INTEGER)
FLOAT_TEXT_IO	FLOAT_IO(FLOAT)
LONG_FLOAT_TEXT_IO	FLOAT_IO(LONG_FLOAT)

The DEC Ada predefined packages and their operations are implemented using OpenVMS Alpha files and input- output facilities. DEC Ada supports asynchronous input- output on OpenVMS Alpha. Familiarity with the following is recommended:

- RMS file organizations and access methods
- OpenVMS file specifications and directories
- OpenVMS File Definition Language (FDL)

GNAT provides I/O facilities that are completely compatible with DEC Ada. The distribution includes the standard DEC Ada versions of all I/O packages, operating in a manner

compatible with DEC Ada. In particular, the following packages are by default the DEC Ada (Ada 83) versions of these packages rather than the renamings suggested in annex J of the Ada 95 Reference Manual:

- TEXT_IO
- SEQUENTIAL_IO
- DIRECT_IO

The use of the standard Ada 95 syntax for child packages (for example, ADA.TEXT_IO) retrieves the Ada 95 versions of these packages, as defined in the Ada 95 Reference Manual. GNAT provides DIGITAL-compatible predefined instantiations of the TEXT_IO packages, and also provides the standard predefined instantiations required by the Ada 95 Reference Manual.

For further information on how GNAT interfaces to the file system or how I/O is implemented in programs written in mixed languages, see the chapter "Implementation of the Standard I/O" in the GNAT Reference Manual. This chapter covers the following:

- Standard I/O packages
- FORM strings
- DIRECT_IO
- SEQUENTIAL_IO
- TEXT_IO
- Stream pointer positioning
- Reading and writing non-regular files
- GET_IMMEDIATE
- Treating TEXT_IO files as streams
- Shared files
- Open modes

22.16 Implementation Limits

The following table lists implementation limits for DEC Ada and GNAT systems.

Compilation Parameter	DEC Ada	GNAT
In a subprogram or entry declaration, maximum number of formal parameters that are of an unconstrained record type	32	No set limit
Maximum identifier length (number of characters)	255	255
Maximum number of characters in a source line	255	255
Maximum collection size (number of bytes)	2**31-1	2**31-1
Maximum number of discriminants for a record type	245	No set limit
Maximum number of formal parameters in an entry or subprogram declaration	246	No set limit

Maximum number of dimensions in an array type	255	No set limit
Maximum number of library units and subunits in a compilation.	4095	No set limit
Maximum number of library units and subunits in an execution.	16383	No set limit
Maximum number of objects declared with the pragma COMMON_OBJECT or PSECT_OBJECT	32757	No set limit
Maximum number of enumeration literals in an enumeration type definition	65535	No set limit
Maximum number of lines in a source file	65534	No set limit
Maximum number of bits in any object	2**31-1	2**31-1
Maximum size of the static portion of a stack frame (approximate)	2**31-1	2**31-1

22.17 Tools

23 Inline Assembler

If you need to write low-level software that interacts directly with the hardware, Ada provides two ways to incorporate assembly language code into your program. First, you can import and invoke external routines written in assembly language, an Ada feature fully supported by GNAT. However, for small sections of code it may be simpler or more efficient to include assembly language statements directly in your Ada source program, using the facilities of the implementation-defined package `System.Machine_Code`, which incorporates the GNAT COMPILE Inline Assembler. The Inline Assembler approach offers a number of advantages, including the following:

- No need to use non-Ada tools
- Consistent interface over different targets
- Automatic usage of the proper calling conventions
- Access to Ada constants and variables
- Definition of intrinsic routines
- Possibility of inlining a subprogram comprising assembler code
- Code optimizer can take Inline Assembler code into account

This chapter presents a series of examples to show you how to use the Inline Assembler. Although it focuses on the Intel x86, the general approach applies also to other processors. It is assumed that you are familiar with Ada and with assembly language programming.

23.1 Basic Assembler Syntax

The assembler used by GNAT and GNAT COMPILE is based not on the Intel assembly language, but rather on a language that descends from the AT&T Unix assembler *as* (and which is often referred to as “AT&T syntax”). The following table summarizes the main features of *as* syntax and points out the differences from the Intel conventions. See the GNAT COMPILE *as* and *gas* (an *as* macro pre-processor) documentation for further information.

Register names

GNAT COMPILE / *as*: Prefix with “%”; for example `%eax`
 Intel: No extra punctuation; for example `eax`

Immediate operand

GNAT COMPILE / *as*: Prefix with “\$”; for example `$4`
 Intel: No extra punctuation; for example `4`

Address

GNAT COMPILE / *as*: Prefix with “\$”; for example `$loc`
 Intel: No extra punctuation; for example `loc`

Memory contents

GNAT COMPILE / *as*: No extra punctuation; for example `loc`
 Intel: Square brackets; for example `[loc]`

Register contents

GNAT COMPILE / *as*: Parentheses; for example `(%eax)`
 Intel: Square brackets; for example `[eax]`

Hexadecimal numbers

GNAT COMPILE / *as*: Leading “0x” (C language syntax); for example 0xA0
 Intel: Trailing “h”; for example A0h

Operand size

GNAT COMPILE / *as*: Explicit in op code; for example `movw` to move a 16-bit word
 Intel: Implicit, deduced by assembler; for example `mov`

Instruction repetition

GNAT COMPILE / *as*: Split into two lines; for example
`rep`
`stosl`
 Intel: Keep on one line; for example `rep stosl`

Order of operands

GNAT COMPILE / *as*: Source first; for example `movw $4, %eax`
 Intel: Destination first; for example `mov eax, 4`

23.2 A Simple Example of Inline Assembler

The following example will generate a single assembly language statement, `nop`, which does nothing. Despite its lack of run-time effect, the example will be useful in illustrating the basics of the Inline Assembler facility.

```
with System.Machine_Code; use System.Machine_Code;
procedure Nothing is
begin
  Asm ("nop");
end Nothing;
```

`Asm` is a procedure declared in package `System.Machine_Code`; here it takes one parameter, a *template string* that must be a static expression and that will form the generated instruction. `Asm` may be regarded as a compile-time procedure that parses the template string and additional parameters (none here), from which it generates a sequence of assembly language instructions.

The examples in this chapter will illustrate several of the forms for invoking `Asm`; a complete specification of the syntax is found in the *GNAT Reference Manual*.

Under the standard GNAT conventions, the `Nothing` procedure should be in a file named `'NOTHING.ADB'`. You can build the executable in the usual way:

```
GNAT MAKE nothing
```

However, the interesting aspect of this example is not its run-time behavior but rather the generated assembly code. To see this output, invoke the compiler as follows:

```
GNAT COMPILE -S -fomit-frame-pointer /CHECKS=SUPPRESS_ALL 'NOTHING.ADB'
```

where the options are:

- c compile only (no bind or link)
- S generate assembler listing
- fomit-frame-pointer
 do not set up separate stack frames

```
/CHECKS=SUPPRESS_ALL
    do not add runtime checks
```

This gives a human-readable assembler version of the code. The resulting file will have the same name as the Ada source file, but with a `.s` extension. In our example, the file `'nothing.s'` has the following contents:

```
.file "NOTHING.ADB"
gcc2_compiled.:
___gnu_compiled_ada:
.text
    .align 4
.globl __ada_nothing
__ada_nothing:
#APP
    nop
#NO_APP
    jmp L1
    .align 2,0x90
L1:
    ret
```

The assembly code you included is clearly indicated by the compiler, between the `#APP` and `#NO_APP` delimiters. The character before the `'APP'` and `'NOAPP'` can differ on different targets. For example, Linux uses `'#APP'` while on NT you will see `'/APP'`.

If you make a mistake in your assembler code (such as using the wrong size modifier, or using a wrong operand for the instruction) GNAT will report this error in a temporary file, which will be deleted when the compilation is finished. Generating an assembler file will help in such cases, since you can assemble this file separately using the `as` assembler that comes with GNAT COMPILER.

Assembling the file using the command

```
as 'nothing.s'
```

will give you error messages whose lines correspond to the assembler input file, so you can easily find and correct any mistakes you made. If there are no errors, `as` will generate an object file `'nothing.out'`.

23.3 Output Variables in Inline Assembler

The examples in this section, showing how to access the processor flags, illustrate how to specify the destination operands for assembly language statements.

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags is
    Flags : Unsigned_32;
    use ASCII;
begin
    Asm ("pushfl"           & LF & HT & -- push flags on stack
        "popl %%eax"       & LF & HT & -- load eax with flags
        "movl %%eax, %0",   -- store flags in variable
        Outputs => Unsigned_32'Asm_Output ("=g", Flags));
    Put_Line ("Flags register:" & Flags'Img);
end Get_Flags;
```

In order to have a nicely aligned assembly listing, we have separated multiple assembler statements in the Asm template string with linefeed (ASCII.LF) and horizontal tab (ASCII.HT) characters. The resulting section of the assembly output file is:

```
#APP
    pushfl
    popl %eax
    movl %eax, -40(%ebp)
#NO_APP
```

It would have been legal to write the Asm invocation as:

```
Asm ("pushfl popl %%eax movl %%eax, %0")
```

but in the generated assembler file, this would come out as:

```
#APP
    pushfl popl %eax movl %eax, -40(%ebp)
#NO_APP
```

which is not so convenient for the human reader.

We use Ada comments at the end of each line to explain what the assembler instructions actually do. This is a useful convention.

When writing Inline Assembler instructions, you need to precede each register and variable name with a percent sign. Since the assembler already requires a percent sign at the beginning of a register name, you need two consecutive percent signs for such names in the Asm template string, thus `%%eax`. In the generated assembly code, one of the percent signs will be stripped off.

Names such as `%0`, `%1`, `%2`, etc., denote input or output variables: operands you later define using `Input` or `Output` parameters to `Asm`. An output variable is illustrated in the third statement in the Asm template string:

```
movl %%eax, %0
```

The intent is to store the contents of the `eax` register in a variable that can be accessed in Ada. Simply writing `movl %%eax, Flags` would not necessarily work, since the compiler might optimize by using a register to hold `Flags`, and the expansion of the `movl` instruction would not be aware of this optimization. The solution is not to store the result directly but rather to advise the compiler to choose the correct operand form; that is the purpose of the `%0` output variable.

Information about the output variable is supplied in the `Outputs` parameter to `Asm`:

