GPR Reference Manual

Release 27.0w

AdaCore

Jun 10, 2026

CONTENTS

1 Introduction **3**
- 1.1 What is GPR? . . . 3
- 1.2 Tools covered . . . 3
- 1.3 Document structure . . . 4
- 1.4 LibGPR2 . . . 4
- 1.5 Related documents . . . 4

2 Project File Language **5**
- 2.1 Overview . . . 5
- 2.2 Lexical Elements . . . 5
- 2.3 Project File Structure . . . 6
- 2.4 Declarations . . . 8
- 2.5 Values . . . 8
- 2.6 Built-in Functions . . . 9
- 2.7 Expressions . . . 11
- 2.8 Typed String Declaration . . . 12
- 2.9 Variables . . . 12
- 2.10 Attribute Declarations . . . 13
- 2.11 Packages . . . 14
- 2.12 Case Constructions . . . 16

3 Project Tree **17**
- 3.1 Project tree structure . . . 17
- 3.2 The `with` clause . . . 17
- 3.3 The `limited with` clause . . . 19

4 Project Kinds **21**
- 4.1 Formal syntax . . . 21
- 4.2 Standard Project . . . 22
- 4.3 Abstract Project . . . 22
- 4.4 Library Project . . . 23
- 4.5 Aggregate Project . . . 24
- 4.6 Aggregate Library Project . . . 25
- 4.7 Configuration Project . . . 26

5 Project Extension **27**
- 5.1 Simple Extension . . . 27
- 5.2 Extension Hierarchies . . . 28
- 5.3 `extends all` . . . 28
- 5.4 Restriction . . . 28

6 Source Resolution **29**
6.1 Source discovery . . . 29
6.2 Basename uniqueness . . . 29
6.3 Extending projects and source shadowing . . . 29
6.4 `extends all` . . . 30
6.5 Multi-unit Ada sources . . . 30

7 Attributes **31**
7.1 Project Level Attributes . . . 32
7.2 Package Binder Attributes . . . 37
7.3 Package Builder Attributes . . . 38
7.4 Package Clean Attributes . . . 38
7.5 Package Compiler Attributes . . . 39
7.6 Package Gnatls Attributes . . . 41
7.7 Package Install Attributes . . . 41
7.8 Package Linker Attributes . . . 42
7.9 Package Naming Attributes . . . 44

8 Knowledge Base **47**
8.1 Embedded KB . . . 47
8.2 KB structure . . . 47
8.3 Run-time KB selection . . . 47
8.4 Customizing the KB . . . 48
8.5 KB validation . . . 48

9 Common Command-Line Options **49**
9.1 Project file . . . 49
9.2 Project and configuration switches . . . 49
9.3 Output and diagnostics . . . 50

10 Out-of-Tree Builds **51**

11 GPRbuild Reference **53**
11.1 Build Engine Selection . . . 53
11.2 Command Line . . . 53
11.3 Switches . . . 54
11.4 Build Execution . . . 56
11.5 Project package . . . 57
11.6 Exit Codes . . . 57

12 GPRconfig Reference **59**
12.1 Command Line . . . 59
12.2 Switches . . . 59
12.3 Interactive mode . . . 61
12.4 Configuration file format . . . 61
12.5 Exit Codes . . . 61

13 GPRclean Reference **63**
13.1 Command Line . . . 63
13.2 Switches . . . 63
13.3 Project package . . . 64
13.4 What is cleaned . . . 64
13.5 Exit Codes . . . 64

14 GPRinstall Reference **65**

14.1 Command Line . . . 65
14.2 Operating modes . . . 65
14.3 Switches . . . 65
14.4 Installation layout . . . 67
14.5 Project package . . . 68
14.6 Exit Codes . . . 68

15 GPRname Reference **69**
15.1 Command Line . . . 69
15.2 Switches . . . 69
15.3 Generated files . . . 71
15.4 Exit Codes . . . 71

16 GPRls Reference **73**
16.1 Command Line . . . 73
16.2 Switches . . . 73
16.3 Status indicators . . . 74
16.4 Output layout . . . 74
16.5 Exit Codes . . . 74

17 GPRinspect Reference **75**
17.1 Command Line . . . 75
17.2 Switches . . . 75
17.3 Output structure . . . 76
17.4 Exit Codes . . . 76

A Environment Variables **79**
A.1 Project search path . . . 79
A.2 Configuration . . . 79
A.3 Build engine . . . 79
A.4 Compiler discovery . . . 80

B Glossary **81**

C GNU Free Documentation License **85**

Index **87**

Version 27.0w
Date: Jun 10, 2026

CHAPTER

ONE

INTRODUCTION

This manual is the reference for the **GNAT Project** (GPR) system: the project file language and the tools that use it.

1.1 What is GPR?

A GPR project file (`.gpr`) describes a software component: where its sources live, how they should be compiled, what libraries or executables are produced, and how it relates to other components. The project file language is declarative - it specifies *what* to build, not *how* - and the GPR tools translate those declarations into concrete build, install, inspect, or clean operations.

GPR is multi-language: a single project tree may combine Ada, C, C++, and other languages. The active *configuration project* (generated by GPRconfig from the *Knowledge base (KB)*) supplies the toolchain-specific details - compiler drivers, default switches, library conventions - for each language.

1.2 Tools covered

This manual covers the following tools:

GPRbuild (***GPRbuild Reference***)
: Multi-language build tool. Compiles sources, binds Ada programs, builds libraries, and links executables using a DAG-based incremental build engine.

GPRclean (***GPRclean Reference***)
: Removes build results produced by GPRbuild.

GPRinstall (***GPRinstall Reference***)
: Installs build results into a prefix directory and maintains manifests for later uninstallation.

GPRls (***GPRls Reference***)
: Lists the sources, units, objects, and dependencies of a project tree, with up-to-date status.

GPRname (***GPRname Reference***)
: Scans source directories and generates a project file with a `Naming` package for non-standard file naming conventions.

