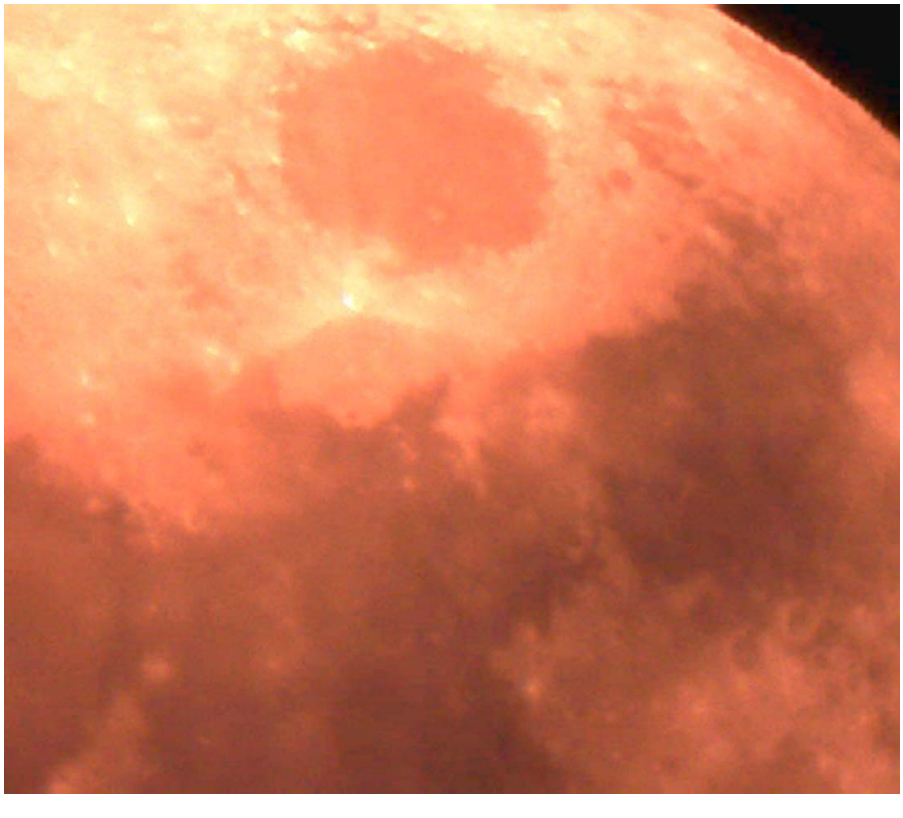
Using the Ada Compiler

XGC Users Guide



www.xgc.com

Using the Ada Compiler

XGC Users Guide

Order Number: XGC-ADA-UG-040422

XGC Software

London
UK
<www.xgc.com>

Using the Ada Compiler: XGC Users Guide

by Ada Core Technologies, Inc. and XGC Software

Publication date April 22, 2004

License

Contents

About This Guide

This guide describes the XGC Ada compiler, the command line options, and details of the user interface.

1. Audience

This guide is written for the experienced programmer who is already familiar with the Ada 95 programming language and with embedded systems programming in general. We assume some knowledge of the target computer architecture.

2. Related Documents

Getting Started with XGC Ada describes how to prepare and run a simple program. It also includes an introduction to more advanced topics.

The Assembler, Linker and Object Code Utilities, which describes the command line options and directives for the post-compiler tools.

Running and Debugging XGC Programs, which describes the instruction set simulator and symbolic debugger.

The *XGC Libraries* documents the library functions available with the XGC C and C++ compilers, and which may be called from Ada programs.

The *XGC Ada Reference Manual Supplement* documents the implementation-defined aspects of the Ada 95 programming language supported by the compiler.

3. Reader's Comments

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

You can send your comments in the following ways:

- Internet electronic mail: readers_comments@xgc.com

Please include the following information along with your comments:

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

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

4. Documentation Conventions

This guide uses the following typographic conventions:

`%`, `$`
: A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bash shell.

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

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

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

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

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

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

Mb/s
: This symbol indicates megabits per second.

MB/s
: This symbol indicates megabytes per second.

Ctrl+x

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

Chapter 1 *Using the Compiler*

This chapter discusses how to compile Ada programs using the gcc command, and the command line switches that can be used to control the behavior of the Ada compiler.

1.1. Compiling Programs

The first step in creating an executable program is to compile the units of the program using the **gcc** 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. XGC Ada compiles generic units when a client instantiates the generic, 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 messages (where *Filename* is the name of the file you compiled):

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

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

```
$ prefix-gcc -c [switches] filename
```

where *filename* is the name of the Ada file (usually having an extension `.ads` for a spec or `.adb` for a body). You specify the `-c` switch to tell **gcc** to compile, but not link, the file. The result of a successful compilation is an object file, which has the same name as the source file but an extension of `.o` and an Ada Library Information (ALI) file, which also has the same name as the source file, but with `.ali` as the extension. XGC Ada 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.

gcc 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, but need not be in your path. The **gcc** driver also calls the assembler and any other utilities needed to complete the generation of the required object-code files.

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

```
$ prefix-gcc -c x.adb y.adb z.c
```

calls **gnat1** (the Ada 95 compiler) twice to compile `x.adb` and `y.adb`, and `cc1` (the ANSI C compiler) once to compile `z.c`. The compiler generates three object files `x.o`, `y.o` and `z.o` and the two ALI files `x.ali` and `y.ali` from the Ada compilations. Any switches apply to all the files listed, except for `-gnatx` switches, which apply only to Ada compilations.

1.2. Switches for GCC

The **gcc** command accepts numerous switches to control the compilation process, which are fully described in this section.

`-c`

Compile. Always use this switch when compiling Ada programs.

Note that you may not use `gcc` without a `-c` switch to compile and link in one step. This is because the binder must be run, and currently `gcc` cannot be used to run the XGC Ada binder.

`-g`

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 `-g` switch if you plan on using the debugger or simulator.

`-I`*dir*

Direct XGC Ada to search the *dir* directory for source files needed by the current compilation (see Section 1.3, “Search Paths and the Run-Time Library” [23]).

`-I-`

Do not look for source files in the directory containing the source file named in the command line (see Section 1.3, “Search Paths and the Run-Time Library” [23]).

`-o` *file*

This switch is used in `gcc` to redirect the generated object file and its associated ALI file. Beware of this switch with XGC Ada, because it may cause the object file and ALI file to have different names which in turn may confuse the binder and the linker.

`-O[`*n*`]`

n controls the optimization level.

n = 0
No optimization

n = 1
Normal optimization, the default if you specify `-O` without an operand.

n = 2
Extensive optimization, the default

n = 3
Extensive optimization with automatic inlining. This applies only to inlining within a unit. See Section 1.2.10, “Subprogram Inlining Control” [20] for details on control of inter-unit inlining.

`-S`

Used in place of `-c` 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.

`-v`

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

`-V` *`ver`*

Execute *`ver`* version of the compiler. This is the `gcc` version, not the XGC Ada version.

`-Wuninitialized`

Generate warnings for uninitialized variables. You must also specify the `-O` switch (in other words, This switch works only if optimization is turned on).

`-gnata`
Assertions enabled. `Pragma Assert` and `pragma Debug` to be activated.

`-gnatb`
Generate brief messages to `stderr` even if verbose mode set.

`-gnatc`
Check syntax and semantics only (no code generation attempted).

`-gnate`
Error messages generated immediately, not saved up till end.

`-gnatE`
Full dynamic elaboration checks.

`-gnatf`
Full errors. Multiple errors per line, all undefined references.

`-gnatg`
Ada style checks enabled.

`-gnatic`
Identifier char set (*c*=1/2/3/4/8/p/f/n/w).

`-gnatje`
Wide character encoding method (*e*=n/h/u/s/e).

`-gnatkn`
Limit file names to *n* (1-999) characters (`k` = krunch).

`-gnatl`
Output full source listing with embedded error messages.

`-gnatmn`
Limit number of detected errors to *n* (1-999).

`-gnatn`
Activate inlining across unit boundaries for subprograms for which pragma `inline` is specified.

`-gnatN`
Activate inlining across unit boundaries for all subprograms (not just those for which pragma `inline` is specified. This is equivalent to using `-gnatn` and adding a pragma `inline` for every subprogram in the program.

`-fno-inline`
Suppresses all inlining, even if other optimization or inlining switches are set.

`-gnato`
Enable other checks, not normally enabled by default, including numeric overflow checking, and access before elaboration checks.

`-gnatp`
Suppress all checks.

`-gnatq`
Don't quit; try semantics, even if parse errors.

`-gnatr`
: Reference manual column layout required.

`-gnats`
: Syntax check only.

`-gnatt`
: Tree output file to be generated.

`-gnatu`
: List units for this compilation.

`-gnatv`
: Verbose mode. Full error output with source lines to `stdout`.

`-gnatwm`
: Warning mode (*m*=`s`, `e`, `l` for suppress, treat as error, elaboration warnings).

`-gnatzm`
: Distribution stub generation (*m*=r/s for receiver/sender stubs).

`-gnat83`
: Enforce Ada 83 restrictions.

`-gnat95`
: Standard Ada 95 mode

You may combine a sequence of XGC Ada switches into a single switch. For example, the specifying the switch

```
-gnatcfi3
```

is equivalent to specifying the following sequence of switches:

```
-gnatc -gnatf -gnati3
```

1.2.1. Error Message Control

The standard default format for error messages is called “brief format”. Brief format messages are written to `stdout` (the standard output 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 and the second integer is the column number. emacs can parse the error messages and point to the referenced character. The following switches allow control over the error message format:

`-gnatv`

The v stands for verbose. The effect is to write long-format error messages to `stdout`. The same program compiled with the `-gnatv` switch 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 switch is used the only source lines output are those with errors.

`-gnatl`

The `l` stands for list. This switch causes a full listing of the file to be generated. The output is 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 `-gnatv` or `-gnatl` switches and standard output is redirected, a brief summary is written to `stderr` (standard error) giving the number of error messages and warning messages generated.