```
Outputs => Unsigned_32'Asm_Output ("=g", Flags);
```

The output is defined by the `Asm_Output` attribute of the target type; the general format is

```
Type'Asm_Output (constraint_string, variable_name)
```

The constraint string directs the compiler how to store/access the associated variable. In the example

```
Unsigned_32'Asm_Output ("=m", Flags);
```

the `"m"` (memory) constraint tells the compiler that the variable `Flags` should be stored in a memory variable, thus preventing the optimizer from keeping it in a register. In contrast,

```
Unsigned_32'Asm_Output ("=r", Flags);
```

uses the `"r"` (register) constraint, telling the compiler to store the variable in a register.

If the constraint is preceded by the equal character (`=`), it tells the compiler that the variable will be used to store data into it.

In the `Get_Flags` example, we used the "g" (global) constraint, allowing the optimizer to choose whatever it deems best.

There are a fairly large number of constraints, but the ones that are most useful (for the Intel x86 processor) are the following:

=	output constraint
g	global (i.e. can be stored anywhere)
m	in memory
I	a constant
a	use <code>eax</code>
b	use <code>ebx</code>
c	use <code>ecx</code>
d	use <code>edx</code>
S	use <code>esi</code>
D	use <code>edi</code>
r	use one of <code>eax</code> , <code>ebx</code> , <code>ecx</code> or <code>edx</code>
q	use one of <code>eax</code> , <code>ebx</code> , <code>ecx</code> , <code>edx</code> , <code>esi</code> or <code>edi</code>

The full set of constraints is described in the GNAT COMPILE and *as* documentation; note that it is possible to combine certain constraints in one constraint string.

You specify the association of an output variable with an assembler operand through the `%n` notation, where *n* is a non-negative integer. Thus in

```
Asm ("pushfl"           & LF & HT & -- push flags on stack
     "popl %%eax"       & LF & HT & -- load eax with flags
     "movl %%eax, %0",   -- store flags in variable
     Outputs => Unsigned_32'Asm_Output ("=g", Flags));
```

`%0` will be replaced in the expanded code by the appropriate operand, whatever the compiler decided for the `Flags` variable.

In general, you may have any number of output variables:

- Count the operands starting at 0; thus `%0`, `%1`, etc.
- Specify the `Outputs` parameter as a parenthesized comma-separated list of `Asm_Output` attributes

For example:

```
Asm ("movl %%eax, %0" & LF & HT &
     "movl %%ebx, %1" & LF & HT &
     "movl %%ecx, %2",
     Outputs => (Unsigned_32'Asm_Output ("=g", Var_A), -- %0 = Var_A
                 Unsigned_32'Asm_Output ("=g", Var_B), -- %1 = Var_B
                 Unsigned_32'Asm_Output ("=g", Var_C))); -- %2 = Var_C
```

where `Var_A`, `Var_B`, and `Var_C` are variables in the Ada program.

As a variation on the `Get_Flags` example, we can use the constraints string to direct the compiler to store the `eax` register into the `Flags` variable, instead of including the store instruction explicitly in the `Asm` template string:

```

with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags_2 is
  Flags : Unsigned_32;
  use ASCII;
begin
  Asm ("pushfl"      & LF & HT & -- push flags on stack
      "popl %%eax",  -- save flags in eax
      Outputs => Unsigned_32'Asm_Output ("=a", Flags));
  Put_Line ("Flags register:" & Flags'Img);
end Get_Flags_2;

```

The "a" constraint tells the compiler that the `Flags` variable will come from the `eax` register. Here is the resulting code:

```

#APP
  pushfl
  popl %eax
#NO_APP
  movl %eax, -40(%ebp)

```

The compiler generated the store of `eax` into `Flags` after expanding the assembler code.

Actually, there was no need to pop the flags into the `eax` register; more simply, we could just pop the flags directly into the program variable:

```

with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Get_Flags_3 is
  Flags : Unsigned_32;
  use ASCII;
begin
  Asm ("pushfl" & LF & HT & -- push flags on stack
      "pop %0",  -- save flags in Flags
      Outputs => Unsigned_32'Asm_Output ("=g", Flags));
  Put_Line ("Flags register:" & Flags'Img);
end Get_Flags_3;

```

23.4 Input Variables in Inline Assembler

The example in this section illustrates how to specify the source operands for assembly language statements. The program simply increments its input value by 1:

```

with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Increment is

  function Incr (Value : Unsigned_32) return Unsigned_32 is
    Result : Unsigned_32;
  begin
    Asm ("incl %0",
        Inputs => Unsigned_32'Asm_Input ("a", Value),
        Outputs => Unsigned_32'Asm_Output ("=a", Result));
    return Result;
  end Incr;

  Value : Unsigned_32;

begin
  Value := 5;
  Put_Line ("Value before is" & Value'Img);
  Value := Incr (Value);
  Put_Line ("Value after is" & Value'Img);
end Increment;

```

The **Outputs** parameter to **Asm** specifies that the result will be in the `eax` register and that it is to be stored in the **Result** variable.

The **Inputs** parameter looks much like the **Outputs** parameter, but with an **Asm_Input** attribute. The `"="` constraint, indicating an output value, is not present.

You can have multiple input variables, in the same way that you can have more than one output variable.

The parameter count (`%0`, `%1`) etc, now starts at the first input statement, and continues with the output statements. When both parameters use the same variable, the compiler will treat them as the same `%n` operand, which is the case here.

Just as the **Outputs** parameter causes the register to be stored into the target variable after execution of the assembler statements, so does the **Inputs** parameter cause its variable to be loaded into the register before execution of the assembler statements.

Thus the effect of the **Asm** invocation is:

1. load the 32-bit value of **Value** into `eax`
2. execute the `incl %eax` instruction
3. store the contents of `eax` into the **Result** variable

The resulting assembler file (with `/OPTIMIZE=ALL` optimization) contains:

```

_increment__incr.1:
    subl $4,%esp
    movl 8(%esp),%eax
#APP
    incl %eax
#NO_APP
    movl %eax,%edx
    movl %ecx, (%esp)
    addl $4,%esp
    ret

```

23.5 Inlining Inline Assembler Code

For a short subprogram such as the `Incr` function in the previous section, the overhead of the call and return (creating / deleting the stack frame) can be significant, compared to the amount of code in the subprogram body. A solution is to apply Ada's `Inline` pragma to the subprogram, which directs the compiler to expand invocations of the subprogram at the point(s) of call, instead of setting up a stack frame for out-of-line calls. Here is the resulting program:

```
with Interfaces; use Interfaces;
with Ada.Text_IO; use Ada.Text_IO;
with System.Machine_Code; use System.Machine_Code;
procedure Increment_2 is

  function Incr (Value : Unsigned_32) return Unsigned_32 is
    Result : Unsigned_32;
  begin
    Asm ("incl %0",
        Inputs => Unsigned_32'Asm_Input ("a", Value),
        Outputs => Unsigned_32'Asm_Output ("=a", Result));
    return Result;
  end Incr;
  pragma Inline (Increment);

  Value : Unsigned_32;

begin
  Value := 5;
  Put_Line ("Value before is" & Value'Img);
  Value := Increment (Value);
  Put_Line ("Value after is" & Value'Img);
end Increment_2;
```

Compile the program with both optimization (`/OPTIMIZE=ALL`) and inlining enabled (`'-gnatpn'` instead of `'/CHECKS=SUPPRESS_ALL'`).

The `Incr` function is still compiled as usual, but at the point in `Increment` where our function used to be called:

```
pushl %edi
call _increment__incr.1
```

the code for the function body directly appears:

```
movl %esi,%eax
#APP
  incl %eax
#NO_APP
movl %eax,%edx
```

thus saving the overhead of stack frame setup and an out-of-line call.

23.6 Other Asm Functionality

This section describes two important parameters to the `Asm` procedure: `Clobber`, which identifies register usage; and `Volatile`, which inhibits unwanted optimizations.

23.6.1 The Clobber Parameter

One of the dangers of intermixing assembly language and a compiled language such as Ada is that the compiler needs to be aware of which registers are being used by the assembly code. In some cases, such as the earlier examples, the constraint string is sufficient to indicate register usage (e.g. "a" for the `eax` register). But more generally, the compiler needs an explicit identification of the registers that are used by the Inline Assembly statements.

Using a register that the compiler doesn't know about could be a side effect of an instruction (like `mull` storing its result in both `eax` and `edx`). It can also arise from explicit register usage in your assembly code; for example:

```
Asm ("movl %0, %%ebx" & LF & HT &
    "movl %%ebx, %1",
    Inputs => Unsigned_32'Asm_Input ("g", Var_In),
    Outputs => Unsigned_32'Asm_Output ("=g", Var_Out));
```

where the compiler (since it does not analyze the `Asm` template string) does not know you are using the `ebx` register.

In such cases you need to supply the `Clobber` parameter to `Asm`, to identify the registers that will be used by your assembly code:

```
Asm ("movl %0, %%ebx" & LF & HT &
    "movl %%ebx, %1",
    Inputs => Unsigned_32'Asm_Input ("g", Var_In),
    Outputs => Unsigned_32'Asm_Output ("=g", Var_Out),
    Clobber => "ebx");
```

The `Clobber` parameter is a static string expression specifying the register(s) you are using. Note that register names are *not* prefixed by a percent sign. Also, if more than one register is used then their names are separated by commas; e.g., "`eax, ebx`".

The `Clobber` parameter has several additional uses:

1. Use the "register" name `cc` to indicate that flags might have changed
2. Use the "register" name `memory` if you changed a memory location

23.6.2 The Volatile Parameter

Compiler optimizations in the presence of Inline Assembler may sometimes have unwanted effects. For example, when an `Asm` invocation with an input variable is inside a loop, the compiler might move the loading of the input variable outside the loop, regarding it as a one-time initialization.

If this effect is not desired, you can disable such optimizations by setting the `Volatile` parameter to `True`; for example:

```
Asm ("movl %0, %%ebx" & LF & HT &
    "movl %%ebx, %1",
    Inputs => Unsigned_32'Asm_Input ("g", Var_In),
    Outputs => Unsigned_32'Asm_Output ("=g", Var_Out),
    Clobber => "ebx",
    Volatile => True);
```

By default, `Volatile` is set to `False` unless there is no `Outputs` parameter.

Although setting `Volatile` to `True` prevents unwanted optimizations, it will also disable other optimizations that might be important for efficiency. In general, you should set `Volatile` to `True` only if the compiler's optimizations have created problems.

23.7 A Complete Example

This section contains a complete program illustrating a realistic usage of GNAT's Inline Assembler capabilities. It comprises a main procedure `Check_CPU` and a package `Intel_CPU`. The package declares a collection of functions that detect the properties of the 32-bit x86 processor that is running the program. The main procedure invokes these functions and displays the information.

The `Intel_CPU` package could be enhanced by adding functions to detect the type of x386 co-processor, the processor caching options and special operations such as the SIMD extensions.

Although the `Intel_CPU` package has been written for 32-bit Intel compatible CPUs, it is OS neutral. It has been tested on DOS, Windows/NT and Linux.

23.7.1 Check_CPU Procedure

```

-----
--
-- Uses the Intel_CPU package to identify the CPU the program is --
-- running on, and some of the features it supports.             --
--
-----

with Intel_CPU;           -- Intel CPU detection functions
with Ada.Text_IO;         -- Standard text I/O
with Ada.Command_Line;    -- To set the exit status

procedure Check_CPU is

  Type_Found : Boolean := False;
  -- Flag to indicate that processor was identified

  Features   : Intel_CPU.Processor_Features;
  -- The processor features

  Signature  : Intel_CPU.Processor_Signature;
  -- The processor type signature

begin

  -----
  -- Display the program banner. --
  -----

  Ada.Text_IO.Put_Line (Ada.Command_Line.Command_Name &
                        ": check Intel CPU version and features, v1.0");
  Ada.Text_IO.Put_Line ("distribute freely, but no warranty whatsoever");
  Ada.Text_IO.New_Line;

  -----
  -- We can safely start with the assumption that we are on at least --
  -- a x386 processor. If the CPUID instruction is present, then we --
  -- have a later processor type.                                     --
  -----

```

```

if Intel_CPU.Has_CPUID = False then

    -- No CPUID instruction, so we assume this is indeed a x386
    -- processor. We can still check if it has a FP co-processor.
    if Intel_CPU.Has_FPU then
        Ada.Text_IO.Put_Line
            ("x386-type processor with a FP co-processor");
    else
        Ada.Text_IO.Put_Line
            ("x386-type processor without a FP co-processor");
    end if; -- check for FPU

    -- Program done
    Ada.Command_Line.Set_Exit_Status (Ada.Command_Line.Success);
    return;

end if; -- check for CPUID

-----
-- If CPUID is supported, check if this is a true Intel processor, --
-- if it is not, display a warning.                                --
-----

if Intel_CPU.Vendor_ID /= Intel_CPU.Intel_Processor then
    Ada.Text_IO.Put_Line ("*** This is a Intel compatible processor");
    Ada.Text_IO.Put_Line ("*** Some information may be incorrect");
end if; -- check if Intel

-----
-- With the CPUID instruction present, we can assume at least a --
-- x486 processor. If the CPUID support level is < 1 then we have --
-- to leave it at that.                                           --
-----

if Intel_CPU.CPUID_Level < 1 then

    -- Ok, this is a x486 processor. we still can get the Vendor ID
    Ada.Text_IO.Put_Line ("x486-type processor");
    Ada.Text_IO.Put_Line ("Vendor ID is " & Intel_CPU.Vendor_ID);

    -- We can also check if there is a FPU present
    if Intel_CPU.Has_FPU then
        Ada.Text_IO.Put_Line ("Floating-Point support");
    else
        Ada.Text_IO.Put_Line ("No Floating-Point support");
    end if; -- check for FPU

    -- Program done
    Ada.Command_Line.Set_Exit_Status (Ada.Command_Line.Success);
    return;

end if; -- check CPUID level

-----
-- With a CPUID level of 1 we can use the processor signature to --
-- determine it's exact type.                                    --
-----

```

```

Signature := Intel_CPU.Signature;

-----
-- Ok, now we go into a lot of messy comparisons to get the      --
-- processor type. For clarity, no attempt to try to optimize the --
-- comparisons has been made. Note that since Intel_CPU does not --
-- support getting cache info, we cannot distinguish between P5   --
-- and Celeron types yet.                                         --
-----

-- x486SL
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0100# and
   Signature.Model          = 2#0100# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("x486SL processor");
end if;

-- x486DX2 Write-Back
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0100# and
   Signature.Model          = 2#0111# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("Write-Back Enhanced x486DX2 processor");
end if;

-- x486DX4
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0100# and
   Signature.Model          = 2#1000# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("x486DX4 processor");
end if;

-- x486DX4 Overdrive
if Signature.Processor_Type = 2#01# and
   Signature.Family         = 2#0100# and
   Signature.Model          = 2#1000# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("x486DX4 OverDrive processor");
end if;

-- Pentium (60, 66)
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0001# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("Pentium processor (60, 66)");
end if;

-- Pentium (75, 90, 100, 120, 133, 150, 166, 200)
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0010# then
   Type_Found := True;
   Ada.Text_IO.Put_Line
     ("Pentium processor (75, 90, 100, 120, 133, 150, 166, 200)");
end if;

```



```

-- Pentium OverDrive (60, 66)
if Signature.Processor_Type = 2#01# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0001# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("Pentium OverDrive processor (60, 66)");
end if;

-- Pentium OverDrive (75, 90, 100, 120, 133, 150, 166, 200)
if Signature.Processor_Type = 2#01# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0010# then
   Type_Found := True;
   Ada.Text_IO.Put_Line
     ("Pentium OverDrive cpu (75, 90, 100, 120, 133, 150, 166, 200)");
end if;

-- Pentium OverDrive processor for x486 processor-based systems
if Signature.Processor_Type = 2#01# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0011# then
   Type_Found := True;
   Ada.Text_IO.Put_Line
     ("Pentium OverDrive processor for x486 processor-based systems");
end if;

-- Pentium processor with MMX technology (166, 200)
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0100# then
   Type_Found := True;
   Ada.Text_IO.Put_Line
     ("Pentium processor with MMX technology (166, 200)");
end if;

-- Pentium OverDrive with MMX for Pentium (75, 90, 100, 120, 133)
if Signature.Processor_Type = 2#01# and
   Signature.Family         = 2#0101# and
   Signature.Model          = 2#0100# then
   Type_Found := True;
   Ada.Text_IO.Put_Line
     ("Pentium OverDrive processor with MMX " &
      "technology for Pentium processor (75, 90, 100, 120, 133)");
end if;

-- Pentium Pro processor
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0110# and
   Signature.Model          = 2#0001# then
   Type_Found := True;
   Ada.Text_IO.Put_Line ("Pentium Pro processor");
end if;

-- Pentium II processor, model 3
if Signature.Processor_Type = 2#00# and
   Signature.Family         = 2#0110# and
   Signature.Model          = 2#0011# then

```

```

    Type_Found := True;
    Ada.Text_IO.Put_Line ("Pentium II processor, model 3");
end if;

-- Pentium II processor, model 5 or Celeron processor
if Signature.Processor_Type = 2#00# and
   Signature.Family      = 2#0110# and
   Signature.Model       = 2#0101# then
    Type_Found := True;
    Ada.Text_IO.Put_Line
      ("Pentium II processor, model 5 or Celeron processor");
end if;

-- Pentium Pro OverDrive processor
if Signature.Processor_Type = 2#01# and
   Signature.Family      = 2#0110# and
   Signature.Model       = 2#0011# then
    Type_Found := True;
    Ada.Text_IO.Put_Line ("Pentium Pro OverDrive processor");
end if;

-- If no type recognized, we have an unknown. Display what
-- we _do_ know
if Type_Found = False then
    Ada.Text_IO.Put_Line ("Unknown processor");
end if;

-----
-- Display processor stepping level. --
-----

Ada.Text_IO.Put_Line ("Stepping level:" & Signature.Stepping'Img);

-----
-- Display vendor ID string. --
-----

Ada.Text_IO.Put_Line ("Vendor ID: " & Intel_CPU.Vendor_ID);

-----
-- Get the processors features. --
-----

Features := Intel_CPU.Features;

-----
-- Check for a FPU unit. --
-----

if Features.FPU = True then
    Ada.Text_IO.Put_Line ("Floating-Point unit available");
else
    Ada.Text_IO.Put_Line ("no Floating-Point unit");
end if; -- check for FPU

-----
-- List processor features. --
-----

```

```

Ada.Text_IO.Put_Line ("Supported features: ");

-- Virtual Mode Extension
if Features.VME = True then
  Ada.Text_IO.Put_Line ("    VME    - Virtual Mode Extension");
end if;

-- Debugging Extension
if Features.DE = True then
  Ada.Text_IO.Put_Line ("    DE    - Debugging Extension");
end if;

-- Page Size Extension
if Features.PSE = True then
  Ada.Text_IO.Put_Line ("    PSE    - Page Size Extension");
end if;

-- Time Stamp Counter
if Features.TSC = True then
  Ada.Text_IO.Put_Line ("    TSC    - Time Stamp Counter");
end if;

-- Model Specific Registers
if Features.MSR = True then
  Ada.Text_IO.Put_Line ("    MSR    - Model Specific Registers");
end if;

-- Physical Address Extension
if Features.PAE = True then
  Ada.Text_IO.Put_Line ("    PAE    - Physical Address Extension");
end if;

-- Machine Check Extension
if Features.MCE = True then
  Ada.Text_IO.Put_Line ("    MCE    - Machine Check Extension");
end if;

-- CMPXCHG8 instruction supported
if Features.CX8 = True then
  Ada.Text_IO.Put_Line ("    CX8    - CMPXCHG8 instruction");
end if;

-- on-chip APIC hardware support
if Features.APIC = True then
  Ada.Text_IO.Put_Line ("    APIC    - on-chip APIC hardware support");
end if;

-- Fast System Call
if Features.SEP = True then
  Ada.Text_IO.Put_Line ("    SEP    - Fast System Call");
end if;

-- Memory Type Range Registers
if Features.MTRR = True then
  Ada.Text_IO.Put_Line ("    MTRR    - Memory Type Range Registers");
end if;

```

```

-- Page Global Enable
if Features.PGE = True then
  Ada.Text_IO.Put_Line ("    PGE    - Page Global Enable");
end if;

-- Machine Check Architecture
if Features.MCA = True then
  Ada.Text_IO.Put_Line ("    MCA    - Machine Check Architecture");
end if;

-- Conditional Move Instruction Supported
if Features.CMOV = True then
  Ada.Text_IO.Put_Line
    ("    CMOV    - Conditional Move Instruction Supported");
end if;