GPRconfig (***GPRconfig Reference***)
: Probes the host for available compilers and generates a configuration project (`.cgpr`) describing the selected toolchains.

GPRinspect (***GPRinspect Reference***)
: Displays the resolved structure and attributes of a loaded project tree, in human-readable or JSON format.

1.3 Document structure

This manual is divided into two parts:

GPR Project Language
: Covers the project file language itself: project kinds and qualifiers, project extension, the project file syntax, source resolution rules, and the attribute reference.

GPR Tools
: Covers the knowledge base and each tool's command-line interface, project packages, and behavior.

Two appendices complete the manual: an *Environment Variables* reference and a *Glossary*.

1.4 LibGPR2

The GPR tools described in this manual are built on top of **LibGPR2**, an Ada library that implements the GPR project model - parsing project files, resolving sources and dependencies, and exposing the result as a queryable object model. It also provides a build infrastructure that can be used to implement custom incremental builders. Developers who need to integrate GPR project loading or build orchestration into their own tools work directly with LibGPR2.

1.5 Related documents

GPR User Guide
: Task-oriented companion to this manual. Where this manual specifies the project file language and tool options precisely, the User Guide explains how to apply them: setting up a project from scratch, organizing sources, handling scenarios, working with libraries, and more. Readers new to GPR should start there.

GPR2 Quick Start
: Introduction to LibGPR2 for developers who want to load and query project trees programmatically. Covers the main entry points with worked examples.

GPR2 Library Reference
: Comprehensive presentation of the main elements of the LibGPR2 API.

CHAPTER

TWO

PROJECT FILE LANGUAGE

GPR project files use an Ada-like syntax. This chapter describes the lexical rules, the structure of a project file, and the language constructs available within it.

2.1 Overview

The fundamental purpose of a project file is to supply **attribute values** to GPR tools. An attribute is a named configuration parameter - source directories, compiler switches, library name, and so on - declared with a `for` clause (see *Attribute Declarations*). Attributes may be *indexed* by a key such as a language name or source file name, and are grouped into **packages** that namespace them by tool (`Compiler'Switches`, `Linker'Switches`, ...).

Typed variables and `case` statements let attribute values vary by configuration without duplicating the project file.

2.2 Lexical Elements

2.2.1 Identifiers

Identifiers follow the same rules as Ada identifiers: they must start with a letter and may be followed by letters, digits, or underscores. Two consecutive underscores are not allowed. Identifiers are **case-insensitive** (`"Name"` and `"name"` are the same identifier).

```
simple_name    ::= identifier
name           ::= simple_name { . simple_name }
attribute_name ::= identifier
package_name   ::= identifier
project_name   ::= name
type_name      ::= identifier
variable_name  ::= identifier

project_ref   ::= project_name
package_ref   ::= package_name
                | project_name . package_name
variable_ref  ::= variable_name
                | package_name . variable_name
                | project_name . variable_name
                | project_name . package_name . variable_name
attribute_ref ::= 'project' ''' attribute_name [ '(' string_expression ')' ]
                | project_ref ''' attribute_name [ '(' string_expression ')' ]
                | package_ref ''' attribute_name [ '(' string_expression ')' ]
builtin_name  ::= 'external' | 'external_as_list' | 'file_as_list'
                | 'lower' | 'upper' | 'split' | 'match' | 'filter_out'
```

(continues on next page)

(continued from previous page)

```
              | 'item_at' | 'remove_prefix' | 'remove_suffix'
              | 'default' | 'alternative'
```

2.2.2 Strings

String literals are written between double quotes, as in Ada. They are in general **case-sensitive**, except where noted (for example, file names are case-insensitive on systems with case-insensitive file systems).

2.2.3 Reserved Words

The following Ada 95 reserved words are used in project files:

```
abstract  all     at       case
end       for     is       limited
null      others  package  renames
type      use     when     with
```

The following identifiers are additional reserved words specific to project files:

```
extends  external  external_as_list  file_as_list  project
alternative  default  filter_out  item_at  lower  match
remove_prefix  remove_suffix  split  upper
```

Note that `aggregate` and `library` are qualifiers that may precede the keyword `project` but are not reserved words.

2.2.4 Comments

Comments follow Ada syntax: two consecutive hyphens (`--`) start a comment that extends to the end of the line.

2.3 Project File Structure

A project file consists of an optional context clause followed by a project declaration.

```
project        ::= context_clause project_declaration

context_clause ::= { with_clause }
with_clause    ::= [ 'limited' ] 'with' path_name { , path_name } ;
path_name      ::= string_literal

project_declaration ::= simple_project_declaration
                      | project_extension

simple_project_declaration ::=
  [ qualifier ] 'project' project_name 'is'
    { declarative_item }
  'end' project_name ;

project_extension ::=
  [ qualifier ] 'project' project_name 'extends' [ 'all' ] path_name 'is'
    { declarative_item }
  'end' project_name ;
```

(continues on next page)

(continued from previous page)

```
qualifier ::= 'abstract'
            | 'library'
            | 'aggregate'
            | 'aggregate' 'library'
            | 'configuration'
            | 'standard'
```

For the semantics of each qualifier see *Project Kinds*. For the semantics of `extends` and `extends all` see *Project Extension*.

2.3.1 Context Clauses

```
context_clause ::= { with_clause }
with_clause    ::= [ 'limited' ] 'with' path_name { , path_name } ;
path_name      ::= string_literal
```

A project may import other projects via `with` clauses. Imported projects make their sources and attributes visible to the importing project.

A `limited with` allows two projects to have mutual visibility of each other's compiled objects without creating a cycle in the attribute-dependency graph. A project imported via `limited with` contributes its compiled artifacts (object files, libraries) but its attributes cannot be referenced by the importing project. This keeps attribute resolution strictly acyclic while still permitting the mutual-view pattern needed by some system architectures.

Path names in `with` clauses are strings containing a path to a `.gpr` file. The `.gpr` extension is optional and will be added automatically if absent. Paths may be absolute or relative. Relative paths are resolved with respect to the directory of the current project file, or against directories listed in *GPR_PROJECT_PATH* and *ADA_PROJECT_PATH*. The directory separator may always be `/` even on Windows.