`-gnatb`

The `b` stands for brief. This switch causes XGC Ada to generate the brief format error messages to `stdout` as well as the verbose format message or full listing.

`-gnatm`*n*

The `m` stands for maximum. *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 `-gnatm2` might yield

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

`-gnatf`

The `f` stands for full. Normally, the compiler suppresses error messages that are likely to be redundant. This switch causes all error messages to be generated. One particular effect is for

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 `-gnatf` switch 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
```

`-gnatq`

The `q` stands for quit (really "don't quit"). 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 switch tells XGC Ada 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.

`-gnate`

Normally, the compiler saves up error messages and generates them at the end of compilation in proper sequence. This switch (the “`e`” stands for error) causes error messages to be generated as soon as they are detected. The use of `-gnate` usually causes error messages to be generated out of sequence. Use this switch when the compiler terminates abnormally because of an internal error. In this case, the error messages may be lost. Sometimes abnormal terminations are the result of mis-handled error messages, so you may want to run with the `-gnate` switch to

determine whether any error messages were generated before the crash.

In addition to error messages, corresponding to illegalities as defined in the 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
```

XGC Ada detects a large number of situations which it considers 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 that XGC Ada issues warnings for (at least some of the time) are:

- Possible infinitely recursive calls
- Out of range values being assigned
- Possible order of elaboration problems
- Unreachable code
- Variables that are never assigned a value
- Variables that are referenced before being initialized
- Task entries that are never accepted
- Duplicate accepts for the same task entry in a select
- Objects that take too much storage

- Unchecked conversion with differing sizes
- Missing return statements in a function
- Incorrect 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 generic body
- Attempt to pack aliased components
- Out of bounds array subscript
- Wrong length on string assignment

Three switches are available to control the handling of warning messages:

`-gnatwu` (warn on unused entities)

This switch causes warning messages 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.

`-gnatwe` (treat warnings as errors)

This switch 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.

`-gnatws` (suppress warnings)

The "`s`" stands for suppress. This switch completely suppresses the output of all warning messages.

`-gnatwl` (warn on elaboration order errors)

This switch causes the generation of additional warning messages relating to elaboration issues. See the separate chapter on elaboration order handling for full details of the use of this switch.

`-gnatx`

Normally the compiler generates full cross-referencing information in the `.ali` file. This information is used by a number of tools, including **gnatfind** and **gnatxref**. The `-gnatx` switch suppresses this information. This saves some space and may slightly speed up compilation, but means that these tools cannot be used.

1.2.2. Debugging and Assertion Control

`-gnata`

The pragmas `Assert` and `Debug` normally have no effect and are ignored. This switch, 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_Error` 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.

1.2.3. Run-Time Checks

If you compile with the default options, XGC Ada 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 **gcc** switches refine this default behavior:

`-gnatp`

Suppress all run-time checks as though you have `pragma Suppress (all_checks)` in your source. Use this switch to improve the performance of the code at the expense of safety in the presence of invalid data or program bugs.

`-gnato`

Enables overflow checking for integer operations. This causes XGC Ada to generate slower and larger executable programs by adding code to check for both overflow and division by zero (resulting in raising `Constraint_Error` as required by Ada semantics). Note that the `-gnato` switch does not affect the code generated for any floating-point operations; it applies

only to integer operations. For floating-point, XGC Ada 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).

`-gnatE`

Enables dynamic checks for access before elaboration on subprogram calls and generic instantiations. For full details of the effect and use of this switch, see Chapter 1, *Using the Compiler* [1].

The setting of these switches 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.

1.2.4. Using GCC for Syntax Checking

`-gnats`

The `s` stands for syntax. Run XGC Ada in syntax checking only mode. For example, the command

```
$ prefix-gcc -c -gnats x.adb
```

compiles file `x.adb` in syntax-check-only mode. You can check a series of files in a single command, and can use wild cards to specify such a group of files. Note that you must specify the `-c` (compile only) flag in addition to the `-gnats` flag.

You may use other switches in conjunction with `-gnats`. In particular, `-gnatl` and `-gnatv` 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, XGC Ada 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 **gnatchop** utility (see Chapter 5, *Renaming Files with gnatchop* [55]).

1.2.5. Using GCC for Semantic Checking

```
-gnatc
```

The `c` stands for check. Cause the compiler to operate in semantic check mode, with full checking for all illegalities specified in the reference manual, but without generation of any source code (no object or ALI file generated).

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

- The needed source files must be accessible (see Section 1.3, “Search Paths and the Run-Time Library” [23]).
- Each file must contain only one compilation unit.
- The file name and unit name must match (see Section A.3, “File Naming Rules” [92]).

The output consists of error messages as appropriate. No object file or ALI file is generated. The checking corresponds exactly to the notion of legality in the Ada reference manual.

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

1.2.6. Compiling Ada 83 Programs

`-gnat83`

Although XGC Ada is primarily an Ada 95 compiler, it accepts this switch to specify that an Ada 83 mode program is being compiled. If you specify this switch, XGC Ada rejects Ada 95 extensions and applies Ada 83 semantics. It is not possible to guarantee this switch does a perfect job; for example, some subtle tests of pathological cases, such as are found in ACVC tests that have been removed from the ACVC suite for Ada 95, may not compile correctly. However for practical purposes, using this switch should ensure that programs that compile correctly under the `-gnat83` switch can be ported reasonably easily to an Ada 83 compiler. This is the main use of the switch.

With few exceptions (most notably the need to use <> on unconstrained generic formal parameters), it is not necessary to use the `-gnat83` switch when compiling Ada 83 programs, because, with rare and obscure 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.

`-gnat95`

This switch specifies normal Ada 95 mode, and cancels the effect of any previously given -gnat83 switch.

1.2.7. Style Checking

`-gnatr`

Normally, XGC Ada permits any code layout consistent with the reference manual requirements. This switch (“`r`” is for "reference manual") enforces the layout conventions suggested by the examples and syntax rules of the Ada Language Reference Manual. For example, an `else` must line up with an `if` and code in the `then` and `else` parts must be indented. The compile considers violations of the layout rules a syntax error if you specify this switch.

`-gnatg`

Enforces a set of style conventions that correspond to the style used in the XGC Ada source code. All compiler units are always compiled with the `-gnatg` switch specified.

You can find the full documentation for the style conventions imposed by `-gnatg` in the body of the package `Style` in the compiler sources (in the file `style.adb`).

You should not normally use the `-gnatg` switch. However, you *must* use `-gnatg` for compiling any language-defined unit, or for adding children to any language-defined unit other than `Standard`.

1.2.8. Character Set Control

`-gnatic`

Normally XGC Ada recognizes the *Latin-1* character set in source program identifiers, as described in the reference manual. This switch causes XGC Ada to recognize alternate character sets in identifiers. *c* is a single character 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

`p`
: IBM PC letters (code page 437) allowed in identifiers

`8`
IBM PC letters (code page 850) allowed in identifiers

`f`
Full upper-half codes allowed in identifiers

`n`
No upper-half codes allowed in identifiers

`w`
Wide-character codes allowed in identifiers

See Section A.2, “Foreign Language Representation” [88], for full details on the implementation of these character sets.

`-gnatje`

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

`n`
No wide characters allowed (default setting)

`h`
Hex encoding

`u`
Upper half encoding

`s`
Shift/JIS encoding

`e`
EUC encoding

See Section A.2.3, “Wide Character Encodings” [90] for full details on the these encoding methods.

1.2.9. File Naming Control

`-gnatk`*n*

Activates file name "crunching". *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 crunching.

For the source file naming rules, see Section A.3, “File Naming Rules” [92].

1.2.10. Subprogram Inlining Control

`-gnatn`

The `n` here is intended to suggest the first syllable of the word "inline". XGC Ada 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 switch. In the absence of this switch, XGC Ada 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 the compiler will access these bodies, creating an extra source dependency for the resulting object file, and where possible, the call will be in-lined. See Section D.3, “Inlining of Subprograms” [135] for further details on when inlining is possible.

`-gnatN`

This switch enforces a more extreme form of inlining across unit boundaries. It causes the compiler to proceed as though the normal (pragma) inlining switch was set, and to assume that there is a pragma `Inline` for every subprogram referenced by the compiled unit.

1.2.11. Auxiliary Output Control

`-gnatt`

Cause XGC Ada to write the internal tree for a unit to a file (with the extension `.atb` for a body or `.ats` for a spec). This is not normally required, but is used by separate analysis tools. Typically these tools do the necessary compilations automatically, so you should never have to specify this switch in normal operation.

`-gnatu`

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

1.2.12. Debugging Control

`-gnatd`*x*

Activate internal debugging switches. *x* is a letter or digit, or string of letters or digits, which specifies the type of debugging outputs desired. Normally these are used only for internal development or system debugging purposes. You can find full documentation for these switches in the body of the `Debug` unit in the compiler source file `debug.adb`.

One switch you may wish to use is `-gnatdg`, which causes a listing of the generated code in Ada source form. For example, all tasking constructs are reduced to appropriate run-time library calls. The syntax of this listing is close to normal Ada with the following additions:

`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).

1.3. Search Paths and the Run-Time Library

With the XGC Ada 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 looks in 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 `-I` switch given on the **gcc** command line, in the order given.

3. Each of the directories listed in the value of the `ADA_INCLUDE_PATH` environment variable. Construct this value exactly as the `PATH` environment variable: a list of directory names separated by colons.

4. The default location for the XGC Ada Run Time Library (RTL) source files. This is determined at the time XGC Ada is built and installed on your system.

Specifying the switch `-I-` 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 `-I` switch.

The compiler outputs its object files and ALI files in the current working directory. Caution: The object file can be redirected with the `-o` switch; however, **gcc** and **gnat1** have not been coordinated on this so the ALI file will not go to the right place. Therefore, you should avoid using the `-o` switch.

The packages `Ada`, `System`, and `Interfaces` and their children make up the XGC Ada 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 XGC Ada distribution provides a fourth hierarchy, consisting of child units of XGC Ada. This is a collection of generally useful routines. See the XGC Ada reference manual for further details.