-- Page Attribute Table
if Features.PAT = True then
  Ada.Text_IO.Put_Line ("    PAT    - Page Attribute Table");
end if;

-- 36-bit Page Size Extension
if Features.PSE_36 = True then
  Ada.Text_IO.Put_Line ("    PSE_36 - 36-bit Page Size Extension");
end if;

-- MMX technology supported
if Features.MMX = True then
  Ada.Text_IO.Put_Line ("    MMX     - MMX technology supported");
end if;

-- Fast FP Save and Restore
if Features.FXSR = True then
  Ada.Text_IO.Put_Line ("    FXSR    - Fast FP Save and Restore");
end if;

-----
-- Program done.  --
-----

Ada.Command_Line.Set_Exit_Status (Ada.Command_Line.Success);

exception

when others =>
  Ada.Command_Line.Set_Exit_Status (Ada.Command_Line.Failure);
  raise;

end Check_CPU;

```

23.7.2 Intel_CPU Package Specification

```

-----
--
-- file: INTEL_CPU.ADS
--
-- *****
-- * WARNING: for 32-bit Intel processors only *
--

```

```

--          *****
--
-- This package contains a number of subprograms that are useful in
-- determining the Intel x86 CPU (and the features it supports) on
-- which the program is running.
--
-- The package is based upon the information given in the Intel
-- Application Note AP-485: "Intel Processor Identification and the
-- CPUID Instruction" as of April 1998. This application note can be
-- found on www.intel.com.
--
-- It currently deals with 32-bit processors only, will not detect
-- features added after april 1998, and does not guarantee proper
-- results on Intel-compatible processors.
--
-- Cache info and x386 fpu type detection are not supported.
--
-- This package does not use any privileged instructions, so should
-- work on any OS running on a 32-bit Intel processor.
--
-----

with Interfaces;          use Interfaces;
-- for using unsigned types

with System.Machine_Code; use System.Machine_Code;
-- for using inline assembler code

with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
-- for inserting control characters

package Intel_CPU is

    -----
    -- Processor bits --
    -----

    subtype Num_Bits is Natural range 0 .. 31;
    -- the number of processor bits (32)

    -----
    -- Processor register --
    -----

    -- define a processor register type for easy access to
    -- the individual bits

    type Processor_Register is array (Num_Bits) of Boolean;
    pragma Pack (Processor_Register);
    for Processor_Register'Size use 32;

    -----
    -- Unsigned register --
    -----

    -- define a processor register type for easy access to
    -- the individual bytes

```

```

type Unsigned_Register is
  record
    L1 : Unsigned_8;
    H1 : Unsigned_8;
    L2 : Unsigned_8;
    H2 : Unsigned_8;
  end record;

for Unsigned_Register use
  record
    L1 at 0 range 0 .. 7;
    H1 at 0 range 8 .. 15;
    L2 at 0 range 16 .. 23;
    H2 at 0 range 24 .. 31;
  end record;

for Unsigned_Register'Size use 32;

-----
-- Intel processor vendor ID --
-----

Intel_Processor : constant String (1 .. 12) := "GenuineIntel";
-- indicates an Intel manufactured processor

-----
-- Processor signature register --
-----

-- a register type to hold the processor signature

type Processor_Signature is
  record
    Stepping      : Natural range 0 .. 15;
    Model         : Natural range 0 .. 15;
    Family        : Natural range 0 .. 15;
    Processor_Type : Natural range 0 .. 3;
    Reserved      : Natural range 0 .. 262143;
  end record;

for Processor_Signature use
  record
    Stepping      at 0 range 0 .. 3;
    Model         at 0 range 4 .. 7;
    Family        at 0 range 8 .. 11;
    Processor_Type at 0 range 12 .. 13;
    Reserved      at 0 range 14 .. 31;
  end record;

for Processor_Signature'Size use 32;

-----
-- Processor features register --
-----

-- a processor register to hold the processor feature flags

type Processor_Features is

```

```

record
    FPU      : Boolean;          -- floating point unit on chip
    VME      : Boolean;          -- virtual mode extension
    DE       : Boolean;          -- debugging extension
    PSE      : Boolean;          -- page size extension
    TSC      : Boolean;          -- time stamp counter
    MSR      : Boolean;          -- model specific registers
    PAE      : Boolean;          -- physical address extension
    MCE      : Boolean;          -- machine check extension
    CX8      : Boolean;          -- cmpxchg8 instruction
    APIC     : Boolean;          -- on-chip apic hardware
    Res_1    : Boolean;          -- reserved for extensions
    SEP      : Boolean;          -- fast system call
    MTRR     : Boolean;          -- memory type range registers
    PGE      : Boolean;          -- page global enable
    MCA      : Boolean;          -- machine check architecture
    CMOV     : Boolean;          -- conditional move supported
    PAT      : Boolean;          -- page attribute table
    PSE_36   : Boolean;          -- 36-bit page size extension
    Res_2    : Natural range 0 .. 31; -- reserved for extensions
    MMX      : Boolean;          -- MMX technology supported
    FXSR     : Boolean;          -- fast FP save and restore
    Res_3    : Natural range 0 .. 127; -- reserved for extensions
end record;

for Processor_Features use
record
    FPU      at 0 range 0 .. 0;
    VME      at 0 range 1 .. 1;
    DE       at 0 range 2 .. 2;
    PSE      at 0 range 3 .. 3;
    TSC      at 0 range 4 .. 4;
    MSR      at 0 range 5 .. 5;
    PAE      at 0 range 6 .. 6;
    MCE      at 0 range 7 .. 7;
    CX8      at 0 range 8 .. 8;
    APIC     at 0 range 9 .. 9;
    Res_1    at 0 range 10 .. 10;
    SEP      at 0 range 11 .. 11;
    MTRR     at 0 range 12 .. 12;
    PGE      at 0 range 13 .. 13;
    MCA      at 0 range 14 .. 14;
    CMOV     at 0 range 15 .. 15;
    PAT      at 0 range 16 .. 16;
    PSE_36   at 0 range 17 .. 17;
    Res_2    at 0 range 18 .. 22;
    MMX      at 0 range 23 .. 23;
    FXSR     at 0 range 24 .. 24;
    Res_3    at 0 range 25 .. 31;
end record;

for Processor_Features'Size use 32;

-----
-- Subprograms --
-----

function Has_FPU return Boolean;

```

```

-- return True if a FPU is found
-- use only if CPUID is not supported

function Has_CPUID return Boolean;
-- return True if the processor supports the CPUID instruction

function CPUID_Level return Natural;
-- return the CPUID support level (0, 1 or 2)
-- can only be called if the CPUID instruction is supported

function Vendor_ID return String;
-- return the processor vendor identification string
-- can only be called if the CPUID instruction is supported

function Signature return Processor_Signature;
-- return the processor signature
-- can only be called if the CPUID instruction is supported

function Features return Processor_Features;
-- return the processors features
-- can only be called if the CPUID instruction is supported

private

-----
-- EFLAGS bit names --
-----

ID_Flag : constant Num_Bits := 21;
-- ID flag bit

end Intel_CPU;
```

23.7.3 Intel_CPU Package Body

```

package body Intel_CPU is

-----
-- Detect FPU presence --
-----

-- There is a FPU present if we can set values to the FPU Status
-- and Control Words.

function Has_FPU return Boolean is

  Register : Unsigned_16;
  -- processor register to store a word

begin

  -- check if we can change the status word
  Asm (

    -- the assembler code
    "finit"           & LF & HT &    -- reset status word
    "movw $0x5A5A, %%ax" & LF & HT &  -- set value status word
    "fnstsw %0"       & LF & HT &    -- save status word
```



```

        "movw %%ax, %0",                -- store status word

        -- output stored in Register
        -- register must be a memory location
        Outputs => Unsigned_16'Asm_output ("=m", Register),

        -- tell compiler that we used eax
        Clobber => "eax");

-- if the status word is zero, there is no FPU
if Register = 0 then
    return False; -- no status word
end if; -- check status word value

-- check if we can get the control word
Asm (

    -- the assembler code
    "fnstcw %0", -- save the control word

    -- output into Register
    -- register must be a memory location
    Outputs => Unsigned_16'Asm_output ("=m", Register));

-- check the relevant bits
if (Register and 16#103F#) /= 16#003F# then
    return False; -- no control word
end if; -- check control word value

-- FPU found
return True;

end Has_FPU;

-----
-- Detect CPUID instruction --
-----

-- The processor supports the CPUID instruction if it is possible
-- to change the value of ID flag bit in the EFLAGS register.

function Has_CPUID return Boolean is

    Original_Flags, Modified_Flags : Processor_Register;
    -- EFLAG contents before and after changing the ID flag

begin

    -- try flipping the ID flag in the EFLAGS register
    Asm (

        -- the assembler code
        "pushfl"          & LF & HT & -- push EFLAGS on stack
        "pop %%eax"       & LF & HT & -- pop EFLAGS into eax
        "movl %%eax, %0"   & LF & HT & -- save EFLAGS content
        "xor $0x200000, %%eax" & LF & HT & -- flip ID flag
        "push %%eax"       & LF & HT & -- push EFLAGS on stack
        "popfl"           & LF & HT & -- load EFLAGS register
    );

```

```

"pushfl"          & LF & HT &      -- push EFLAGS on stack
"pop %1",          -- save EFLAGS content

-- output values, may be anything
-- Original_Flags is %0
-- Modified_Flags is %1
Outputs =>
  (Processor_Register'Asm_output ("=g", Original_Flags),
   Processor_Register'Asm_output ("=g", Modified_Flags)),

-- tell compiler eax is destroyed
Clobber => "eax");

-- check if CPUID is supported
if Original_Flags(ID_Flag) /= Modified_Flags(ID_Flag) then
  return True; -- ID flag was modified
else
  return False; -- ID flag unchanged
end if; -- check for CPUID

end Has_CPUID;

-----
-- Get CPUID support level --
-----

function CPUID_Level return Natural is

  Level : Unsigned_32;
  -- returned support level

begin

  -- execute CPUID, storing the results in the Level register
  Asm (

    -- the assembler code
    "cpuid", -- execute CPUID

    -- zero is stored in eax
    -- returning the support level in eax
    Inputs => Unsigned_32'Asm_input ("a", 0),