A given project name may appear only once across all `with` clauses of the same project. Cycles without `limited with` are forbidden.

2.3.2 Child projects

A project whose name is a dotted name (`Parent.Child`) is a *child project* of `Parent`. This is purely a naming convention expressing a close relationship between the two; a child project does **not** implicitly import its parent. An explicit `with` or `extends` clause is required.

The child project file name uses a dash in place of the dot: `Math_Proj.Tests` lives in `math_proj-tests.gpr`.

```
-- math_proj-tests.gpr
with "math_proj.gpr";
project Math_Proj.Tests is
   --  Parent variables and attributes accessible via the parent name prefix:
   Obj_Dir := Math_Proj.Object_Dir;
   ...
end Math_Proj.Tests;
```

Once the parent is imported or extended, its variables and attributes are accessible using the parent project name as a prefix (e.g. `Math_Proj.Object_Dir`).

2.3.3 Source ownership

Each project directly owns a set of **immediate sources**: files identified through its source-related attributes (source directories, explicit file lists, etc.). The full set of **sources** visible to a project extends this with the immediate sources of every project it depends on, directly or indirectly (unless overridden by extension). For the rules governing basename uniqueness, search order, and source shadowing in extensions, see *Source Resolution*.

2.4 Declarations

Declarations introduce types, variables, attributes, and packages. They are processed sequentially in the order they appear; a name becomes visible immediately after its declaration.

```
declarative_item ::= simple_declarative_item
                   | typed_string_declaration
                   | package_declaration

simple_declarative_item ::= variable_declaration
                          | typed_variable_declaration
                          | attribute_declaration
                          | case_construction
                          | empty_declaration

empty_declaration ::= 'null' ;
```

An empty declaration (`null;`) is valid anywhere a declaration is allowed and has no effect.

Scope rules:

Declarations are scoped to either the project level or the enclosing package. The following rules apply:

- **Typed string declarations** may only appear at the project level. Once declared, the type is visible throughout the entire project file, including inside packages, and may be referenced from other projects using a qualified name (`Project_Name.Type_Name`).
- **Package declarations** may only appear at the project level; packages cannot be nested.
- **Variable declarations** (typed or untyped) may appear at the project level or inside a package. A project-level variable is visible throughout the project file. A variable declared inside a package is local to that package and can be referenced from outside only via a qualified name (`Package_Name.Variable_Name` or `Project_Name.Package_Name.Variable_Name`).
- **Attribute declarations** may appear at the project level (setting project-level attributes) or inside a package (setting package attributes). Attributes from other projects or packages are accessible via qualified names.
- **Case constructions** may appear at both levels. The discriminant must be a typed variable already declared before the `case` construct. Inside a case arm, only attribute declarations, variable declarations (for variables already declared before the `case`), nested case constructions, and `null` are allowed - type and package declarations are forbidden.

2.5 Values

GPR projects manipulate two kinds of values:

String
: A single string, e.g. `"debug"` or `"src/main.adb"`.

List of strings
: An ordered sequence of strings, e.g. `("-O2", "-g")`. The empty list is written `()`.

Attributes and variables each hold one of these two kinds. The kind of an attribute is fixed by the language specification (see *Attributes*); the kind of an untyped variable is inferred from its first declaration.

Values may incorporate previously declared variables and attributes via references, call built-in functions, and may be combined using the `&` concatenation operator. Once any operand is a list the result is a list, and the list must be the left operand. See *Built-in Functions* and *Expressions* for the formal grammar.

2.6 Built-in Functions

Built-in functions may be used inside expressions. Their names are not reserved words and may be used as variable names elsewhere; a name is interpreted as a built-in call only when immediately followed by `(`.

```
builtin_call ::= builtin_name '(' [ expression { , expression } ] ')'
```

A built-in call may return either a string or a list of strings depending on the function; see *Built-in Functions* for the specific signatures of each.

2.6.1 The `External` function

`External` retrieves a string value from the build environment. The first argument is the name of an external variable; the optional second argument is the default value. Its value is resolved, in priority order, from:

1. The `-X\ *name*\ =\ *value*` command-line switch.
2. An environment variable of the same name.
3. The second argument, if supplied (the default value).

If none of these sources provides a value, an error is reported.

`External` is typically used to initialize **typed variables** (see *Typed String Declaration*), which are then referenced in `case` constructions to vary attribute values across build scenarios. Such variables are commonly called *scenario variables*.

```
type Build_Mode_Type is ("debug", "release");
Build_Mode : Build_Mode_Type := external ("BUILD_MODE", "debug");
```

2.6.2 The `External_As_List` function

The `External_As_List` function retrieves a list of strings from the environment by splitting an external variable on a separator.

```
external_as_list_value ::=
  'external_as_list' ( string_literal , string_literal )
```

The first argument is the external variable name; the second is the separator string. The value is looked up on the command line first, then as an environment variable. If undefined, an empty list is returned (no error). Leading and trailing separators are discarded.

Key differences from `External`:

- No default-value parameter; returns `()` when the variable is undefined.
- Returns a list, not a string.

Example:

```
--  If SWITCHES is "-O2,-g", External_As_List ("SWITCHES", ",")
--  returns ("-O2", "-g").
```

2.6.3 The `File_As_List` function

> **Note**
>
> This function is not available in tools based on the legacy GPR1 framework.

`File_As_List` reads a text file and returns its lines as a list of strings.

```
list ::= 'file_as_list' ( string_literal )
```

The argument is the path to the file. Returns `()` if the file does not exist or is empty.

```
Source_Files := file_as_list ("sources.txt");
```

2.6.4 String Manipulation Functions

> **Note**
>
> The functions in this section are not available in tools based on the legacy GPR1 framework. This includes all tools that have not yet migrated to the GPR2 library.