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

1.4. 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 need 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 or 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 and performing a compilation never obsoletes anything. The only way you can obsolete something and require recompilations is to modify one of the dependent source files.
- There is no library as such, apart from the ALI files (see Section A.8, “The Ada Library Information Files” [98]., 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 (findable by the search-paths mechanism described above), or you will receive a fatal error message.

1.5. Examples

The following are some typical Ada compilation command line examples:

`$ prefix-gcc -c xyz.adb`
: Compile body in file `xyz.adb` with all default options.

`$ prefix-gcc -c -O2 -gnatp -gnata xyz-def.adb`
: Compile the child unit package in file `xyz-def.adb` with extensive optimizations, checks suppressed, and pragma `Assert`/`Debug` statements enabled.

`$ prefix-gcc -c -gnatc abc-def.adb`
: Compile the subunit in file `abc-def.adb` in semantic-checking-only mode.

Chapter 2

Binding with gnatbind

This chapter describes the XGC Ada binder, **gnatbind**, which is used to bind compiled XGC Ada objects. The **gnatbind** 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 satisfying the rules in Chapter 10 of the *Ada 95 Reference Manual*.

3. Generates a main program incorporating the given elaboration order. This program is a small C source file that must be subsequently compiled using the C compiler. 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 as comments in the generated C program, to be read by the **gnatlink** utility used to link the Ada application.

2.1. Running gnatbind

The form of the **gnatbind** command is

```
$ prefix-gnatbind [switches] mainprog.ali [switches]
```

where *mainprog*.adb is the Ada file containing the main program unit body. If no switches are specified, **gnatbind** constructs a C file whose name is `b_`*mainprog*`.c`. For example, if given the parameter "`hello.ali`", for a main program contained in file `hello.adb`, the binder output file would be `b_hello.c`.

When doing consistency checking, the binder takes any source files it can locate into consideration. 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 **gcc** command). If it can located this source file, the time stamps or source checksums must match. In other words, any ALI files mentioning 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 from 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 **gcc -c hello.adb** to compile the main program.

2. Enter **gcc -c p.ads** to compile package `P`.

3. Edit file `p.ads`.

4. Enter **gnatbind hello.ali**.

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 the spec and body of the main program. You need not normally be concerned with the contents of these files, but they are similar to the following:

```
pragma No_Run_Time;
package Ada_Main is
   Main_Priority : Integer;
   pragma Export (C, Main_Priority, "_main_priority");

   Queuing_Policy : constant Character := ' ';
   pragma Export (C, Queuing_Policy, "_queuing_policy");


   procedure adafinal;
   pragma Export (C, adafinal);

   procedure adainit;
   pragma Export (C, adainit);

   procedure Break_Start;
   pragma Export (C, Break_Start, "__break_start");

   function main
      return Integer;
   pragma Export (C, main, "main");

   type Version_32 is mod 2 ** 32;
   u00001 : constant Version_32 := 16#2B8FEC61#;
```

```
   u00002 : constant Version_32 := 16#26C73A9A#;
   u00003 : constant Version_32 := 16#6B5AC2B6#;
   u00004 : constant Version_32 := 16#37D0E260#;
   u00005 : constant Version_32 := 16#2299004B#;
   u00006 : constant Version_32 := 16#34601D38#;
   u00007 : constant Version_32 := 16#08D76389#;
   u00008 : constant Version_32 := 16#2359F9ED#;
   u00009 : constant Version_32 := 16#48E7D060#;
   u00010 : constant Version_32 := 16#34054F96#;

   pragma Export (C, u00001, "helloB");
   pragma Export (C, u00002, "text_ioS");
   pragma Export (C, u00003, "xgcS");
   pragma Export (C, u00004, "xgc__text_ioB");
   pragma Export (C, u00005, "xgc__text_ioS");
   pragma Export (C, u00006, "ada__exceptionsB");
   pragma Export (C, u00007, "ada__exceptionsS");
   pragma Export (C, u00008, "adaS");
   pragma Export (C, u00009, "systemS");
   pragma Export (C, u00010, "ada__io_exceptionsS");

end 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");

with System;

package body Ada_Main is

   procedure adainit is
   begin
      Main_Priority := -1;
      -- Ada'Elab_Spec;
         Ada.Exceptions'Elab_Spec;
      -- System'Elab_Spec;
      -- Ada.Exceptions'Elab_Body;
      -- Ada.Io_Exceptions'Elab_Spec;
      -- Xgc'Elab_Spec;
         Xgc.Text_Io'Elab_Spec;
      -- Xgc.Text_Io'Elab_Body;
      -- Text_Io'Elab_Spec;
      -- Hello'Elab_Body;
```

```
   end adainit;

   procedure adafinal is
   begin
      null;
   end adafinal;

   procedure Break_Start is
   begin
      null;
   end Break_Start;

   function main return Integer is

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

   begin
      adainit;
      Break_Start;
      Ada_Main_Program;
      adafinal;
      return 0;
   end;

-- BEGIN Object file/option list
   --   ./hello.o
   --   -L./
   --   -L/opt/erc32-ada-1.7/lib/gcc-lib/erc-coff/2.8.1/adalib/
   --   -lgnat
-- END Object file/option list

end Ada_Main;
```

The list of unsigned constants gives the version number information. Version numbers are computed by combining all the characters from the source file, omitting blanks and characters in comments. These values are used for implementation of the `Version` and `Body_Version` attributes.

Finally, a set of comments gives full names of all the object files required to be linked for the Ada component of the program. As seen in the previous example, 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.

2.2. Consistency-Checking Modes

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

`-s`

Require source files to be present. In this mode, the binder insists on being able to locate all source files that are referenced and checks their consistency. In normal mode, if a source file cannot be located it is simply ignored. If you specify this switch, a missing source file is an error.

`-x`

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 because it was last compiled and you specify the this switch, the binder will not detect that the object file is out of date with the source file. Note that this is the mode that is automatically used by **gnatmake** because in this case the checking against sources has already been performed by **gnatmake**.

2.3. Binder Error Message Control

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

`-v`

Verbose mode. In the normal mode, brief error messages are generated to `stderr`. If this switch is present, a header is written to `stdout` and any error messages are directed to `stdout`. All that is written to `stderr` is a brief summary message.

`-b`

Generate brief error messages to `stderr` even if verbose mode is specified. This is relevant only when used with the `-v` switch.

`-m`*n*

Limits the number of error messages to *n*, a decimal integer in the range 1-999. The binder terminates immediately if this limit is reached.

`-r`

Renames the generated main program from `main` to **gnat_main**. This is useful in the case of some cross-building environments, where the actual main program is separate from the one generated by **gnatbind**.

`-ws`

Suppress all warning messages.

`-we`

Treat any warning messages as fatal errors.

`-t`

Ignore time stamp errors. Any time stamp error messages are treated as warning messages. This switch essentially disconnects the normal consistency checking, and the resulting program may have undefined semantics if inconsistent units are present. *This means that `-t` should be used only in unusual situations, with extreme care.*

2.4. Elaboration Control

The following switches provide additional control over the elaboration order. See Appendix C, *Handling Elaboration Order* [111] for full details.

`-f`

Requests the binder to ignore suggestions from the compiler about implied `Elaborate_All` pragmas, and to use full reference manual semantics in an attempt to find a legal elaboration order, even if it seems likely that this order will cause an elaboration exception.

`-h`

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 switch 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

2.5. Output Control

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

`-e`

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 `stdout`.

`-l`

Output chosen elaboration order. The output is written to `stdout`.

`-o` *`file`*

Set name of output file to *`file`* instead of the normal `b_`*`prog`*`.c` default. You would normally give *`file`* an extension of `.c` because it will be a C source program.

`-c`

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.

2.6. Binding for Non-Ada Main Programs

In our description in this chapter so far we have assumed the main program is in Ada and the task of the binder is to generate a corresponding function `main` to pass control to this Ada main program. XGC Ada 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 XGC Ada. The following switch is used in this situation:

`-n`

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.

`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 `-n` switch is given, more than one ALI file may appear on the command line for **gnatbind**. 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 subprograms.

The binder takes the name of its output file from the first specified ALI file, unless overridden by the use of the `-o file`, The output file is a C source file, which must be compiled using the C compiler. This compilation occurs automatically as part of the **gnatmake** processing.

2.7. Summary of Binder Switches

The following are the switches available with **gnatbind**:

`-b`
: Generate brief messages to `stderr` even if verbose mode set.

`-c`
: Check only, no generation of binder output file.

`-e`
: Output complete list of elaboration-order dependencies.

`-aI`
: Specify directory to be searched for source file.

`-aO`
: Specify directory to be searched for ALI files.

`-I`
: Specify directory to be searched for source and ALI files.

`-I-`
Do not look for sources in the current directory where **gnatbind** was invoked, and do not look for ALI files in the directory containing the ALI file named in the **gnatbind** command line.

`-l`
Output-chosen elaboration order.

`-m`*`n`*
Limit number of detected errors to *`n`* (1-999).

`-n`
No main program.

`-o` *`file`*
Name the output file *`file`* (default is `b_`*`xxx`*`.c`).

`-s`
Require all source files to be present.

`-t`
Ignore time-stamp errors.

`-v`
Verbose mode. Write error messages, header, summary output to `stdout`.

`-w`*`x`*
Warning mode (*`x`*=s/e for suppress/treat as error)

`-x`
Exclude source files (check object consistency only).

You may obtain this listing by running the program **gnatbind** with no arguments.

2.8. Search Paths for gnatbind

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 **gcc** (see Section 1.3, “Search Paths and the Run-Time Library” [23]). For ALI files the directories searched are:

1. The directory containing the ALI file named in the command line, unless the switch `-I-` is specified.

2. All directories specified by `-I` switches on the **gnatbind** command line, in the order given.

3. Each of the directories listed in the value of the `ADA_OBJECTS_PATH` environment variable. Construct this value the same as the `PATH` environment variable: a list of directory names separated by colons.