    -- eax is stored in Level
    Outputs => Unsigned_32'Asm_output ("=a", Level),

    -- tell compiler ebx, ecx and edx registers are destroyed
    Clobber => "ebx, ecx, edx");

  -- return the support level
  return Natural (Level);

end CPUID_Level;

-----
-- Get CPU Vendor ID String --
-----

```

```

-- The vendor ID string is returned in the ebx, ecx and edx register
-- after executing the CPUID instruction with eax set to zero.
-- In case of a true Intel processor the string returned is
-- "GenuineIntel"

function Vendor_ID return String is

    Ebx, Ecx, Edx : Unsigned_Register;
    -- registers containing the vendor ID string

    Vendor_ID : String (1 .. 12);
    -- the vendor ID string

begin

    -- execute CPUID, storing the results in the processor registers
    Asm (

        -- the assembler code
        "cpuid",    -- execute CPUID

        -- zero stored in eax
        -- vendor ID string returned in ebx, ecx and edx
        Inputs => Unsigned_32'Asm_input ("a", 0),

        -- ebx is stored in Ebx
        -- ecx is stored in Ecx
        -- edx is stored in Edx
        Outputs => (Unsigned_Register'Asm_output ("=b", Ebx),
                    Unsigned_Register'Asm_output ("=c", Ecx),
                    Unsigned_Register'Asm_output ("=d", Edx));

        -- now build the vendor ID string
        Vendor_ID( 1) := Character'Val (Ebx.L1);
        Vendor_ID( 2) := Character'Val (Ebx.H1);
        Vendor_ID( 3) := Character'Val (Ebx.L2);
        Vendor_ID( 4) := Character'Val (Ebx.H2);
        Vendor_ID( 5) := Character'Val (Edx.L1);
        Vendor_ID( 6) := Character'Val (Edx.H1);
        Vendor_ID( 7) := Character'Val (Edx.L2);
        Vendor_ID( 8) := Character'Val (Edx.H2);
        Vendor_ID( 9) := Character'Val (Ecx.L1);
        Vendor_ID(10) := Character'Val (Ecx.H1);
        Vendor_ID(11) := Character'Val (Ecx.L2);
        Vendor_ID(12) := Character'Val (Ecx.H2);

        -- return string
        return Vendor_ID;

    end Vendor_ID;

    -----
    -- Get processor signature --
    -----

function Signature return Processor_Signature is

    Result : Processor_Signature;

```

```

    -- processor signature returned
begin
    -- execute CPUID, storing the results in the Result variable
    Asm (
        -- the assembler code
        "cpuid",    -- execute CPUID

        -- one is stored in eax
        -- processor signature returned in eax
        Inputs => Unsigned_32'Asm_input ("a", 1),

        -- eax is stored in Result
        Outputs => Processor_Signature'Asm_output ("a", Result),

        -- tell compiler that ebx, ecx and edx are also destroyed
        Clobber => "ebx, ecx, edx");

    -- return processor signature
    return Result;
end Signature;

-----
-- Get processor features --
-----

function Features return Processor_Features is
    Result : Processor_Features;
    -- processor features returned
begin
    -- execute CPUID, storing the results in the Result variable
    Asm (
        -- the assembler code
        "cpuid",    -- execute CPUID

        -- one stored in eax
        -- processor features returned in edx
        Inputs => Unsigned_32'Asm_input ("a", 1),

        -- edx is stored in Result
        Outputs => Processor_Features'Asm_output ("d", Result),

        -- tell compiler that ebx and ecx are also destroyed
        Clobber => "ebx, ecx");

    -- return processor signature
    return Result;
end Features;

end Intel_CPU;

```

24 Performance Considerations

The GNAT system provides a number of options that allow a trade-off between

- performance of the generated code
- speed of compilation
- minimization of dependences and recompilation
- the degree of run-time checking.

The defaults (if no options are selected) aim at improving the speed of compilation and minimizing dependences, at the expense of performance of the generated code:

- no optimization
- no inlining of subprogram calls
- all run-time checks enabled except overflow and elaboration checks

These options are suitable for most program development purposes. This chapter describes how you can modify these choices, and also provides some guidelines on debugging optimized code.

24.1 Controlling Run-Time Checks

By default, GNAT generates all run-time checks, except arithmetic overflow checking for integer operations and checks for access before elaboration on subprogram calls. The latter are not required in default mode, because all necessary checking is done at compile time. Two gnat qualifiers, `‘/CHECKS=SUPPRESS_ALL’` and `‘/CHECKS=OVERFLOW’` allow this default to be modified. See Section 3.2.5 [Run-Time Checks], page 47.

Our experience is that the default is suitable for most development purposes.

We treat integer overflow specially because these are quite expensive and in our experience are not as important as other run-time checks in the development process. Note that division by zero is not considered an overflow check, and divide by zero checks are generated where required by default.

Elaboration checks are off by default, and also not needed by default, since GNAT uses a static elaboration analysis approach that avoids the need for run-time checking. This manual contains a full chapter discussing the issue of elaboration checks, and if the default is not satisfactory for your use, you should read this chapter.

For validity checks, the minimal checks required by the Ada Reference Manual (for case statements and assignments to array elements) are on by default. These can be suppressed by use of the `‘-gnatVn’` qualifier. Note that in Ada 83, there were no validity checks, so if the Ada 83 mode is acceptable (or when comparing GNAT performance with an Ada 83 compiler), it may be reasonable to routinely use `‘-gnatVn’`. Validity checks are also suppressed entirely if `‘/CHECKS=SUPPRESS_ALL’` is used.

Note that the setting of the qualifiers controls the default setting of the checks. They may be modified using either `pragma Suppress` (to remove checks) or `pragma Unsuppress` (to add back suppressed checks) in the program source.

24.2 Optimization Levels

The default is optimization off. This results in the fastest compile times, but GNAT makes absolutely no attempt to optimize, and the generated programs are considerably larger and slower than when optimization is enabled. You can use the `/OPTIMIZE` on the `GNAT COMPILE` command line to control the optimization level:

```
/OPTIMIZE=NONE
    no optimization (the default)

/OPTIMIZE=SOME
    medium level optimization

/OPTIMIZE=ALL
    full optimization

/OPTIMIZE=INLINING
    full optimization, and also attempt automatic inlining of small subprograms
    within a unit (see Section 24.4 [Inlining of Subprograms], page 269).
```

Higher optimization levels perform more global transformations on the program and apply more expensive analysis algorithms in order to generate faster and more compact code. The price in compilation time, and the resulting improvement in execution time, both depend on the particular application and the hardware environment. You should experiment to find the best level for your application.

Note: Unlike some other compilation systems, `GNAT COMPILE` has been tested extensively at all optimization levels. There are some bugs which appear only with optimization turned on, but there have also been bugs which show up only in *unoptimized* code. Selecting a lower level of optimization does not improve the reliability of the code generator, which in practice is highly reliable at all optimization levels.

Note regarding the use of `/OPTIMIZE=INLINING`: The use of this optimization level is generally discouraged with GNAT, since it often results in larger executables which run more slowly. See further discussion of this point in see Section 24.4 [Inlining of Subprograms], page 269.

24.3 Debugging Optimized Code

Since the compiler generates debugging tables for a compilation unit before it performs optimizations, the optimizing transformations may invalidate some of the debugging data. You therefore need to anticipate certain anomalous situations that may arise while debugging optimized code. This section describes the most common cases.

1. *The "hopping Program Counter"*: Repeated 'step' or 'next' commands show the PC bouncing back and forth in the code. This may result from any of the following optimizations:
 - *Common subexpression elimination*: using a single instance of code for a quantity that the source computes several times. As a result you may not be able to stop on what looks like a statement.

- *Invariant code motion*: moving an expression that does not change within a loop, to the beginning of the loop.
 - *Instruction scheduling*: moving instructions so as to overlap loads and stores (typically) with other code, or in general to move computations of values closer to their uses. Often this causes you to pass an assignment statement without the assignment happening and then later bounce back to the statement when the value is actually needed. Placing a breakpoint on a line of code and then stepping over it may, therefore, not always cause all the expected side-effects.
2. *The "big leap"*: More commonly known as *cross-jumping*, in which two identical pieces of code are merged and the program counter suddenly jumps to a statement that is not supposed to be executed, simply because it (and the code following) translates to the same thing as the code that *was* supposed to be executed. This effect is typically seen in sequences that end in a jump, such as a `goto`, a `return`, or a `break` in a C `qualifier` statement.
 3. *The "roving variable"*: The symptom is an unexpected value in a variable. There are various reasons for this effect:
 - In a subprogram prologue, a parameter may not yet have been moved to its "home".
 - A variable may be dead, and its register re-used. This is probably the most common cause.
 - As mentioned above, the assignment of a value to a variable may have been moved.
 - A variable may be eliminated entirely by value propagation or other means. In this case, GCC may incorrectly generate debugging information for the variable

In general, when an unexpected value appears for a local variable or parameter you should first ascertain if that value was actually computed by your program, as opposed to being incorrectly reported by the debugger. Record fields or array elements in an object designated by an access value are generally less of a problem, once you have ascertained that the access value is sensible. Typically, this means checking variables in the preceding code and in the calling subprogram to verify that the value observed is explainable from other values (one must apply the procedure recursively to those other values); or re-running the code and stopping a little earlier (perhaps before the call) and stepping to better see how the variable obtained the value in question; or continuing to step *from* the point of the strange value to see if code motion had simply moved the variable's assignments later.

24.4 Inlining of Subprograms

A call to a subprogram in the current unit is inlined if all the following conditions are met:

- The optimization level is at least `/OPTIMIZE=SOME`.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else that `GNAT COMPILE` cannot support in inlined subprograms.
- The call occurs after the definition of the body of the subprogram.

- Either **pragma Inline** applies to the subprogram or it is small and automatic inlining (optimization level **/OPTIMIZE=INLINING**) is specified.

Calls to subprograms in **with**'ed units are normally not inlined. To achieve this level of inlining, the following conditions must all be true:

- The optimization level is at least **/OPTIMIZE=SOME**.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else **GNAT COMPILE** cannot support in inlined subprograms.
- The call appears in a body (not in a package spec).
- There is a **pragma Inline** for the subprogram.
- The **/INLINE** qualifier is used in the **GNAT COMPILE** command line

Note that specifying the **'/INLINE=PRAGMA'** qualifier causes additional compilation dependencies. Consider the following:

```
package R is
  procedure Q;
  pragma Inline (Q);
end R;
package body R is
  ...
end R;

with R;
procedure Main is
begin
  ...
  R.Q;
end Main;
```

With the default behavior (no **'/INLINE=PRAGMA'** qualifier specified), the compilation of the **Main** procedure depends only on its own source, **'MAIN.ADB'**, and the spec of the package in file **'R.ADS'**. This means that editing the body of **R** does not require recompiling **Main**.

On the other hand, the call **R.Q** is not inlined under these circumstances. If the **'/INLINE=PRAGMA'** qualifier is present when **Main** is compiled, the call will be inlined if the body of **Q** is small enough, but now **Main** depends on the body of **R** in **'R.ADB'** as well as on the spec. This means that if this body is edited, the main program must be recompiled. Note that this extra dependency occurs whether or not the call is in fact inlined by **GNAT COMPILE**.

The use of front end inlining with **'-gnatN'** generates similar additional dependencies.

Note: The **/INLINE=SUPPRESS** qualifier can be used to prevent all inlining. This qualifier overrides all other conditions and ensures that no inlining occurs. The extra dependences resulting from **'/INLINE=PRAGMA'** will still be active, even if this qualifier is used to suppress the resulting inlining actions.

Note regarding the use of **/OPTIMIZE=INLINING**: There is no difference in inlining behavior between **/OPTIMIZE=ALL** and **/OPTIMIZE=INLINING** for subprograms with an ex-

explicit pragma `Inline` assuming the use of `/INLINE=PRAGMA` or `-gnatN` (the qualifiers that activate inlining). If you have used pragma `Inline` in appropriate cases, then it is usually much better to use `/OPTIMIZE=ALL` and `/INLINE=PRAGMA` and avoid the use of `/OPTIMIZE=INLINING` which in this case only has the effect of inlining subprograms you did not think should be inlined. We often find that the use of `/OPTIMIZE=INLINING` slows down code by performing excessive inlining, leading to increased instruction cache pressure from the increased code size. So the bottom line here is that you should not automatically assume that `/OPTIMIZE=INLINING` is better than `/OPTIMIZE=ALL`, and indeed you should use `/OPTIMIZE=INLINING` only if tests show that it actually improves performance.

24.5 Coverage Analysis

GNAT supports the Digital Performance Coverage Analyzer (PCA), which allows the user to determine the distribution of execution time across a program, see Section 20.8 [Profiling], page 203 for details of usage.

GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

-	
-83 (gnathtml)	201
-c (gnatname)	106
-d (gnathtml)	201
-d (gnatname)	106
-D (gnatname)	106
-f (GNAT LINK)	88
-f (gnathtml)	202
-fstack-check	49
'-gnatem' (GNAT COMPILE)	56
'-gnatN' (GNAT COMPILE)	53
'-gnatN' qualifier	19
'-gnatT' (GNAT COMPILE)	49
'-gnatwp' (GNAT COMPILE)	39
'-gnatwp' (GNAT COMPILE)	39
-h (gnatname)	106
-I (gnathtml)	202
-l (gnathtml)	202
-o (gnathtml)	202
-p (gnathtml)	202
-P (gnatname)	106
-sc (gnathtml)	202
-t (gnathtml)	202
-v (gnatname)	106
-w	40
/	
'/83' (GNAT COMPILE)	51
/ACTIONS=BIND (GNAT MAKE)	92
/ACTIONS=COMPILE (GNAT MAKE)	92
/ACTIONS=LINK (GNAT MAKE)	93
/ALL_FILES (GNAT MAKE)	92
/ALL_UNITS (GNAT LIST)	188
/ASM (GNAT COMPILE)	30
/BIND_FILE=ADA (GNAT BIND)	81
/BIND_FILE=ADA (GNAT LINK)	87
/BIND_FILE=C (GNAT BIND)	81
/BIND_FILE=C (GNAT LINK)	88
/BINDER_QUALIFIERS (GNAT MAKE)	96
'/CHECKS=ASSERTIONS' (GNAT COMPILE)	40
'/CHECKS=ELABORATION' (GNAT COMPILE)	49, 55
'/CHECKS=OVERFLOW' (GNAT COMPILE)	48, 267
'/CHECKS=SUPPRESS_ALL' (GNAT COMPILE)	48, 267
/COMPILATION (GNAT CHOP)	101
/COMPILER_QUALIFIERS (GNAT MAKE)	96
/CONDITIONAL_SOURCE_SEARCH (GNAT MAKE)	95
/CONFIG=file (GNAT LIBRARY)	185
/CONTINUE_ON_ERROR (GNAT MAKE)	93
/CREATE=directory (GNAT LIBRARY)	185
/DEBUG (GNAT COMPILE)	30
/DEBUG (GNAT LINK)	88
/DEBUG=TRACEBACK (GNAT LINK)	88
/DELETE=directory (GNAT LIBRARY)	185
/DEPENDENCIES (GNAT LIST)	188
/DEPENDENCIES_LIST (GNAT MAKE)	94
/DO_OBJECT_CHECK (GNAT MAKE)	94
/ELABORATION_DEPENDENCIES (GNAT BIND)	81
'/ERROR_LIMIT' (GNAT COMPILE)	34
/EXECUTABLE (GNAT LINK)	88
/EXECUTABLE (GNAT MAKE)	94
'/EXPAND_SOURCE' (GNAT COMPILE)	53
/FILE_NAME_MAX_LENGTH (GNAT CHOP)	101
'/FILE_NAME_MAX_LENGTH' (GNAT COMPILE)	52
/FORCE_COMPILE (GNAT MAKE)	92
/HELP (GNAT BIND)	81
/IDENTIFIER_CHARACTER_SET (GNAT COMPILE) ..	51
/IN_PLACE (GNAT MAKE)	93
'/INLINE=PRAGMA' (GNAT COMPILE)	52, 270
'/INLINE=PRAGMA' qualifier	18
/INLINE=SUPPRESS (GNAT COMPILE)	270
/LIBRARY_SEARCH (GNAT MAKE)	95
/LINKER_OPTION_LIST (GNAT BIND)	81
/LINKER_QUALIFIERS (GNAT MAKE)	96
'/LIST' (GNAT COMPILE)	34
/MAPPING (GNAT MAKE)	92
/MINIMAL_RECOMPILATION (GNAT MAKE)	93
/NOCURRENT_DIRECTORY (GNAT COMPILE)	30
/NOCURRENT_DIRECTORY (GNAT MAKE)	95
'/NOLOAD' (GNAT COMPILE)	50
/NOMAIN (GNAT BIND)	82
/NOMAIN (GNAT MAKE)	94
/NOOUTPUT (GNAT BIND)	81
/NOSTD_INCLUDES (GNAT MAKE)	95
/NOSTD_LIBRARIES (GNAT MAKE)	95
/NOTIME_STAMP_CHECK (GNAT BIND)	80
/OBJECT_LIST (GNAT BIND)	81
/OBJECT_SEARCH (GNAT MAKE)	95
/OPTIMIZE (GNAT COMPILE)	268
/ORDER_OF_ELABORATION (GNAT BIND)	81
/OUTPUT (GNAT BIND)	81
/OUTPUT=OBJECTS (GNAT LIST)	188
/OUTPUT=OPTIONS (GNAT LIST)	188
/OUTPUT=SOURCES (GNAT LIST)	188
/OUTPUT=UNITS (GNAT LIST)	188
/OUTPUT=VERBOSE (GNAT LIST)	188
/OVERWRITE (GNAT CHOP)	101
/PESSIMISTIC_ELABORATION (GNAT BIND)	80
/PRESERVE (GNAT CHOP)	101
/PROCESSES (GNAT MAKE)	93
/QUIET (GNAT CHOP)	101
/QUIET (GNAT MAKE)	94
/READ_SOURCES=ALL (GNAT BIND)	79
/READ_SOURCES=NONE (GNAT BIND)	79
/REASONS (GNAT MAKE)	94
/REFERENCE (GNAT CHOP)	101
/REPORT_ERRORS=BRIEF (GNAT BIND)	79
'/REPORT_ERRORS=FULL' (GNAT COMPILE)	35
/REPORT_ERRORS=VERBOSE (GNAT BIND)	79

'/REPORT_ERRORS=VERBOSE' (GNAT COMPILE)	33
'/REPRESENTATION_INFO' (GNAT COMPILE)	55
/RESTRICTION_LIST (GNAT BIND)	81
/RUNTIME_SYSTEM (GNAT BIND)	84
/RUNTIME_SYSTEM (GNAT COMPILE)	30
/RUNTIME_SYSTEM (GNAT FIND)	169
/RUNTIME_SYSTEM (GNAT LIST)	188
/RUNTIME_SYSTEM (GNAT MAKE)	95
/RUNTIME_SYSTEM (GNAT XREF)	168
/SEARCH (GNAT COMPILE)	30
/SEARCH (GNAT MAKE)	95
/SET=directory (GNAT LIBRARY)	185
/SKIP_MISSING (GNAT MAKE)	95
/SOURCE_SEARCH (GNAT MAKE)	95
/SWITCH_CHECK (GNAT MAKE)	94
'/SYNTAX_ONLY' (GNAT COMPILE)	50
'/TRACE_UNITS' qualifier	212
'/TREE_OUTPUT' (GNAT COMPILE)	53
'/TRY_SEMANTICS' (GNAT COMPILE)	35
/UNIQUE (GNAT MAKE)	94
'/UNIQUE_ERROR_TAG' (GNAT COMPILE)	34
'/UNITS_LIST' (GNAT COMPILE)	53
/VERBOSE (GNAT CHOP)	101
/VERBOSE (GNAT COMPILE)	31
/VERBOSE (GNAT LINK)	88
'/WARNINGS=BIASED_ROUNDING' (GNAT COMPILE)	37
'/WARNINGS=BRIEF' (GNAT COMPILE)	34
'/WARNINGS=CONDITIONALS' (GNAT COMPILE)	37
'/WARNINGS=ELABORATION' (GNAT COMPILE)	39
/WARNINGS=ERROR (GNAT BIND)	80
'/WARNINGS=ERROR' (GNAT COMPILE)	38
'/WARNINGS=HIDING' (GNAT COMPILE)	38
'/WARNINGS=IMPLEMENTATION' (GNAT COMPILE)	38
'/WARNINGS=IMPLICIT_DEREFERENCE' (GNAT COMPILE)	38
'/WARNINGS=NOBIASED_ROUNDING' (GNAT COMPILE)	37
'/WARNINGS=NOCONDITIONALS' (GNAT COMPILE)	38
'/WARNINGS=NOELABORATION' (GNAT COMPILE)	39
'/WARNINGS=NOHIDING' (GNAT COMPILE)	38
'/WARNINGS=NOIMPLEMENTATION' (GNAT COMPILE)	39
'/WARNINGS=NOIMPLICIT_DEREFERENCE' (GNAT COMPILE)	38
'/WARNINGS=NOOPTIONAL' (GNAT COMPILE)	37
'/WARNINGS=NOOVERLAYS' (GNAT COMPILE)	39
'/WARNINGS=NOREDUNDANT' (GNAT COMPILE)	39
'/WARNINGS=NOUNREFERENCED_FORMALS' (GNAT COMPILE)	38
'/WARNINGS=NOUNUSED' (GNAT COMPILE)	40
'/WARNINGS=OPTIONAL' (GNAT COMPILE)	37
'/WARNINGS=OVERLAYS' (GNAT COMPILE)	39
'/WARNINGS=REDUNDANT' (GNAT COMPILE)	39
/WARNINGS=SUPPRESS (GNAT BIND)	80
'/WARNINGS=SUPPRESS' (GNAT COMPILE)	40
'/WARNINGS=UNREFERENCED_FORMALS' (GNAT COMPILE)	38

'/WARNINGS=UNUSED' (GNAT COMPILE)	40
/WIDE_CHARACTER_ENCODING (GNAT COMPILE)	52
'/XDEBUG' (GNAT COMPILE)	55
'/XREF=SUPPRESS' (GNAT COMPILE)	56