`Split`

Splits a string on a separator and returns the parts as a list. Empty parts are not included. Returns `()` if the string is empty or consists entirely of separators.

```
Split ( string_literal , string_literal )
```

```
--  Split ("-gnatf,-gnatv", ",")  =>  ("-gnatf", "-gnatv")
```

`Lower` / `Upper`

Return the argument with all characters converted to lower or upper case. Accept a string or a list.

```
--  Lower ("FOO")          =>  "foo"
--  Upper (("one","two"))  =>  ("ONE", "TWO")
```

`Remove_Prefix` / `Remove_Suffix`

Remove a fixed prefix or suffix from a string or from each element of a list, if present.

```
--  Remove_Prefix (("libone", "two", "libthree"), "lib")
--    =>  ("one", "two", "three")

--  Remove_Suffix ("libZ.so", ".so")  =>  "libZ"
```

`Filter_Out`

Removes from a list all elements matching a regular-expression pattern.

```
--  Filter_Out (("value1", "or", "another", "one"), ".*o.*")
--    =>  ("value1")
```

Match

Returns elements of a string or list that match a regular-expression pattern. An optional third argument provides a replacement pattern applied to each match.

```
--  Match ("x86_64-linux-gnu", "linux")  =>  "linux"

--  Match (("value1","or","another","one"), "(.*r)", "r:\1")
--    =>  ("r:or", "r:another")
```

Item_At

Returns one element from a list by index. Negative indices count from the end (`"-1"` is the last element).

```
--  Item_At (("one","two","three","last"), "2")   =>  "two"
--  Item_At (("one","two","three","last"), "-1")  =>  "last"
```

Default / Alternative

`Default` returns its second argument when the first is the empty string; otherwise it returns the first argument.

`Alternative` returns its second argument when the first is *not* the empty string; otherwise it returns the first argument.

```
--  Default     ("",  "fallback")  =>  "fallback"
--  Default     ("x", "fallback")  =>  "x"
--  Alternative ("",  "fallback")  =>  ""
--  Alternative ("x", "fallback")  =>  "fallback"
```

2.7 Expressions

An **expression** builds a value from literals, variable references, attribute references, built-in function calls, and concatenation:

```
string_literal ::= "{string_element}"    --  Same as Ada

string_expression ::= string_literal
    | variable_ref
    | attribute_ref
    | builtin_call
    | string_expression & string_expression

string_list ::= ( string_expression { , string_expression } )
    | variable_ref
    | attribute_ref
    | builtin_call

term       ::= string_expression | string_list
expression ::= term { & term }
```

Concatenation rules follow Ada conventions. Once any operand is a list, the result is a list:

```
function "&" (X : String;      Y : String)      return String;
function "&" (X : String_List; Y : String)      return String_List;
function "&" (X : String_List; Y : String_List) return String_List;
```

Example:

```
List  := () & File_Name;                  --  one element
List2 := List & (File_Name & ".orig");    --  two elements
Big   := List & List2;                    --  three elements
--  Illegal := "gnat.adc" & List2;        --  error: list must be left operand
```

2.8 Typed String Declaration

A **type declaration** introduces a finite set of string literals. Variables declared with this type are restricted to the listed values. Type declarations may only appear at the project level, not inside a package.

```
typed_string_declaration ::=
  'type' type_name 'is'
    ( string_literal { , string_literal } ) ;
```

String literals in the list are case-sensitive and must be distinct. Example:

```
type OS is ("GNU/Linux", "Unix", "Windows", "VMS");
```

Variables of a string type are called **typed variables**; all others are **untyped variables**. A type declared in another project may be referenced using a qualified name (`Project_Name.Type_Name`).

2.9 Variables

Variables store a string or list-of-strings value and may appear in expressions. A variable declaration creates the variable and assigns it a value. Before its first declaration, a variable defaults to "" (empty string).

2.9.1 Typed Variables

A typed variable must be declared exactly once. Its type restricts the values it may hold, and because it can only be set once, all `case` constructions in the file see a consistent value - making it behave effectively as a constant.

```
typed_variable_declaration ::=
  variable_name : type_name := string_expression ;
```

```
type OS_Type is ("GNU/Linux", "Unix", "Windows");
OS : OS_Type := external ("OS", "GNU/Linux");
```

2.9.2 Untyped Variables

An untyped variable may be declared and reassigned any number of times. Its kind (string or list) is fixed by the first declaration; subsequent declarations must match.

```
variable_declaration ::= variable_name := expression ;
```

```
Name      := "readme.txt";
Save_Name := Name & ".saved";
Flags     := ("-O2", "-g");
```

2.9.3 Variable References

A variable may be referenced by its simple name or a qualified name (`variable_ref` as defined in *Identifiers*):

```
Mode                      --  variable in current scope
Compiler.Opt_Level        --  variable in a package
Common.Build_Mode         --  variable in an imported project
Common.Compiler.Opt_Level --  variable in a package of an imported project
```

A simple name resolves to the current package (if any) or the current project. Qualified names may refer to a package in the current project, an imported project, a base project (direct or indirect), or a package within any of those.

2.10 Attribute Declarations

Attributes are the primary mechanism for communicating information to build tools. An attribute declaration uses the `for ... use` syntax:

```
attribute_declaration ::=
    'for' attribute_name 'use' expression ;
  | 'for' attribute_name '(' string_expression ')' 'use' expression ;
```

The optional parenthesised string expression is the **index** of the attribute. Indexed attributes associate different values with different keys - for example, per-language compiler switches:

```
package Compiler is
   for Switches ("Ada") use ("-O2", "-gnat2022");
   for Switches ("C")   use ("-O2", "-Wall");
end Compiler;
```

An attribute may also be declared without an index, in which case it has a single value for the whole project or package:

```
for Library_Name use "mylib";
```

Different attributes accept different kinds of indexes:

Language (case-insensitive)
: A language identifier such as `"Ada"` or `"C"`. Used by most `Compiler`, `Binder`, `Linker`, and `Naming` attributes.

  ```
  for Compiler'Switches ("Ada") use ...;
  ```