4. The default location for the XGC Ada Run-Time Library (RTL) files, determined when XGC Ada was built and installed on your system.

In the binder the switch `-I` is used to specify both source and library file paths. Use `-aI` instead if you just want to specify source paths only and `-aO` if you want to specify library paths only. This means that for the binder `-I`*dir* is equivalent to `-aI`*dir* `\-aO\/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 XGC Ada Run-Time Library, together with the package XGC Ada and its children which contain a set of useful additional library functions provided by XGC Ada. 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 the assistance in using the RTL, a major use of search paths is in compiling sources from multiple directories. This can make development environments much more flexible.

2.9. Examples of gnatbind Usage

This section contains a number of examples of using the XGC Ada binding utility **gnatbind**.

gnatbind hello.ali
: The main program `Hello` (source program in `hello.adb`) is bound using the standard switch settings. The generated main program is `b_hello.c`. This is the normal, default use of the binder.

gnatbind main.ali -o mainprog.c -x -e
: The main program `Main` (source program in `main.adb`) is bound, excluding source files from the consistency checking, generating the file `mainprog.c`.

gnatbind -x main_program.ali -o mainprog.c
: This command is exactly the same as the previous example. Switches may appear anywhere in the command line, and single letter switches may be combined into a single switch.

gnatbind -n math.ali dbase.ali -o 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 **gnatbind** generates the file `control.c` containing the `adainit` and `adafinal` routines to be called before and after accessing the Ada subprograms.

Chapter 3 *Linking with gnatlink*

This chapter discusses **gnatlink**, a utility program used to link Ada programs and build an executable file. This program is basically a simple process which invokes the linker (via the **gcc** command) with a correct list of object files and library references. **gnatlink** 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.

*3.1. Running **gnatlink***

The form of the **gnatlink** command is

```
prefix-gnatlink [switches] mainprog[.ali] [non-Ada objects] [gcc options]
```

where *mainprog*`.ali` references the ALI file of the main program. The `.ali` extension of this file can be omitted. From this reference, **gnatlink** locates the corresponding binder file `b_`*mainprog*`.c` and, using the information in this file along with the list of non-Ada

objects and linker options, constructs a 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. This list may also include linker switches.

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

3.2. Switches for gnatlink

The following switches are available with the **gnatlink** utility:

-o *exec-name*

exec-name specifies an alternative name for the generated executable program. If this switch is omitted, the executable is called the name of the main unit. So **gnatlink try.ali** creates an executable called try.

-v

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

-g

The option to include debugging information causes the C bind file (in other words, b_*mainprog*.c) to be compiled with -g. In addition, the binder does not delete the b_*mainprog*.c and b_*mainprog*.o files. Without -g, the binder removes these files by default.

`-gnatlink` *name*

name is the name of the linker to be invoked. You normally omit this switch, in which case the default name for the linker is (**gcc**).

Chapter 4

Making Programs with gnatmake

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. Re-bind and re-link.
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 in-lined subprograms.

gnatmake 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 XGC Ada 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 **gnatmake**.

*4.1. Running **gnatmake***

The **gnatmake** command has the form:

```
$ prefix-gnatmake switches unit_or_file_name
```

The only required argument is *unit_or_file_name*, which specifies the compilation unit that is the main program. There are two ways to specify this:

- By giving the lowercase name of the compilation unit (`gnatmake` *`unit`*). In this case **gnatmake** will use the switches {`-a`I*dir*} and {`-I`*dir*} to locate the appropriate file.
- By giving the name of the source containing it (`gnatmake` *`[dir/]file`*`.adb`). If no relative or absolute directory *dir* is specified, the input source file will be searched for in the directory where **gnatmake** was invoked. **gnatmake** will not use the switches {`-a`I*dir*} and {`-I`*dir*} to locate the source file.

All **gnatmake** output (except when you specify `-M`) is to `stderr`. The output produced by the `-M` switch is send to `stdout`.

*4.2. Switches for **gnatmake***

You may specify any of the following switches to **gnatmake**:

`-a`

Consider all files in the make process, even the XGC Ada internal system files (for example, the predefined Ada library files), and also any locked files. Locked files are filed whose ALI file is write protected. By default, **gnatmake** does not check these files, because the assumption is that the XGC Ada 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 switch if you are working on XGC Ada itself. `-f` is also useful in conjunction with `-a` if you need to recompile an entire application, including run-time files, using special configuration pragma settings, such as a non-standard `Float_Representation` pragma. By default `gnatmake -a` compiles all XGC Ada internal files with **gcc -c -gnatg** rather than **gcc -c**.

`-c`

Compile only. Do not perform binding and linking. If the root unit specified by *`unit_or_file_name`* is not a main unit, this is the default. Otherwise **gnatmake** will attempt binding and linking unless all objects are up to date and the executable is more recent than the objects.

`-f`

Force recompilations. Recompile all sources, even though some object files may be up to date, but don't recompile predefined or XGC Ada internal files or locked files (files with a write protected ALI file), unless the `-a` switch is also specified.

`-j`*`n`*

Use *`n`* processes to carry out the (re)compilations. If you have a multiprocessor machine, compilations will occur in parallel. In the event of compilation errors, messages from various compilations might get interspersed (but **gnatmake** 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.

`-k`

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 **gnatmake** terminates.

`-M`

Check if all objects are up to date. If they are output the object dependences to `stdout` 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 `-aI` and `-I` switches. If you use **gnatmake -M -q** (see `-q` below), only the source file names, without relative paths, are output. If you just specify the `-M` switch, dependencies of the XGC Ada internal system files are omitted. This is typically what you want. If you also specify the `-a` switch, dependencies of the XGC Ada internal files are also listed. Note that dependencies of the objects in external Ada libraries (see switch `-aL`*`dir`* in the following list) are never reported.

`-i`

In normal mode, **gnatmake** compiles all object files and ALI files into the current directory. If the `-i` switch 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 **gnatmake** will automatically maintain and update this organization. If no ALI files are found on the Ada object path (See Section 1.3, “Search Paths and the Run-Time Library” [23]), 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. These are dummy files, so **gnatmake** will be forced to recompile the corresponding source

files, but it will be put the resulting object and ALI files in the location where it found the dummy file.

`-m`

Specifies that the minimum necessary amount of re-compilation be performed. Ignore 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 switch will tell gnatmake not to recompile files that depend on it (provided other sources on which these files depend have undergone no semantic modifications).

`-n`

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.

`-o` *`exec_name`*

Output executable name. The name of the final executable program will be *`exec_name`*. If the `-o` switch is omitted the default name for the executable will be the name of the input file in appropriate form for an executable file.

`-q`

Quiet. When this flag is not set, the commands carried out by **gnatmake** are displayed.

`-v`

Verbose. Displays the reason for all recompilations **gnatmake** decides are necessary.

gcc `switches`
: The switch `-g` or any uppercase switch (other than `-A`, or `-L`) or any switch that is more than one character is passed to **gcc** (e.g. `-O`, `-gnato`, etc.)

Source and library search path switches:

`-aI`*`dir`*

: When looking for source files also look in directory *`dir`*. The order in which source files search is undertaken is described in Section 1.3, “Search Paths and the Run-Time Library” [23].

`-aL`*`dir`*

: Consider *`dir`* as being an externally provided Ada library. Instructs **gnatmake** 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`*. You still need to specify the location of the specs for these units by using the switches `-aI`*`dir`* or `-I`*`dir`*. Note: this switch is provided for compatibility with previous versions of **gnatmake**. The easier method of causing standard libraries to be excluded from consideration is to write protect the corresponding ALI files.

`-aO`*`dir`*

: When searching for library and object files, look in directory *`dir`*. The order in which library files are searched is described in Section 2.8, “Search Paths for gnatbind” [37].

`-A`*`dir`*

: Equivalent to `-aL`*`dir`* `-aI`*`dir`*.

`-I`*`dir`*

: Equivalent to `-aO`*`dir`* `-aI`*`dir`*.

`-I-`

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 **gnatmake** was invoked.

`-L`*`dir`*

Add directory *`dir`* to the list of directories in which the linker will search for libraries. This is equivalent to `-largs -L`*`dir`*.

*4.3. Mode switches for **gnatmake***

The mode switches allow the inclusion of switches to be passed on to the compiler, binder or linker. The effect of a mode switch is to cause all subsequent switches up to the end of the switch list, or up to the next mode switch, to be interpreted as switches to be passed on to the designated component.

`-cargs` *`switches`*

Compiler switches. Here *`switches`* is a list of switches that are valid switches for **gcc**. They will be passed on to all compile steps performed by **gnatmake**.

`-bargs` *`switches`*

Binder switches. Here *`switches`* is a list of switches that are valid switches for **gcc**. They will be passed on to all bind steps performed by **gnatmake**.

`-largs` *`switches`*

Linker switches. Here *`switches`* is a list of switches that are valid switches for **gcc**. They will be passed on to all link steps performed by **gnatmake**.

4.4. Notes on the Command Line

This section contains some additional useful notes on the operation of the **gnatmake** command.

- If **gnatmake** finds no ALI files, it recompiles the main program and all other units required by the main program. This means that **gnatmake** can be used for the initial compile, as well as during the re-edit development cycle.
- If you enter `gnatmake` *`file`*`.adb`, where *`file`*`.adb` is a subunit or body of a generic unit, **gnatmake** recompiles *`file`*`.adb` (because it finds no ALI) and stops, issuing a warning.
- In **gnatmake** the switch `-I` is used to specify both source and library file paths. Use `-aI` instead if you just want to specify source paths only and `-aO` if you want to specify library paths only.
- **gnatmake** examines both an ALI file and its corresponding object file for consistency. If an ALI is more recent than its corresponding object, or the object is missing, the corresponding source will be recompiled. Note that **gnatmake** expects an ALI and the corresponding object file to be in the same directory.
- **gnatmake** 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 `-f` switch will not recompile these files unless `-a` is also specified.
- **gnatmake** 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

```
$ prefix-gnatmake -aIinclude-dir  -aLobj-dir  main
```

- Using **gnatmake** along with the `-s (minimal re-compilation)` switch provides an extremely powerful tool: you can freely 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 so obsolescence of debugging information cannot go unnoticed.