/ZERO_MAIN (GNAT BIND)	83

—

__gnat_finalize	70
__gnat_initialize	70
__gnat_set_globals	65, 67
_main	200

A

Access before elaboration	47
Access-to-subprogram	162
ACVC, Ada 83 tests	51
Ada	86, 212
Ada 83 compatibility	51
Ada 95 Language Reference Manual	2
Ada expressions	208
Ada Library Information files	19
Ada.Characters.Latin_1	12
ADA_INCLUDE_PATH	57
ADA_OBJECTS_PATH	85
adafinal	69, 82
adainit	65, 82
Address Clauses, warnings	39
'ali' files	19
Annex A	212
Annex B	212
Arbitrary File Naming Conventions	105
Asm	22
Assert	40
Assertions	40

B

Biased rounding	37
Binder consistency checks	80
Binder output file	21
Binder, multiple input files	82
Breakpoints and tasks	210

C

C	22
C++	22
Calling Conventions	21
Check, elaboration	49
Check, overflow	48
Check.CPU procedure	252
Checks, access before elaboration	47
Checks, division by zero	47
Checks, elaboration	143
Checks, overflow	267
Checks, suppressing	48

COBOL 22
 code page 437 12
 code page 850 12
 Command line length 88
 Compatibility 221
 Compilation model 11
 Conditionals, constant 37
 Configuration pragmas 103
 Consistency checks, in binder 80
 Convention Ada 21
 Convention Asm 22
 Convention Assembler 22
 Convention C 22
 Convention C++ 22
 Convention COBOL 22
 Convention Default 22
 Convention DLL 23
 Convention External 22
 Convention Fortran 22
 Convention Stdcall 23
 Convention Stubbed 23
 Convention Win32 23
 Conventions 3
CR 11
 Cyrillic 12

D

Debug 40
Debug Pool 191
 Debugger 205
 Debugging 205
 Debugging Generic Units 210
 Debugging information, including 88
 Debugging options 53
Default 22
 Dependencies, producing list 94
 Dependency rules 91
 Dereferencing, implicit 38
 Division by zero 47
DLL 23

E

Elaborate 145
Elaborate_All 145
Elaborate_Body 145
 Elaboration checks 49, 143
 Elaboration control 141, 163
 Elaboration of library tasks 153
 Elaboration order control 27
 Elaboration, warnings 39
Eliminate 195
EMACS 8
 End of source file 11
 Error messages, suppressing 35
 EUC Coding 13
 Exceptions 209

Export 199
External 22

F

FDL, GNU Free Documentation License 273
FF 11
 File names 15, 16
 File naming schemes, alternative 16
 Floating-Point types 222
 Foreign Languages 21
 Formals, unreferenced 38
Fortran 22

G

GDB 205
 Generic formal parameters 51
 Generics 18, 210
GNAT 86, 213
 GNAT Abnormal Termination or Failure to
 Terminate 211
GNAT BIND 59
GNAT CHOP 99
 GNAT compilation model 11
GNAT ELIM 195
GNAT FIND 167
GNAT KRUNCH 177
 GNAT library 27
GNAT LIBRARY 185
GNAT LINK 87
GNAT LIST 187
GNAT MAKE 91
GNAT PREPROCESS 181
GNAT STUB 193
GNAT XREF 167
 ‘GNAT.ADC’ 15, 104
gnat_argc 85
gnat_argv 85
GNAT_STACK_LIMIT 49
GNAT1 29

H

Hiding of Declarations 38
HT 11

I

Implicit dereferencing	38
Inline	18, 270
Inlining	28
Inlining, warnings	39
Intel_CPU package body	262
Intel_CPU package specification	258
Interfaces	86, 212
Interfacing to Ada	21
Interfacing to Assembly	22
Interfacing to C	22
Interfacing to C++	22
Interfacing to COBOL	22
Interfacing to Fortran	22
Internal trees, writing to file	53

L

Latin-1	11, 12
Latin-2	12
Latin-3	12
Latin-4	12
Latin-5	12
LF	11
Library browser	187
Library builder	185
Library tasks, elaboration issues	153
Linker libraries	95
LSE	202

M

Machine_Overflows	48
Main Program	69
Mixed Language Programming	20
Multiple units, syntax checking	50

N

No code generated	29
No_Entry_Calls_In_Elaboration_Code	157

O

Object file list	71
Order of elaboration	141
Other Ada compilers	21
Overflow checks	48, 267

P

Parallel make	93
PCA	203
Performance	267
pragma Elaborate	145
pragma Elaborate_All	145
pragma Elaborate_Body	145

pragma Inline	270
pragma Preelaborate	145
pragma Pure	144
pragma Suppress	267
pragma Unsuppress	267
Pragmas, configuration	103
Preelaborate	145
Pure	144

R

Recompilation, by GNAT MAKE	96
Rounding, biased	37
RTL	30

S

SDP_Table_Build	66
Search paths, for GNAT MAKE	95
Shift JIS Coding	13
Source file, end	11
Source files, suppressing search	95
Source files, use by binder	59
Source_File_Name pragma	15, 16
Source_Reference	101
Stack Overflow Checking	49
stack traceback	213
stack unwinding	213
Starlet	234
Stdcall	23
storage, pool, memory corruption	191
Stubbed	23
Style checking	43
SUB	11
Subunits	18
Suppress	48, 267
Suppressing checks	48
SYS\$ERROR	33, 34
SYS\$OUTPUT	33
System	86, 213
System.IO	57

T

Task switching	210
Tasks	209
Time Slicing	49
Time stamp checks, in binder	80
traceback	213
traceback, non-symbolic	213
traceback, symbolic	217
Tree file	195
Typographical conventions	3

U

Unsuppress	49, 267
Upper-Half Coding	13

V

Validity Checking	41
Version skew (avoided by GNAT MAKE)	6
Volatile parameter	251
VT	11

W

Warning messages	35
Warnings	80
Warnings, treat as error	38
Win32	23
Writing internal trees	53

Z

Zero Cost Exceptions	66
-----------------------------------	----

Table of Contents

About This Guide	1
What This Guide Contains	1
What You Should Know before Reading This Guide	2
Related Information	2
Conventions	2
1 Getting Started with GNAT	5
1.1 Running GNAT	5
1.2 Running a Simple Ada Program	5
1.3 Running a Program with Multiple Units	6
1.4 Using the GNAT MAKE Utility	7
1.5 Editing with EMACS	8
2 The GNAT Compilation Model	11
2.1 Source Representation	11
2.2 Foreign Language Representation	11
2.2.1 Latin-1	11
2.2.2 Other 8-Bit Codes	12
2.2.3 Wide Character Encodings	12
2.3 File Naming Rules	14
2.4 Using Other File Names	15
2.5 Alternative File Naming Schemes	15
2.6 Generating Object Files	17
2.7 Source Dependencies	18
2.8 The Ada Library Information Files	19
2.9 Binding an Ada Program	20
2.10 Mixed Language Programming	20
2.10.1 Interfacing to C	20
2.10.2 Calling Conventions	21
2.11 Building Mixed Ada & C++ Programs	23
2.11.1 Interfacing to C++	23
2.11.2 Linking a Mixed C++ & Ada Program	24
2.11.3 A Simple Example	25
2.11.4 Adapting the Run Time to a New C++ Compiler	26
2.12 Comparison between GNAT and C/C++ Compilation Models	27
2.13 Comparison between GNAT and Conventional Ada Library Models	27

3	Compiling Using GNAT COMPILE	29
3.1	Compiling Programs	29
3.2	Qualifiers for GNAT COMPILE	29
3.2.1	Output and Error Message Control	33
3.2.2	Debugging and Assertion Control	40
3.2.3	Validity Checking	41
3.2.4	Style Checking	43
3.2.5	Run-Time Checks	47
3.2.6	Stack Overflow Checking	49
3.2.7	Run-Time Control	49
3.2.8	Using GNAT COMPILE for Syntax Checking	50
3.2.9	Using GNAT COMPILE for Semantic Checking	50
3.2.10	Compiling Ada 83 Programs	51
3.2.11	Character Set Control	51
3.2.12	File Naming Control	52
3.2.13	Subprogram Inlining Control	52
3.2.14	Auxiliary Output Control	53
3.2.15	Debugging Control	53
3.2.16	Units to Sources Mapping Files	56
3.3	Search Paths and the Run-Time Library (RTL)	56
3.4	Order of Compilation Issues	57
3.5	Examples	58
4	Binding Using GNAT BIND	59
4.1	Running GNAT BIND	59
4.2	Generating the Binder Program in C	71
4.3	Consistency-Checking Modes	79
4.4	Binder Error Message Control	79
4.5	Elaboration Control	80
4.6	Output Control	80
4.7	Binding with Non-Ada Main Programs	81
4.8	Binding Programs with No Main Subprogram	82
4.9	Summary of Binder Qualifiers	83
4.10	Command-Line Access	85
4.11	Search Paths for GNAT BIND	85
4.12	Examples of GNAT BIND Usage	86
5	Linking Using GNAT LINK	87
5.1	Running GNAT LINK	87
5.2	Qualifiers for GNAT LINK	87
5.3	Setting Stack Size from GNAT LINK	88
5.4	Setting Heap Size from GNAT LINK	89

6	The GNAT Make Program GNAT MAKE.....	91
6.1	Running GNAT MAKE	91
6.2	Qualifiers for GNAT MAKE	91
6.3	Mode Qualifiers for GNAT MAKE	96
6.4	Notes on the Command Line	96
6.5	How GNAT MAKE Works.....	97
6.6	Examples of GNAT MAKE Usage.....	98
7	Renaming Files Using GNAT CHOP	99
7.1	Handling Files with Multiple Units	99
7.2	Operating GNAT CHOP in Compilation Mode	99
7.3	Command Line for GNAT CHOP.....	100