File / Glob (case-sensitivity host-dependent)
: A source file simple name or a glob pattern (`*`, `?`, `[]`). When a glob pattern is used, the attribute applies to every source file whose name matches that pattern. Some attributes also accept `others` as an index, which matches any source file not matched by a more specific index in the same project. Whether `others` is accepted is specified per attribute in *Attributes*.

  ```
  for Compiler'Switches ("main.adb") use ...;
  ```

  ```
  for Compiler'Switches ("*.c") use ...;
  ```

File / Glob / Language (case-sensitivity host-dependent for files, case-insensitive for languages)
: Accepts a source file simple name, a glob pattern, or a language identifier. Strings containing dots or glob characters (`*`, `?`, `[`, `]`) are treated as file names or glob patterns; all other strings are treated as language identifiers. Some attributes also accept `others` as a catch-all index. Whether `others` is accepted is specified per attribute in *Attributes*.

  When multiple declarations for the same attribute exist in a project with different indexes, the value that applies to a given source file is resolved in the following priority order:

1. An index that is an exact match on the file name.
2. An index that is a glob pattern matching the file name.
3. An index matching the file's language.
4. The `others` index, if present.

Used by `Compiler'Switches` and `Roots`.

Unit (case-insensitive)
An Ada unit name. Used by `Naming'Spec` and `Naming'Body`.

```
for Naming'Spec ("My.Package") use "my-package.ads";
```

String
An arbitrary string key. Used for instance by the aggregate-project attribute `External`, where the index is the name of an external variable.

```
for External ("BUILD_MODE") use "release";
```

Attribute references (`attribute_ref` as defined in *Identifiers*) allow reading values from other projects or packages:

```
for Switches ("Ada") use Common.Compiler'Switches ("Ada") & ("-g");
```

For the full list of attributes and their semantics, see *Attributes*. Individual tools may define additional attributes in their own packages; refer to each tool's documentation.

2.11 Packages

A project file may contain **packages**, which group related attributes - typically all attributes used by one tool. A given package name may appear at most once per project file.

```
package_declaration ::= package_spec | package_renaming | package_extension

package_spec ::=
  'package' package_name 'is'
     { simple_declarative_item }
  'end' package_name ;
```

The standard packages recognized in all project files are:

`Binder`
Options for the binder (`gnatbind` / GPRbuild).

`Builder`
Global build options (executable names, global compilation switches).

`Clean`
Options for `gprclean`.

`Compiler`
Compilation options per language.

`Install`
Options for `gprinstall`.

`Linker`
Link options.

`Naming`
Source-file naming conventions.

Other tool-specific packages may be defined by individual tools; refer to each tool's documentation.

Note

Each tool only reads the packages it recognizes; unknown package *declarations* are silently ignored. However, referencing an attribute or variable from an unknown package in an expression - for example, reading `Clean'Switches` inside a project loaded by GPRbuild - will cause a parsing error and the project will be rejected. Avoid cross-package attribute references unless you can guarantee that every tool loading the project knows the referenced package.

A minimal (empty) package:

```
project Simple is
   package Builder is
   end Builder;
end Simple;
```

A package may contain attribute declarations, variable declarations, and case constructions.

Note

When a name could refer to either a project or a package, it always designates the project. Avoid naming projects after standard package names or names starting with `gnat`.

2.11.1 Package Renaming

A package may be defined by a **renaming declaration**, which gives the new package the same attributes as a package declared in another project. The renamed project must appear in the current project's context clause (or be its base project). No attributes may be added or overridden in a renaming; use a package extension for that.

```
package_renaming ::=
  'package' package_name 'renames'
    project_name . package_name ;
```

Package renaming is a common way to share settings: define the authoritative package in one project file and rename it in all projects that need the same settings.

2.11.2 Package Extension

A **package extension** works like a renaming but also allows adding or overriding attributes. It is particularly useful in project extension: a package inherited from the base project can be explicitly extended to add or override specific attributes without replacing it entirely.

```
package_extension ::=
  'package' package_name 'extends'
    project_name . package_name 'is'
     { simple_declarative_item }
  'end' package_name ;
```

2.12 Case Constructions

A `case` construction selects attribute and variable declarations based on the value of a typed variable, enabling conditional project configuration.

```
case_construction ::=
  'case' variable_ref 'is' { case_item } 'end' 'case' ;

case_item ::=
  'when' discrete_choice_list '=>'
    { case_construction
    | attribute_declaration
    | variable_declaration
    | empty_declaration }

discrete_choice_list ::= string_literal { '|' string_literal } | 'others'
```

Rules:

- All choices must be distinct.
- All values of the type must be covered, either explicitly or via `others`.
- `others` must be the last alternative.
- The case expression must be a variable (typed or untyped) whose value is known at load time.
- Inside a `case`, only case constructions, attribute declarations, variable declarations (for already-declared variables), and `null` declarations are allowed. Type and package declarations are not.

Example:

```
project MyProj is
   type OS_Type is ("GNU/Linux", "Unix", "Windows", "VMS");
   OS : OS_Type := external ("OS", "GNU/Linux");

   package Compiler is
      case OS is
         when "GNU/Linux" | "Unix" =>
            for Switches ("Ada") use ("-gnath");
         when "Windows" =>
            for Switches ("Ada") use ("-gnatP");
         when others =>
            null;
      end case;
   end Compiler;
end MyProj;
```

CHAPTER

THREE

PROJECT TREE

A GPR project file rarely exists in isolation. A system is typically split across several project files, each responsible for one component. This chapter describes how multiple project files are connected into a project tree.

3.1 Project tree structure

A GPR project file rarely exists in isolation. A system is typically split across several project files, each responsible for one component. The connected graph of project files forms the **project tree**.

The project tree is a directed acyclic graph (DAG) rooted at the **root project** - the project passed directly to the build tool (for example `gprbuild -P root.gpr`). Two kinds of edges connect projects in the tree:

Import (`with` clause)
: The importing project depends on the imported project for compilation and for attribute visibility.