4.5. How gnatmake Works

Generally **gnatmake** 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 **gnatmake** 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.

gnatmake works by first checking if the specified main unit is up to date. If so, no compilations is required for the main unit. If not, **gnatmake** 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 recursively applies the above procedure test on all these files.

This process ensures that **gnatmake** 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 XGC Ada to compile in any order is critical in allowing an order of compilation to be chosen that guarantees that **gnatmake** will recompute a correct set of new dependencies if necessary.

4.6. Examples of gnatmake Usage

prefix`-gnatmake 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`.

prefix`-gnatmake -q Main_Unit -cargs -O2 -bargs -l`
: 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. **gnatmake** will operate in quiet mode, not displaying commands it is executing.

Chapter 5

Renaming Files with gnatchop

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

5.1. Handling Files with Multiple Units

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

The **gnatchop** utility allows both of these rules to be relaxed, allowing XGC Ada to process files which contain multiple compilation units and files with arbitrary file names. The approach used by **gnatchop** is to read the specified file and generate one or more output files, containing one unit per file and with proper file names as required by XGC Ada.

If you want to permanently restructure a set of foreign files so that they match the XGC Ada rules and do the remaining development using the XGC Ada structure, you can simply use **gnatchop** 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 **gnatchop** each time you compile, regarding the source files that it writes as temporary files that you throw away.

5.2. Command Line for gnatchop

The **gnatchop** command has the form:

```
$ prefix-gnatchop switches 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 files, in normal XGC Ada format, concatenated together.

When run in default mode, **gnatchop** generates one output file in the current directory for each unit in the file. 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

```
$ prefix-gnatchop 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.

`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.

5.3. Switches for gnatchop

gnatchop recognizes the following switches:

`-k`*`mm`*

Limit generated file names to the specified number characters. This is useful if the resulting set of files is required to be inter-operable with systems which limit the length of file names.

`-r`

Generate `Source_Reference` pragmas. Use this switch if the output files are regarded as temporary and development is to be done in terms of the original unchopped file. This switch causes `Source_Reference` pragmas to be inserted into each of the generated files to refers 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 `-g` switch of **gcc** or **gnatmake** 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.

`-s`

Write a compilation script to `stdout` containing **gcc** commands to compile the generated files.

`-w`

Overwrite existing file names. Normally **gnatchop** regards it as a fatal error situation if there is already a file with the same name as a file it would otherwise output. This switch bypasses this check, and any such existing files will be silently overwritten.

5.4. Examples of gnatchop Use

gnatchop -w hello_s.ada src/files
Chops the source file `hello_s.ada`. The output files will be placed in the directory `src/files`, overwriting any files with matching names in that directory (no files in the current directory are modified).

gnatchop -s -r collect
Chops the source file `collect` into the current directory. A compilation script is also generated, and all output files have `Source_Reference` pragmas, so error messages will refer back to the file `collect` with proper line numbers.

gnatchop archive
Chops the source file `archive` into the current directory. One useful application of **gnatchop** is in sending sets of sources around, for example in email messages. The required sources are simply concatenated (for example, using a UNIX `cat` command), and then **gnatchop** is used at the other end to re-constitute the original file names.

Chapter 6

Cross-Referencing with gnatxref

The compiler generates cross-referencing information (unless you set the `-gnatx` switch), which is saved in the `.ali` files. This information indicates where in the source each entity is declared and referenced.

The two tools **gnatxref** and **gnatfind** 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 **gnatfind** is intended for locating definitions and/or references to a specified entity or entities, whereas **gnatxref** is oriented to generating a full report of all cross-references.

*6.1. Command Line of **gnatxref***

The **gnatxref** command line is of the following form:

```
$ prefix-gnatxref [switches] files
```

6.2. Switches for gnatxref

The following list contains the descriptions of the cross-referencing flags available with **gnatxref**:

`-a`
: If this switch is present, **gnatfind** and **gnatxref** 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 **gnatfind** and **gnatxref** much faster, and their output much smaller.

`-aI`*`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 **gnatmake**.

`-aO`*`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 **gnatmake**.

`-f`
: If this switch is set, the output file names will be preceded by their directory (if the file was found in the search path). If this switch is not set, the directory will not be printed.

`-g`
: If this switch is set, information is output only for library-level entities, ignoring local entities. The use of this switch may accelerate **gnatfind** and **gnatxref**.

`-I`*`DIR`*
: Equivalent to -aO*`DIR`* -aI*`DIR`*.

`-P`*`FILE`*
: Specify a project file to use. By default, **gnatxref** and **gnatfind** 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 `-aI` and `-aO`.

`-u`
Output only unused symbols. This may be really useful if you give your main compilation unit on the command line, as **gnatxref** will then display every unused entity and 'with'ed package.

`-v`
Instead of producing the default output, **gnatxref** will generate a `tags` file that can be used by vi. See examples of **gnatxref** usage for examples how to use this feature. The tags file is output to the standard output, thus you will have to redirect it to a file.

All these switches 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 **gnatxref -ag** instead of **gnatxref -a -g**.

6.3. Command Line of gnatfind

The **gnatfind** command line is of the following form:

```
$ prefix-gnatfind pattern[:sourcefile[:line[:column]]] [files]
```

where

pattern
An entity will be output only if it matches the regular expression found in *pattern* (See Section 6.3.1, “Regular expressions in gnatfind and gnatxref” [62]).

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 upper-lower case 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
gnatfind will look for references, bodies or declarations of symbols referenced in `sourcefile`, at line *line* and column *column*. See examples of **gnatfind** usage, 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.

files
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.

At least one of *sourcefile* or *pattern* has to be present on the command line.

6.3.1. Regular expressions in gnatfind and gnatxref

As specified in the section about **gnatfind**, the pattern can be a regular expression. Actually, there are to 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 lower-case characters in it (and at least one character

6.4. Example of the Use of gnatxref

test.adb

```
01 with Part1; -- unused
02 with Part2; use Part2;
03 procedure Test is
04
05    Thing : Number;
06    type Client is record
07       Number : Integer;
08       State : Boolean;
09    end record;
10    type Color is (Red, Green); -- unused
11    My_Client : Client;
12
13 begin
14    My_Client.Number := 1;
15    My_Client.State := True;
16    Thing := 20;
17    Thing := Thing + Thing;
18 end;
```

part1.ads

```
01 package Part1 is
02    type Useless is new Integer;
03 end;
```

part2.ads

```
01 package Part2 is
02    type Number is new Integer range 1 .. 1000;
```

```
03    The_Number : constant := 42;
04 end;
```

The result of invoking **gnatxref test** is:

```
$ prefix-gnatxref test
Part1 U part1.ads:1:9 {} {test.adb:1:6 }
Part2 U part2.ads:1:9 {} {test.adb:2:6 2:17 }
Number I part2.ads:2:9 {} {test.adb:5:12 }
Test U test.adb:3:11 {} {}
Thing I test.adb:5:4 {test.adb:16:4 17:4 } {test.adb:17:13 17:21 }
Client R test.adb:6:9 {} {test.adb:11:16 }
Number I test.adb:7:7 {} {test.adb:14:14 }
State E test.adb:8:7 {} {test.adb:15:14 }
Color E test.adb:10:9 {} {}
My_Client R test.adb:11:4 {test.adb:14:14 15:14 } {test.adb:14:4 15:4 }
```

Chapter 7

Shortening File Names with gnatkr

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, and also describes the **gnatkr** utility that can be used to determine the result of applying this shortening.

7.1. About gnatkr

The normal rule in using XGC Ada if default file names are used is that the file name must be derived from the unit name. The exact default rule is: Take the unit name and replace all dots by hyphens, except that if such a replacement occurs in the second character position of a name and the first character is one of a/g/s/i then replace the dot by the character ~ (tilde) instead of a minus. The reason for this special exception is to avoid clashes with the standard names for children of System, Ada and Interfaces, which use the prefixes -s -a and -i respectively.

The `-gnatk`*nn* switch of the compiler activates a "crunching" 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 63, the value of nn is usually set to 63, 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 might be appropriate. The default value of 63 for OpenVMS need not be specified.

The **gnatkr** utility can be used to determine the crunched name for a given file, when crunched to a specified maximum length.

7.2. Using *gnatkr*

The **gnatkr** command has the form

```
$ prefix-gnatkr name [length]
```

name can be an Ada name with dots or the XGC Ada 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`. **gnatkr** takes this to be an extension if there are no other dots in the name and the whole name is in lowercase.

length represents the length of the crunched name. The default without any argument given is 8 characters. A length of zero stands for unlimited, in other words no chop except for system files which are always 8.

The output is the crunched name. The output has an extension only if the original argument was a file name with an extension.

7.3. Crunching 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 lowercase for all letter, except that a hyphen in the second character position is replaced by a tilde if the first character is a, i, g, or s.

The extension is `.ads` for a specification and `.adb` for a body. Crunching does not affect the extension, but the file name is shorted 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 not 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 krunch 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  14
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 crunched to eight characters. The crunching 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 tilde, to avoid confusion with system file names.

As an example of this special rule, consider `ada-strings-wide_fixed.adb`, which gets crunched 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 crunching is used then it is your responsibility to ensure that no name clashes occur. The utility program **gnatkr** is supplied for conveniently determining the crunched name of a file.

*7.4. Examples of **gnatkr** Usage*

```
$ prefix-gnatkr very_long_unit_name.ads
velounna.ads
$ prefix-gnatkr very_long_unit_name.ads 6
vlunna.ads
```

```
$ prefix-gnatkr very_long_unit_name.ads 0
very_long_unit_name.ads
$ prefix-gnatkr grandparent-parent-child.ads
grparchi.ads
$ prefix-gnatkr Grandparent.Parent.Child
grparchi
```

Chapter 8

Preprocessing with gnatprep

The **gnatprep** utility provides a simple preprocessing capability for Ada programs. It is designed for use with XGC Ada, but is not dependent on any special features of XGC Ada.