7.4	Qualifiers for GNAT CHOP	100
7.5	Examples of GNAT CHOP Usage.....	102
8	Configuration Pragmas.....	103
8.1	Handling of Configuration Pragmas.....	103
8.2	The Configuration Pragmas Files.....	103
9	Handling Arbitrary File Naming Conventions	
	Using gnatname	105
9.1	Arbitrary File Naming Conventions.....	105
9.2	Running gnatname	105
9.3	Qualifiers for gnatname.....	105
9.4	Examples of gnatname Usage.....	106
10	GNAT Project Manager	109
10.1	Introduction	109
10.1.1	Project Files	109
10.2	Examples of Project Files.....	110
10.2.1	Common Sources with Different Qualifiers and Different Output Directories	110
	Source Files	111
	Specifying the Object Directory	111
	Specifying the Exec Directory	112
	Project File Packages.....	112
	Specifying Qualifier Settings	112
	Main Subprograms	113
	Source File Naming Conventions	113
	Source Language(s)	113
10.2.2	Using External Variables	113
10.2.3	Importing Other Projects	114
10.2.4	Extending a Project	115
10.3	Project File Syntax	116
10.3.1	Basic Syntax	117
10.3.2	Packages.....	117

10.3.3	Expressions	118
10.3.4	String Types	119
10.3.5	Variables	119
10.3.6	Attributes	120
10.3.7	Associative Array Attributes	122
10.3.8	case Constructions	122
10.4	Objects and Sources in Project Files	123
10.4.1	Object Directory	123
10.4.2	Exec Directory	123
10.4.3	Source Directories	124
10.4.4	Source File Names	124
10.5	Importing Projects	125
10.6	Project Extension	126
10.7	External References in Project Files	126
10.8	Packages in Project Files	127
10.9	Variables from Imported Projects	128
10.10	Naming Schemes	129
10.11	Library Projects	130
10.12	Qualifiers Related to Project Files	131
10.13	Tools Supporting Project Files	132
10.13.1	GNAT MAKE and Project Files	132
10.13.1.1	Qualifiers and Project Files	132
10.13.1.2	Project Files and Main Subprograms	134
10.13.2	The GNAT Driver and Project Files	135
10.14	An Extended Example	136
10.15	Project File Complete Syntax	138
11	Elaboration Order Handling in GNAT ...	141
11.1	Elaboration Code in Ada 95	141
11.2	Checking the Elaboration Order in Ada 95	143
11.3	Controlling the Elaboration Order in Ada 95	144
11.4	Controlling Elaboration in GNAT - Internal Calls	147
11.5	Controlling Elaboration in GNAT - External Calls	150
11.6	Default Behavior in GNAT - Ensuring Safety	151
11.7	Elaboration Issues for Library Tasks	153
11.8	Mixing Elaboration Models	158
11.9	What to Do If the Default Elaboration Behavior Fails ...	158
11.10	Elaboration for Access-to-Subprogram Values	162
11.11	Summary of Procedures for Elaboration Control	162
11.12	Other Elaboration Order Considerations	163

12	The Cross-Referencing Tools GNAT XREF and GNAT FIND	167
12.1	GNAT XREF Qualifiers	167
12.2	GNAT FIND Qualifiers	168
12.3	Project Files for GNAT XREF and GNAT FIND	170
12.4	Regular Expressions in GNAT FIND and GNAT XREF	172
12.5	Examples of GNAT XREF Usage	173
12.5.1	General Usage	173
12.6	Examples of GNAT FIND Usage	174
13	File Name Krunching Using GNAT KRUNCH..	177
13.1	About GNAT KRUNCH	177
13.2	Using GNAT KRUNCH	177
13.3	Krunching Method	177
13.4	Examples of GNAT KRUNCH Usage	179
14	Preprocessing Using GNAT PREPROCESS	181
14.1	Using GNAT PREPROCESS	181
14.2	Qualifiers for GNAT PREPROCESS	181
14.3	Form of Definitions File	182
14.4	Form of Input Text for GNAT PREPROCESS	182
15	The GNAT Run-Time Library Builder GNAT LIBRARY	185
15.1	Running GNAT LIBRARY	185
15.2	Qualifiers for GNAT LIBRARY	185
15.3	Example of GNAT LIBRARY Usage	185
16	The GNAT Library Browser GNAT LIST ...	187
16.1	Running GNAT LIST	187
16.2	Qualifiers for GNAT LIST	188
16.3	Example of GNAT LIST Usage	189
17	Finding Memory Problems with GNAT Debug Pool	191
18	Creating Sample Bodies Using GNAT STUB	193
18.1	Running GNAT STUB	193
18.2	Qualifiers for GNAT STUB	193

19	Reducing the Size of Ada Executables with GNAT ELIM	195
19.1	About GNAT ELIM	195
19.2	Eliminate Pragma	195
19.3	Tree Files	195
19.4	Preparing Tree and Bind Files for GNAT ELIM	196
19.5	Running GNAT ELIM	196
19.6	Correcting the List of Eliminate Pragmas	197
19.7	Making Your Executables Smaller	197
19.8	Summary of the GNAT ELIM Usage Cycle	197
20	Other Utility Programs	199
20.1	Using Other Utility Programs with GNAT	199
20.2	The GNAT STANDARD Utility Program	199
20.3	The External Symbol Naming Scheme of GNAT	199
20.4	Ada Mode for Glide	200
20.4.1	General Features:	200
20.4.2	Ada Mode Features That Help Understanding Code:	200
20.4.3	Glide Support for Writing Ada Code:	201
20.5	Converting Ada Files to html with gnathtml	201
20.6	Installing gnathtml	202
20.7	LSE	202
20.8	Profiling	202
21	Running and Debugging Ada Programs ..	205
21.1	The GNAT Debugger GDB	205
21.2	Running GDB	206
21.3	Introduction to GDB Commands	206
21.4	Using Ada Expressions	208
21.5	Calling User-Defined Subprograms	208
21.6	Using the Next Command in a Function	209
21.7	Breaking on Ada Exceptions	209
21.8	Ada Tasks	209
21.9	Debugging Generic Units	210
21.10	GNAT Abnormal Termination or Failure to Terminate	211
21.11	Naming Conventions for GNAT Source Files	212
21.12	Getting Internal Debugging Information	213
21.13	Stack Traceback	213
21.13.1	Non-Symbolic Traceback	213
21.13.1.1	Tracebacks From an Unhandled Exception	213
21.13.1.2	Tracebacks From Exception Occurrences	215
21.13.1.3	Tracebacks From Anywhere in a Program	216

21.13.2	Symbolic Traceback	217
21.13.2.1	Tracebacks From Exception Occurrences	217
21.13.2.2	Tracebacks From Anywhere in a Program	218
22	Compatibility with DEC Ada	221
22.1	Ada 95 Compatibility	221
22.2	Differences in the Definition of Package System	221
22.3	Language-Related Features	222
22.3.1	Integer Types and Representations	222
22.3.2	Floating-Point Types and Representations	222
22.3.3	Pragmas Float_Representation and Long_Float	223
22.3.4	Fixed-Point Types and Representations	224
22.3.5	Record and Array Component Alignment	224
22.3.6	Address Clauses	224
22.3.7	Other Representation Clauses	225
22.4	The Package STANDARD	225
22.5	The Package SYSTEM	226
22.6	Tasking and Task-Related Features	227
22.7	Implementation of Tasks in DEC Ada for OpenVMS Alpha Systems	227
22.7.1	Assigning Task IDs	228
22.7.2	Task IDs and Delays	228
22.7.3	Task-Related Pragmas	228
22.7.4	Scheduling and Task Priority	229
22.7.5	The Task Stack	229
22.7.6	External Interrupts	229
22.8	Pragmas and Pragma-Related Features	229
22.8.1	Restrictions on the Pragma INLINE	231
22.8.2	Restrictions on the Pragma INTERFACE	232
22.8.3	Restrictions on the Pragma SYSTEM_NAME ..	232
22.9	Library of Predefined Units	232
22.9.1	Changes to DECLIB	233
22.10	Bindings	233
22.10.1	Shared Libraries and Options Files	234
22.10.2	Interfaces to C	234
22.11	Main Program Definition	234
22.12	Implementation-Defined Attributes	235
22.13	Compiler and Run-Time Interfacing	235
22.14	Program Compilation and Library Management	236
22.15	Input-Output	239
22.16	Implementation Limits	240
22.17	Tools	241

23	Inline Assembler	243
23.1	Basic Assembler Syntax	243
23.2	A Simple Example of Inline Assembler	244
23.3	Output Variables in Inline Assembler	245
23.4	Input Variables in Inline Assembler	248
23.5	Inlining Inline Assembler Code	249
23.6	Other <code>asm</code> Functionality	250
23.6.1	The <code>Clobber</code> Parameter	250
23.6.2	The <code>Volatile</code> Parameter	251
23.7	A Complete Example	251
23.7.1	<code>Check_CPU</code> Procedure	252
23.7.2	<code>Intel_CPU</code> Package Specification	258
23.7.3	<code>Intel_CPU</code> Package Body	262
24	Performance Considerations	267
24.1	Controlling Run-Time Checks	267
24.2	Optimization Levels	267
24.3	Debugging Optimized Code	268
24.4	Inlining of Subprograms	269
24.5	Coverage Analysis	271
	GNU Free Documentation License	273
	ADDENDUM: How to use this License for your documents	279
	Index	281