Extension (`extends` clause)
: The extending project inherits sources and attributes from a base project. See *Project Extension*.

Aggregation (`Project_Files`) is a third relationship specific to aggregate and aggregate library projects; see *Aggregate Project*.

Every project in the tree is loaded once and shared, regardless of how many other projects import it.

3.2 The `with` clause

A project declares its imports at the top of the file, before the project declaration, using `with` clauses:

```
with "libs/utils/utils.gpr";
with "libs/crypto/crypto.gpr";

project My_App is
   for Source_Dirs use ("src");
   for Main        use ("main.adb");
end My_App;
```

Multiple paths may appear in a single clause, separated by commas:

```
with "utils.gpr", "crypto.gpr";
```

A given project may appear at most once across all `with` clauses of the same importing project. Duplicates are an error.

3.2.1 What a plain `with` makes visible

From within the body of the importing project, a plain `with` exposes all of the following from the imported project:

Types and variables
Typed string type definitions and variable declarations are accessible via the `Imported_Project.Name` notation.

Attributes and packages
Attribute values may be read using `Imported_Project'Attribute` (or `Imported_Project.Package'Attribute`). Packages may be renamed into the importing project with a `package P renames Imported_Project.P` declaration.

Source files
Sources of the imported project are compiled as needed; the resulting object files and Ada Library Information files (`*.ali`) are made available to the importing project's sources.

Example - sharing settings from an abstract project:

```
with "common.gpr";

project My_App is
   --  Read a typed variable from Common
   Mode : Common.Build_Mode_Type := Common.Build_Mode;

   --  Alias the whole Compiler package from Common
   package Compiler renames Common.Compiler;

   --  Extend the linker switches defined in Common
   package Linker is
      for Switches ("Ada") use
        Common.Linker'Switches ("Ada") & ("-lm");
   end Linker;
end My_App;
```

3.2.2 Path resolution

The path in a `with` clause is a string naming a `.gpr` file. The `.gpr` extension is optional and appended automatically if absent. The directory separator is always `/`, even on Windows.

Paths are resolved in the following order:

1. If the path is absolute, it is used directly.
2. Otherwise it is resolved relative to the directory of the importing project file.
3. If no file is found there, each directory listed in *GPR_PROJECT_PATH* is searched in order.
4. Then the directories in *ADA_PROJECT_PATH* are searched.
5. Finally the default project paths provided by the selected toolchain(s) are searched.

The first match terminates the search. Installed libraries typically rely on steps 3-5, so they can be imported by bare name without a path:

```
with "gnatcoll.gpr";   -- resolved via GPR_PROJECT_PATH or toolchain paths
```

3.3 The `limited with` clause

Because attribute evaluation must be acyclic, a cycle formed entirely from plain `with` edges is an error. The `limited with` variant relaxes this constraint by restricting what is visible across the import:

```
limited with "component_b.gpr";

project Component_A is
   for Source_Dirs use ("src_a");
end Component_A;
```

With a `limited with`:

Visible
: Source files of the imported project are compiled as part of the tree; their object files and ALI files are available to the importing project's sources.

Not visible
: The imported project's types, variables, attributes, and packages cannot be referenced inside the importing project's body.

The `limited` modifier applies to the *attribute-visibility* graph only; it has no effect on the *source-dependency* graph. Two components whose Ada units are mutually recursive can therefore coexist in the same tree:

```
--  component_a.gpr
limited with "component_b.gpr";
project Component_A is
   for Source_Dirs use ("src_a");
end Component_A;

--  component_b.gpr
with "component_a.gpr";
project Component_B is
   for Source_Dirs use ("src_b");
end Component_B;
```

Here `Component_B` may reference attributes of `Component_A`, but `Component_A` may not reference attributes of `Component_B`.

A cycle requires at least one `limited with` edge to break the attribute cycle; without one the project loader reports an error.

> **Note**
>
> A need for `limited with` sometimes signals that the two components share a common abstraction that should be factored into a third project on which both depend. Consider restructuring before reaching for `limited with`.

CHAPTER

FOUR

PROJECT KINDS

Every GPR project file has a **kind** that determines what the project produces, which attributes and packages are valid within it, and how other projects may reference it.

The kind is expressed via a **qualifier** - one or two identifiers placed immediately before the `project` keyword. The qualifier may be omitted, in which case the kind is inferred from the attributes declared inside the project:

- A project that declares both `Library_Name` and `Library_Dir` is treated as a library project.
- A project that declares `for Source_Dirs use ();`, `for Source_Files use ();`, or `for Languages use ();` is treated as an abstract project.
- Otherwise the project is treated as standard.

For non-standard project kinds, **using an explicit qualifier is best practice**: it documents intent clearly, GPR tools validate that the declared attributes are consistent with the stated kind, and the kind is immediately visible to readers without having to inspect the attribute declarations. Omitting the qualifier for a plain standard project is common and acceptable.

4.1 Formal syntax

```
simple_project_declaration ::=
  [ qualifier ] 'project' <project_>name 'is'
    { declarative_item }
  'end' <project_>name ;

project_extension ::=
  [ qualifier ] 'project' <project_>name 'extends' [ 'all' ] path_name 'is'
    { declarative_item }
  'end' <project_>name ;

qualifier ::= 'abstract'
            | 'library'
            | 'aggregate'
            | 'aggregate' 'library'
            | 'configuration'
            | 'standard'
```

The qualifier is optional. When present, it must match the kind inferred from the project's attributes. Omitting it for a plain standard project is common; for all other kinds, stating the qualifier explicitly is recommended.

4.2 Standard Project

Qualifier: `standard` (or omitted)

A standard project contains source files and produces executables or object files. It is the default project kind. The qualifier may be stated explicitly:

```
standard project My_App is
   for Source_Dirs use ("src");
   for Main use ("main.adb");
end My_App;
```

The `standard` qualifier is optional and most often omitted:

```
project My_App is
   for Source_Dirs use ("src");
   for Main use ("main.adb");
end My_App;
```

Key properties:

- May declare any attribute or package.
- May import other projects via `with` clauses.
- May extend another project, either directly (`extends`) or across an entire project hierarchy (`extends all`). See *Project Extension*.
- May be extended by other projects.