*8.1. Using **gnatprep***

To call **gnatprep** use

```
$ prefix-gnatprep infile outfile deffile switches
```

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 XGC Ada, 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.

switches
: is an optional sequence of switches as described in the next section

*8.2. Switches for **gnatprep***

`-c`
: 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.

`-b`
: 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.

`-r`
: 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 switch forces `-b` if `-c` is not set, so that source line numbers are not modified.

`-s`
: Causes a sorted list of symbol names and values to be listed on the standard output file.

`-v`
: Causes gnatprep to output a copyright notice including the version number of gnatprep.

Note: if neither `-b` nor `-c` is present, then preprocessor lines and deleted lines are completely removed from the output, unless -r is specified, in which case -b is assumed.

8.3. 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.

8.4. Form of input text for ***gnatprep***

The input text may contain preprocessor conditional inclusion lines, and also general symbol substitution sequences. The preprocessor conditional inclusion commands have the form

```
#if [not] symbol [then]
lines
#elsif [not] symbol [then]
lines
#elsif [not] symbol [then]
lines
...
#else
```

```
lines
#end if;
```

For these Boolean tests, the symbol must have either the value True or False. If the value is True, then the corresponding lines are included, and if the value is False, they are excluded. It is an error to reference a symbol not defined in the symbol definitions file, or to reference a symbol that has 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 in column one, 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, that is it must be preceded only by spaces or horizontal tabs.

Symbol substitution is obtained by using the sequence:

```
$symbol
```

anywhere within a source line, except in a comment. 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.

Chapter 9

Browsing the Library with gnatls

gnatls 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.

*9.1. Running **gnatls***

The **gnatls** command has the form:

```
$ prefix-gnatls switches object_or_ali_file
```

The main argument is the list of object or `ali` files (see Section A.8, “The Ada Library Information Files” [98].) for which information is requested.

In normal mode, without additional option, **gnatls** 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:

```
$ prefix-gnatls *.o
./demo1.o            demo1            DIF demo1.adb
./demo2.o            demo2             OK demo2.adb
./hello.o            h1                OK hello.adb
./instr-child.o      instr.child      MOK instr-child.adb
./instr.o            instr             OK instr.adb
./tef.o              tef              DIF tef.adb
./text_io_example.o  text_io_example   OK text_io_example.adb
./tgef.o             tgef             DIF tgef.adb
```

The first line can be interpreted as follows: the main unit which is contained in object file `demo1.o` 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, MOK, DIF or NFP:

`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 re-compilation (this information is not currently used by **gnatmake** but may be in some future version of the system).

`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.

9.2. Switches for gnatls

gnatls recognizes the following switches:

`-a`

Consider all units, including those of the predefined Ada library. Especially useful with -d.

`-d`

List sources from which specified units depend on.

`-h`

Output the list of options.

`-o`

Only output information about object files.

`-s`

Only output information about source files.

`-u`

Only output information about compilation units.

`-aO`*dir*, `-aI`*dir*, `-I`*dir*, `-I-`
Source and Object path manipulation. Same meaning as the equivalent gnatmake flags. See Section 4.2, “Switches for **gnatmake**” [46]

`-v`

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.

9.3. Example of the Use of gnatls

Example of using the verbose switch. Note how the source and object paths are affected by the -I switch.

```
$ prefix-gnatls -v hello
GNATLS 1.7.5e Copyright 1997-2001 Free Software Foundation, Inc.

Source Search Path:
   <Current_Directory>
   /opt/.../lib/gcc-lib/prefix/2.8.1/adainclude/


Object Search Path:
   <Current_Directory>
   /opt/.../lib/gcc-lib/prefix/2.8.1/adalib/

./hello.o
```

```
   Unit =>
     Name   => hello
     Kind   => subprogram body
     Flags  => No_Elab_Code
     Source => hello.adb    unchanged
```

Examples of use of the dependency list. Note the use of the -s switch which gives a straight list of source file. This can be useful for building specialized scripts.

```
$ prefix-gnatls -d hello
./hello.o   hello
   OK hello.adb
   OK io.ads

$ prefix-gnatls -d -s -a hello

hello.adb
io.ads
/opt/.../lib/gcc-lib/prefix/2.8.1/adainclude/system.ads
/opt/.../lib/gcc-lib/prefix/2.8.1/adainclude/s-unstyp.ads
```

Chapter 10

Other Utility Programs

This chapter discusses some other utility programs that are included with XGC Ada.

10.1. Using Other Utility Programs With XGC Ada

The object files generated by XGC Ada are in standard system format and in particular the debugging information uses this format. This means programs generated by XGC Ada can be used with existing utilities that depend on these formats.

10.2. The gnatpsys Utility Program

Many of the definitions in package System are implementation dependent. Furthermore, although the source of the package System is available for inspection, it uses special attributes for parameterizing many of the critical values, so the source is not informative.

The **gnatpsys** utility is designed to deal with this situation. It is an Ada program that when it runs, dynamically determines the values of all the relevant parameters in System, and prints them out in the form of an Ada source listing for System that shows all the values that are of interest. This output is generated to `stdout`.

To determine the value of any parameter in package System, simply run **gnatpsys** 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 system.

10.3. The gnatpsta 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 **gnatpsta** utility is designed to deal with this situation. It is an Ada program that when it runs, 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 that shows all the values that are of interest. This output is generated to `stdout`.

To determine the value of any parameter in package Standard, simply run **gnatpsta** 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 system.

10.4. The External Symbol Naming Scheme of XGC Ada

In order to interpret the output from XGC Ada, 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 names are in all lower-case 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 *`var1_name`*, and the link name for *`Var2`* is *`var2_link_name`*.

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`*.

Appendix A *The Compilation Model*

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

A.1. Source Representation

Ada source programs are represented in standard text files, using Latin-1 coding. Latin-1 is the ASCII character set with additional characters used for representing foreign languages (see Section A.2, “Foreign Language Representation” [88] for support of international 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#

The end of physical lines is marked by any of the following sequences: LF, CR, CR-LF, or LF-CR. Standard UNIX-format files simply use LF to terminate physical lines. The other combinations are recognized to provide convenient processing for files 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# (**Ctrl**+**Z**) is also represented as signaling 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.

A.2. Foreign Language Representation

XGC Ada 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.

A.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. This includes extended

letters used by European languages, such as 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 of library unit `Ada.Characters.Latin_1`. 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.

A.2.2. Other Eight-Bit Codes

XGC Ada also supports several other eight-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.

IBM PC (code page 437)

This code page is the normal default for PCs in the USA. 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 lower case 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 lower case equivalences that are recognized, see the file `csets.adb` in the XGC Ada compiler sources. You will need to obtain a full source release of XGC Ada to obtain this file.

A.2.3. Wide Character Encodings

XGC Ada allows wide character codes to appear in character and string literals, and also optionally in identifiers, using the following possible encoding schemes:

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

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

Where `a`, `b`, `c`, `d` are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, ["A345"] is used to represent the wide character with code `16#A345#`. This scheme is compatible with use of the full Wide_Character set, and is also the method used for wide character encoding in the standard ACVC (Ada Compiler Validation Capability) test suite distributions.

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 to F) is represented as two bytes, `16#ab#` and `16#cd#`. The second byte may never be a format control character, but is not required to be in the upper half. This method can be also used for shift-JIS or EUC, where the internal coding matches the external coding.

Shift JIS Coding

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

EUC Coding

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

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.

A.3. File Naming Rules

The default file name is determined by the name of the unit 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 lowercase for all letters.

A special exception arises if the file name according to the above rules has one of the characters a,g,i, or s and the second character is a minus. In this case, the character /tilde/dollar sign/ is used in place of the The reason for this special exception is to avoid clashes with the standard names for children of System, Ada, Interfaces, and XGC Ada, which use the prefixes -s -a -i and -g respectively.

The extension is `.ads` for a spec and `.adb` for a body. The following table 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 XGC Ada are to be used on operating systems with limited file name lengths. See Section 7.2, “Using **gnatkr**” [68].

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, or alternatively you can specify the exact file names that you want to be used, as described in the next section.

A.4. Using Other File Names

In the previous section, we have described the default rules used by XGC Ada to determine the file name in which a given unit resides. It is often convenient to follow these default rules, and if you do then 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 are used. XGC Ada allows arbitrary file names to be used via the Source_File_Name pragma. The form of this pragma is as shown in the following examples:

```
pragma Source_File_Name (My_Utilities.Stacks,
  Spec_File_Name => "myutilst_a.ada");
pragma Source_File_name (My_Utilities.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 must have an identifier which indicates if the file name is for the spec

or the body, and the file name itself is given as a static string constant.

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. See Appendix B, *Handling of Configuration Pragmas* [109] for more details on how the **gnat.adc** file is created and used. XGC Ada allows completely arbitrary file names to be specified using the source file name pragma. However, if the file name specified has an extension other than `.ads` `.adb` or `.ada` it is necessary to use a special syntax when compiling the file. The name in this case must be preceded by the special sequence -x followed by a space, as in:

```
$ prefix-gcc -c -x peculiar_file_name.sim
```

If **gnatmake** is used, then it handles non standard file names automatically. One special case arises if the main unit has a non-standard file name, in this case, the **gnatmake** argument must be this non-standard file name. It is not possible to use the normal unit name form of the **gnatmake** command in this case.

A.5. Naming of XGC Ada Source Files

If you want to examine the workings of the XGC Ada 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 standard. 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`.

- All files prefixed with `exp` perform AST normalization and expansion, using the same numbering scheme. 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 XGC Ada run-time routines).

A.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 completely independently from their parent units (though, of course, the spec of the parent unit must be present).

- 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, because XGC Ada always handles generic instantiation using 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 `.o` as usual.

You may wish to compile other files for the purpose of checking 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 correct before compiling clients of this spec, because such compilations will fail if there is an error in the spec.

XGC Ada provides the 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 `-gnatc` switch.

A.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 `-gnatn` switch, the object file depends on the file containing the body of this subprogram as well as on the file containing the spec. Similarly if the `-gnatN` switch is used, then the unit is dependent on all body files.

- The object file for a parent unit depends on all its subunit body files.

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 larger

because of the inclusion of 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. The determination of the necessary recompilations may be done automatically using **gnatmake**.

A.8. The Ada Library Information Files

Each compilation actually generates two output files. The first of these is the normal object file with a `.o` extension. The second is a text file containing the dependency information file. It has the same name but with an `.ali` extension. This file is known as the Ada Library Information (ALI) file.

You normally need not be concerned with the contents of this file, but this section is included in case you want to understand how these files are being used. Each ALI file consists of a series of lines of the form:

```
Key_Character parameter parameter ...
```

The first two lines in the file identify the library output version and `Standard` version. These are required to be consistent across the entire set of compilation units in your program.

```
V "xxxxxxxxxxxxxxxx"
```

This line indicates the library output version, as defined in **gnatvsn.ads**. It ensures that separate object modules of a program are consistent. It must be changed if anything changes that would affect successful binding of modules compiled separately. Examples of such changes are modifications in the format of the library information described in this package, modifications to calling sequences, or to the way data is represented.

```
S "xxxxxxxxxxxxxxxx"
```

This line contains information regarding types declared in packages `Standard` as stored in `Gnatvsn.Standard_Version`. The purpose (on systems where, for example, the size of `Integer` can be set by command line switches) is to ensure that all units in a program are compiled with a consistent set of options.

```
M type [priority]
```

This line is present only for a unit that can be a main program. *`type`* is either `P` for a parameterless procedure or `F` for a function returning a value of integral type. The latter is for writing a main program that returns an exit status. *`priority`* is present only if there was a valid pragma `Priority` in the corresponding unit to set the main task priority. It is an unsigned decimal integer.

```
F x
```

This line is present if a pragma Float_Representation or Long_Float is used to specify other than the default floating-point format. This option applies only to implementations of XGC Ada for the Digital Alpha Systems. The character x is 'I' for IEEE_Float, 'G' for VAX_Float with Long_Float using G_Float, and 'D' for VAX_Float for Long_Float with D_Float.

```
P L=x Q=x T=x
```

This line is present if the unit uses tasking directly or indirectly, and has one or more valid xxx_Policy pragmas that apply to the unit. The arguments are as follows

```
L=x (locking policy)
```

This is present if a valid Locking_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. “`C`” for Ceiling_Locking).

```
Q=x (queuing policy)
```

This is present if a valid Queuing_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. "P" for Priority_Queuing).

```
T=x (task_dispatching policy)
```

This is present if a valid Task_Dispatching_Policy pragma applies to the unit. The single character indicates the policy in effect (e.g. "F" for FIFO_Within_Priorities).

Following these header lines, a set of information lines appears for each compilation unit that appears in the corresponding object file. In particular, when a package body or subprogram body is compiled there will be two sets of information, one for the spec and one for the body, with the entry for the body appearing first. This is the only case in which a single ALI file contains more than one unit. Note that subunits do not count as compilation units for this purpose, and generate no library information, because they are inlined. The lines for each compilation unit have the following form:

```
U unit-name source-name version [attributes]
```

This line identifies the unit to which this section of the library information file applies. *unit-name* is the unit name in internal format, as described in package Uname, and *source-file* is the name of the source file containing the unit.

version is the version given as eight hexadecimal characters with lowercase letters. This value is a hash code that includes contributions from the time stamps of this unit and all its semantically dependent units.

The optional *attributes* are a series of two-letter codes indicating information about the unit. They give the nature of the unit, and also the information provided by categorization pragmas.

EB
: Unit has pragma Elaborate_Body.

NE
: Unit has no elaboration routine. All subprogram specs are in this category, as are subprogram bodies if access before elaboration checks are being generated. Package bodies and specs may or may not have NE set, depending on whether or not elaboration code is required.

PK
: Unit is a package, rather than a subprogram.

PU
: Unit has pragma Pure.

PR
: Unit has pragma Preelaborate.

RC
: Unit has pragma Remote_Call_Interface.

RT
: Unit has pragma Remote_Types.

SP
: Unit has pragma Shared_Passive.

SU
: Unit is a subprogram, rather than a package.

The attributes may appear in any order, separated by spaces. Another line in the ALI file has the following form:

```
W unit-name [source-name lib-name [E] [EA] [ED]]
```

One of these lines is present for each unit mentioned in an explicit `with` clause by the current unit. *`unit-name`* is the unit name in internal format. *`source-name`* is the file name of the file that must be compiled to compile that unit (usually the file for the body, except for packages that have no body). *`lib-name`* is the file name of the library information file that contains the results of compiling the unit. The `E` and `EA` parameters are present if pragma `Elaborate` or pragma `Elaborate_All`, respectively, apply to this unit. `ED` is used to indicate that the compiler has determined that a pragma `Elaborate_All` for this unit would be desirable. See Appendix C, *Handling Elaboration Order* [111] for details on the use of the ED parameter.

Following the unit information is an optional series of lines that indicate the usage of pragma `Linker_Options`. For each appearance of pragma `Linker_Options` in any of the units for which unit lines are present, a line of the form

```
L string
```

appears where *`string`* is the string from the pragma enclosed in quotes. Within the quotes, the following can occur:

- 7-bit graphic characters other than " or {
- "" (indicating a single " character)
- {hh} indicating a character whose code is hex hh

For further details, see `Stringt.Write_String_Table_Entry` in the file `stringt.ads`. Note that wide characters in the form {hhhh} cannot be produced, because `pragma Linker_Option` accepts only `String`, not `Wide_String`.

Finally, at the end of the ALI file is a series of lines that indicate the source files on which the compiled units depend. This is used by the binder for consistency checking and look like:

```
D source-name time-stamp [comments]
```

where *comments*, if present, must be separated from the time stamp by at least one blank. Currently this field is unused.

Blank lines are ignored when the library information is read, and separate sections of the file are separated by blank lines to ease readability. Extra blanks between fields are also ignored.

A.9. Representation of Time Stamps

All compiled units are marked with a time stamp, which is derived from the source file. The binder uses these time stamps to ensure consistency of the set of units that constitutes a single program. Time stamps are twelve-character strings of the form *YYMMDDHHMMSS*. Each two-character field has the following meaning:

`YY`
: year (2 low order digits)

`MM`
: month (2 digits 01-12)

`DD`
: day (2 digits 01-31)

`HH`
: hour (2 digits 00-23)

`MM`
: minutes (2 digits 00-59)

`SS`
: seconds (2 digits 00-59)

Time stamps may be compared lexicographically (in other words, the order of Ada comparison operations on strings) to determine which is later or earlier. However, in normal mode, only equality comparisons have any effect on the semantics of the library. Later/earlier comparisons are used only for determining the most informative error messages to be issued by the binder. Note that this means that despite the fact that only two digits are used for the

year, there are no “year 2000” problems with this representation choice.

The time stamp is the actual stamp stored with the file without any adjustment resulting from time zone comparisons. This avoids problems in using libraries across networks with clients spread across multiple time zones, but may mean the time stamp will differ from that displayed in a directory listing. For example, in UNIX systems, file time stamps are stored in Greenwich Mean Time (GMT), but the **ls** command displays local times.

A.10. Binding an Ada Program

When using languages such as C and C++, the only remaining step in building an executable program once the source files have been compiled is linking the object modules together. This means 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 in 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 not possible to build a program containing these two clients. These rules are enforced by the XGC Ada binder, which also determines an elaboration order consistent with the Ada rules.

The XGC Ada binder is run after all the object files for a program have been compiled. It is given the name of the main program unit, and from this it determines the set of units required by the program, 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 C, that contains calls to the required elaboration procedures, followed by a call to the main program. This C program is compiled using the C compiler to generate the object file for the main program. The name of the C file is `b_`*xxx*`.c` 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.

A.11. Mixed Language Programming

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

1. Compile the Ada units to produce a set of object files and ALI files.

2. Compile the other language files to generate object files.

3. Run the Ada binder on the Ada main program.

4. Compile the Ada main program.

5. Link the Ada main program, Ada objects and other language objects.

If the main program is in some language other than Ada, you use a special option of the binder to generate callable routines to initialize and finalize the Ada units. You must insert calls to these routines in the main program, or some other appropriate point. 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 places the initialization and finalization subprograms into file `b_`*`xxx`*`.c` instead of the main program.

A.12. Comparison of XGC Ada With C/C++ Compilation Model

The XGC Ada 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 no 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 putting together inconsistent programs is outside the compiler, in a make file, 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 **gnatbind** might complain there was no valid order of elaboration, a C++ compiler would simply construct a program that malfunctioned at run time.

A.13. Comparison of XGC Ada With Ada Library Model

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 Languages Reference Manual. If you have not used such a system, please go on to the next section.

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

Appendix B

Handling of Configuration Pragmas

In Ada 95, configuration pragmas include those pragmas described as being configuration pragmas in the Ada 95 Reference Manual, as well as implementation dependent pragmas that are configuration pragmas. See the individual descriptions of pragmas in the XGC Ada Reference Manual for details on these additional XGC Ada-specific configuration pragmas. Most notably, the pragma `Source_File_Reference`, which allows specifying non-default names for source files, is a configuration pragma.

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.

B.1. The gnat.adc File

In the case of XGC Ada, 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**,

and if this file is present, then it is expected to contain one or more configuration pragmas that will be applied to the current compilation.

Configuration pragmas may be entered into the `gnat.adc` file either by running **gnatchop** on a source file that consists only of configuration pragmas, or, usually more convenient in practice, by direct editing of the `gnat.adc` file, which is a standard format source file.

Appendix C

Handling Elaboration Order

This Appendix describes the handling of elaboration code in Ada 95, and in XGC Ada, and in particular discusses how the order of elaboration can be controlled, automatically or as specified explicitly by the program.

C.1. Elaboration Code in Ada 95

Ada 95 provides rather general mechanisms for executing code at elaboration time, that is 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 is:

```
Power_Up_Latch : Boolean := Check_Power (High);
```

Package initialization code:

> Code between `begin` and `end` at the outer level of a package body is executed as part of the package body elaboration code.

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 is considered an inappropriate way to structure a program.

An important concern arises in the context of this code, which is that we have to be sure that it is elaborated in an appropriate order. What we have is lots of little sections of elaboration code, potentially one section of code 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`, that 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`.

Now we would never have any elaboration order problems 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.

One 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.

C.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 kind of ordering problems himself, and it is quite possible to write a program in which an incorrect elaboration order can give surprising results as a result of referencing variables before they are initialized as intended. Ada 95 is designed to be a safe language, so this approach is clearly not acceptable. 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 the elaboration order is incorrect, then an exception (`Program_Error`) is raised.

Elaboration control

Facilities are provided for the programmer to control the order of elaboration to prevent such exceptions from being raised.

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 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 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 here is that if the body has been elaborated, then any variables it references must have been elaborated, so by checking for the body being elaborated, we are guaranteed 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 is in fact safe to call. However, it really would not be right to rely on this, because it would mean that the caller was relying on details of the implementation in the body, which is something we always try to avoid in Ada.

To get an idea of how this might be implemented, consider the following model implementation. A Boolean variable is associated with each subprogram and generic unit. This variable is initially set to False, and is set to True when the body is elaborated. Every call or instantiation checks the variable, and raises `Program_Error` if the variable is False.

C.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. However, this is not sufficient. Although we certainly prefer an exception to getting the wrong results, we need ways of avoiding the exception. To achieve this, Ada 95 provides a number of features for controlling the order of elaboration, and 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 mm;
end x;
```

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 so that it is impossible for any call to any subprogram in the unit to result in an elaboration problem. This means that the compiler does not need to worry about the order of elaboration for 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 fierce restrictions on a unit, but the 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`. Now the standard rules require the spec of unit `A` to be elaborated before the with'ing unit, and given the pragma in `A`, we also know that the body of `A` will be elaborated before `B`, so 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 have 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, this 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 no elaboration order is possible.

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. But, as we have discussed, it is not always possible to use one of these three pragmas. So we also provide methods for clients to control the order of elaboration:

pragma Elaborate (unit)

This pragma is placed in the context clause, after a with statement, 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 you know that you will be making calls, directly or indirectly, at elaboration time to subprograms in a given 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 elaboration 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.

But 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 procedure 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, Preelaborate, or Elaborate_Body, then the client should have an 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 wrong
: 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 wrong
: 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 right
: One ore more acceptable elaboration orders exists, 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. Even if a program that does not follow this rule happens to be safe, this state of affairs may deteriorate silently as a result of maintenance changes.

C.4. Controlling Elaboration in XGC Ada - Internal Calls

In the case of internal calls, that is 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, and indeed XGC Ada 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 seen
        >>> 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 Y;
```

Note that in this particular case, it is probably the case that, because `One` does not access any global variables, the call really would be safe, but 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 elaboration 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, and no warning is generated, and no `Program_Error` exception will be raised. Things get a bit 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 XGC Ada 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" may occur before body is seen
        >>> warning: Program_Error may be raised at run time
        >>> 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 X;
```

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 on run-time flow of control in the general case. 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 till run time whether the incorrect call to A would actually occur, so `Program_Error` might or might not be raised. If XGC Ada felt more ambitious, it could 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 types given above will usually correspond to real errors, and should be examined carefully, and typically eliminated. In the rare case that a warning is bogus, it can be suppressed by any of the following methods:

- Compile with the `-gnatws` switch 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, XGC Ada 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.

C.5. Controlling Elaboration in XGC Ada - 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`), and the question is 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 you have a with for a package `X`, then you are assured that the spec of `X` is elaborated before you are, but you are not assured that the body of `X` is elaborated before you are. 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 react that the binder is being silly, 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 this:

```
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 arise only in connection with elaboration code, works fine. A rule that says that you must elaborate the body first of anything you `with` cannot work in this case (the body of `X` `with`'s `Y`, which means you want to elaborate the body of `Y` first,

but that `with`'s `X`, which means you want 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 XGC Ada binder will in fact always 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 kind can get into trouble, as we discussed in the previous sections, so XGC Ada provides a number of facilities for assisting the programmer in developing programs that are robust with respect to elaboration order.

C.6. Default Behavior in XGC Ada - Ensuring Safety

The default behavior in XGC Ada ensures elaboration safety. What XGC Ada does in its default mode is to automatically and implicitly implement the rule we previously suggested as the right approach. Let's restate the rule:

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`, `Preelaborate`, or `Elaborate_Body`, then the client should have an `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.

What XGC Ada does is to trace all calls that are potentially made from elaboration code, and put 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 XGC Ada is not guaranteed.

If it is important to guarantee portability, then the compilations should use the `-gnatwl` (warn on elaboration problems) switch. 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 really 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 this implicit pragma. However it is safer to include the pragma explicitly in the source. If this unit is compiled with the `-gnatwl` switch, 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 the missing pragmas.

C.7. What to do if the Default Elaboration Behavior Fails

If the binder cannot find an acceptable order, it outputs quite 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 -gnatwl
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 separate off 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 large complex legacy codes.

Perform dynamic checks
: If the compilations are done using the `-gnatE` (dynamic elaboration check) switch, then XGC Ada behaves in a quite different manner. Dynamic checks are generated for all calls that could possibly result in raising an exception. With this

switch, the compiler does not generate implicit `Elaborate_All` pragmas. The behavior then is exactly as specified in the 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 switch, it cannot be used selectively.

Suppress checks
: One difficulty with generating dynamic checks is that they generate a significant extra overhead at run time, both in space and time. If you are absolutely sure that your program cannot raise any elaboration exceptions, then you can use the `-f` switch for the **gnatbind** step, or `-bargs -f` if you are using **gnatmake**. This switch tells the binder to ignore any implicit `Elaborate_All` pragmas that were generated by the compiler, and suppresses any circularity messages that they cause. The resulting executable will work fine if there are no elaboration problems, but if there are some order of elaboration problems, then they will not be detected, and unexpected results may occur.

It is hard to generalize on which of these three 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 `-gnatE` or `-f` but the danger in either case is that, even if the XGC Ada binder finds a correct elaboration error free order, it may not always do so, and certainly a binder from another Ada compiler may not do so. A combination of testing and analysis (for which the warnings generated with the `-gnatwl` switch can be useful) must be used to ensure that the program is free of errors. One switch that is useful in this testing is the `-h (horrible elaboration order)` switch for **gnatbind**. Normally the binder tries to find an order that has the best chance of succeeding in avoiding elaboration problems. With this switch, the binder plays a kind of devil's advocate role, and tries to choose the order that has the best chance of failing. If your program works even with this switch, 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 succeeds 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 `-gnatE` switch, and then passes fine. Alternatively, the entire suite can be run using this switch. It is never wrong to run with the dynamic elaboration switch 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.

C.8. 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 we now have a situation where it is impossible at compile time to tell exactly which procedure is being called. This means that it is not possible to analyze the elaboration requirements statically at compile time in this case.

If at the time the access value is created, 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, then 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 time the access value is created, then any use of the access value must do a dynamic check, and this dynamic check will fail, raising a `Program_Error` exception if the body still has not been elaborated. XGC Ada will generate the necessary checks, and in addition, if the `-gnatwl` switch 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 has been elaborated, will also result in the raising of a `Program_Error` exception

C.9. Summary of Procedures for Elaboration Control

First, compile your program with the default options, using none of the special elaboration control switches. 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 `-gnatwl` switch 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 `-gnatE` switch to generate dynamic elaboration checks, or, if you are sure there really are no elaboration problems, use the `-f` switch for the binder to cause it to ignore implicit `Elaborate_All` pragmas generated by the compiler.

Appendix D

Performance Considerations

The XGC Ada compiler provides a number of options that allow a trade off between

- performance of the generated code;
- speed of compilation;
- minimization of dependencies and recompilation;
- and the degree of run-time checking.

The defaults if no options are selected are aimed at improving the performance of the generated code:

- optimization level 2
- 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.

D.1. Controlling Run-time Checks

By default, XGC Ada produces all run-time checks except arithmetic overflow checking for integer operations (including division by zero) and checks for access before elaboration on subprogram calls.

Two gnat switches, `-gnatp` and `-gnato` allow this default to be modified. See Section 1.2.3, “Run-Time Checks” [14].

Our experience is that the default is suitable for most development purposes.

We treat integer overflow and elaboration checks 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 the setting of the switches 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.

D.2. Optimization Levels

The default is optimization set to -O2. This results in the best optimization for most applications. Where debugging is made difficult because of optimizations, the you can use the `-O0` switch. In this case XGC Ada makes absolutely no attempt to optimize, and the generated programs are considerably larger and slower.

`-O0`
: no optimization

`-O1`
: medium level optimization

`-02`
: full optimization (the default)

`-03`
: full optimization, and also attempt automatic inlining of small subprograms within a unit (see Section D.3, “Inlining of Subprograms” [135]).

The penalty in compilation time, and the 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 the case with some other compiler systems, **gcc** 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.

D.3. 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 `-01`.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else that **gcc** 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 `-03`) 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 `-O1`.
- The called subprogram is suitable for inlining: It must be small enough and not contain nested subprograms or anything else **gcc** 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 `-gnatn` switch is used in the **gcc** command line.

Note that specifying the `-gnatn` switch 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 `-gnatn` switch specified), the compilation of the subprogram `Main` 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 `-gnatn` switch 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 the spec. This means that if the 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 **gcc**.

Note The `-fno-inline` switch can be used to prevent all inlining. This switch overrides all other conditions and ensures that no inlining occurs. The extra dependencies resulting from `-gnatn` will still be active, even if this switch is used to suppress the resulting inlining actions.

Index

Symbols

A

B

C