> **Note**
>
> The `Main` attribute is honored only when the project is the **root project** of a build invocation, or is directly listed in the `Project_Files` attribute of an aggregate project. When a standard project is imported via a `with` clause by another project, its `Main` attribute is ignored and no executable is produced.

4.3 Abstract Project

Qualifier: `abstract`

An abstract project has no source files. It is used to share attributes, packages, and typed variable definitions across multiple concrete projects.

```
abstract project Common_Switches is
   package Compiler is
      for Switches ("Ada") use ("-O2", "-gnat2022");
   end Compiler;
end Common_Switches;
```

Constraints:

- Either the `abstract` qualifier is present, or - for backward compatibility - at least one of `Source_Dirs`, `Source_Files`, or `Languages` must be declared empty. Prefer the explicit qualifier in new project files.
- If it extends another project, the base project must also be abstract.
- May be imported by any other project kind, including aggregate and aggregate library projects.

4.4 Library Project

Qualifier: `library`

A library project builds a library - a static archive or a dynamic/shared image - rather than executables. It is the primary mechanism for producing reusable compiled components.

```
library project My_Lib is
   for Source_Dirs   use ("src");
   for Languages     use ("Ada");
   for Library_Name  use "mylib";
   for Library_Dir   use "lib";
   for Library_Kind  use "static";
end My_Lib;
```

> **Note**
>
> For backward compatibility, a project without the `library` qualifier is implicitly treated as a library project when it declares both `Library_Name` and `Library_Dir`. Prefer the explicit qualifier in new project files.

Required attributes:

`Library_Name`
: Name of the library to build (e.g. `"mylib"` produces `libmylib.a` or `libmylib.so`).

`Library_Dir`
: Directory where the built library is placed. Must differ from the object directory (`Object_Dir`).

Library kinds

Controlled by the `Library_Kind` attribute:

`"static"`
: Archive of object files (default). Linked directly into the final executable.

`"static-pic"`
: Archive built with position-independent code, suitable for later inclusion in a shared library.

`"dynamic"` / `"relocatable"`
: Shared library loaded at program start. Changes to the library implementation require no relink of the final executable.

By default, dynamic library projects may only import other dynamic library projects; importing non-library or static library projects requires setting `Shared_Library_Prefix` and related configuration attributes.

Shared library versioning

On Linux, the `Library_Version` attribute controls shared library versioning:

```
library project My_Lib is
   for Library_Name    use "mylib";
   for Library_Dir     use "lib";
   for Library_Kind    use "dynamic";
   for Library_Version use "libmylib.so.1.2.3";
end My_Lib;
```

The value is the full versioned file name of the library. The build tool produces a symlink for every prefix of the version string:

```
lib/
  libmylib.so.1.2.3   -- the actual shared object (SONAME: libmylib.so.1)
  libmylib.so.1.2      -- symlink -> libmylib.so.1.2.3
  libmylib.so.1        -- symlink -> libmylib.so.1.2
  libmylib.so          -- symlink -> libmylib.so.1
```

The SONAME embedded in the library is the major-version form of `Library_Version` (e.g. `libmylib.so.1` from `libmylib.so.1.2.3`).

`Library_Version` has no effect on `static` or `static-pic` libraries.

4.4.1 Stand-alone Library Projects

A **stand-alone library** (SAL) bundles its own elaboration code so that the final program does not need to drive elaboration of each unit individually. Depending on the platform and configuration, elaboration may be triggered automatically at load time or may require an explicit call from the main subprogram (see *Auto-initialization* below). A library becomes stand-alone when at least one of the following attributes is set:

- `Library_Interface` - list of Ada unit names that form the public interface of the library.
- `Interfaces` - list of source file names (language-agnostic alternative to `Library_Interface`).

Units listed in the interface are visible to importers; other units in the library are implementation details and may not be depended upon directly.

The `Library_Standalone` attribute refines the elaboration model:

`"standard"`
: Default SAL: the library provides an elaboration entry point that the program's elaboration must call.

`"encapsulated"`
: Encapsulated SAL: the library is fully self-contained; all its dependencies are either encapsulated SALs themselves or are part of the language runtime. No external elaboration call is required.

`"no"`
: Not stand-alone (default when no interface is declared).

Auto-initialization

Whether elaboration is called automatically depends on the configuration. When `Library_Auto_Init_Supported` is `"true"` in the active configuration project, automatic initialization is available. The library project may then set `Library_Auto_Init` to `"true"` (to opt in) or `"false"` (to opt out); when not set, it defaults to `Library_Auto_Init_Supported`.

Setting `Library_Auto_Init` to `"true"` when `Library_Auto_Init_Supported` is `"false"` triggers a warning at load time.

4.5 Aggregate Project

Qualifier: `aggregate`

An aggregate project groups a coherent set of related project files so that they can be built with a single GPRbuild invocation. It does not contain sources itself; all sources come from the constituent projects listed in `Project_Files`. It may also set default values for external variables seen by all constituent projects via the `External` attribute.

Each directly aggregated project becomes the root of its own independent *namespace*. Ada requires compilation-unit names to be unique within a namespace, so duplicate unit names across different namespaces are permitted and do not cause conflicts. Components shared by multiple namespaces are deduplicated and built only once.

```
aggregate project All is
   for Project_Files use ("subsystem_a/a.gpr",
                          "subsystem_b/b.gpr",
                          "subsystem_c/c.gpr");
end All;
```

Required attribute: `Project_Files` - a list of paths to constituent `.gpr` files. Paths are relative to the aggregate project file's directory. Glob patterns (`*` and `**`) are supported. Constituent projects may themselves be aggregate projects.

Permitted packages: `Builder` only (`Switches` and `Global_Compilation_Switches` sub-attributes).

Permitted project-level attributes:

`Project_Files`
: Constituent project files.

`Project_Path`
: Extra directories searched when resolving `with` clauses inside constituent projects.

`External`
: Replaces the default value of an external variable as seen by all constituent projects. An explicit `-X` switch on the command line takes precedence. Only meaningful when this project is the root.

`Object_Dir`
: Directory where GPR-aware tools may store per-project data, such as the file index used to speed up reloading of a previously built project. Only meaningful when this project is the root.

Import and use restrictions:

- May only `with` abstract or other aggregate projects.
- Cannot be imported by non-aggregate projects; it is valid only as a root project or as a constituent (directly or transitively) of another aggregate project.
- Cannot be extended.
- When used as a constituent of another aggregate, only its `Project_Files` list is taken into account: `External` and `Object_Dir` are ignored, and the aggregated sub-projects are folded directly into the root aggregate's namespace structure.

4.6 Aggregate Library Project

Qualifier: `aggregate library`

An aggregate library project builds a single library by collecting the object files produced by a set of constituent projects. It combines the properties of a library project and an aggregate project: it has `Library_Name` and `Library_Dir`, and it lists its constituents via `Project_Files`.

Warning

Despite sharing the `aggregate` qualifier and the `Project_Files` attribute with aggregate projects, aggregate library projects are a fundamentally different construct. An aggregate library project is an ordinary library project that collects object files from its constituents into a single library: it can be `with`-ed by any other project and may appear anywhere in the dependency graph. It does not support the `External` attribute, and constituent subtrees are not isolated - Ada unit names must be unique across all constituents.

```
aggregate library project Full_Lib is
   for Project_Files use ("module_a/a.gpr",
                          "module_b/b.gpr");
   for Library_Name use "full";
   for Library_Dir  use "lib";
end Full_Lib;
```

The resulting library may be of any kind (`static`, `static-pic`, `dynamic`), independently of the library kinds of the constituent projects.

An aggregate library project is designed to incorporate its constituent projects without forcing them to be rebuilt. To enforce this, the `Compiler` package cannot be declared in an aggregate library project: constituent objects are used in the form already produced by each constituent project.

Declaring `Object_Dir` is strongly recommended. It serves two purposes: GPR-aware tools may store per-project metadata there, and when the aggregate library's kind differs from its constituents' - for example, building a dynamic library from static constituent projects - the toolchain recompiles affected sources with position-independent code (`-fPIC`) and stores the results in `Object_Dir`, keeping constituent directories untouched. If not declared, the project directory is used as default.

Required attributes: `Library_Name`, `Library_Dir`, `Project_Files`.

Stand-alone aggregate library projects: `Library_Interface` or `Interfaces` may be declared to make the aggregate library stand-alone, including the encapsulated variant (`Library_Standalone => "encapsulated"`).

Library compatibility: As with any library project, the standard compatibility rules apply: a dynamic aggregate library cannot import non-library or static library projects. It may also extend an abstract project.

4.7 Configuration Project

Qualifier: `configuration`

A configuration project describes the tools available for a given target and runtime - compilers, linkers, and so on - as well as the switches required to invoke each of them. It is used internally by all GPR tools (GPRbuild, GPRclean, GPRinstall, GPRls, etc.) and by any tool built on top of the GPR2 library. Any such tool can generate a configuration project automatically; GPRconfig is also available as a standalone tool to produce one independently. The file extension is `.cgpr`.

Configuration projects are part of the project tree in a special role: they supply the toolchain attributes that all other projects in the tree rely on. They can be loaded separately from the user project tree - via `--config` or `--autoconf` - and are never referenced by `with` clauses in regular project files.

For the format and contents of configuration projects, see *GPRconfig Reference*.

CHAPTER

FIVE

PROJECT EXTENSION

Most project kinds may extend another project using the `extends` clause. The extending project inherits sources and settings from the base and may selectively override them. An extending project can also import other projects alongside its base.

Aggregate projects and aggregate library projects may extend abstract projects to inherit their attributes and package settings.

```
project Patched extends "original/original.gpr" is
   --  Source files here shadow those in original.gpr.
   --  All other sources and settings are inherited.
end Patched;
```

5.1 Simple Extension

5.1.1 What is inherited

- **Source files** - All source files from the base are implicitly present. A source file in the extending project whose basename matches one from the base shadows the base version. See *Source Resolution* for the full shadowing rules.
- **Attribute values** - Project-level attributes not declared in the extending project are inherited from the base.
- **Packages** - Any package not declared in the extending project is inherited in full from the base. A package that *is* declared in the extending project replaces the entire base package; there is no per-attribute inheritance within a package. Use package extension (`package Compiler extends Base.Compiler is ...`) to inherit a base package and add or override individual attributes.

Exception: `Linker'Linker_Options` is never inherited.

5.1.2 Excluding inherited sources

To remove an inherited source without providing a replacement, list its file name in `Excluded_Source_Files`:

```
project Work extends "../base/base.gpr" is
   for Source_Files use ("pack.ads");
   --  New spec does not require a body
   for Excluded_Source_Files use ("pack.adb");
end Work;
```

`Excluded_Source_List_File` accepts the same information as a path to a text file with one file name per line.

5.2 Extension Hierarchies

Extension may be chained: an extending project may itself be extended further. The `extends` relationship forms a linear chain (each project has at most one direct base). A project that extends another may also import additional projects normally.

5.2.1 Import redirection

When loading a project tree, the build system automatically redirects imports to their extending versions. If:

- project *A* extends *B* and imports *C*, and
- *C* extends *D*, and
- *B* imports *D*,

then in *A*'s context, *B*'s import of *D* is transparently replaced by an import of *C*. Any reference to *D* anywhere in the base hierarchy is similarly redirected, ensuring that the entire tree consistently uses *C* instead of *D*.

This redirection applies transitively: if the base hierarchy has further imports of *D* through intermediate projects, those are all replaced by *C* as well.

5.3 `extends all`

When working with a large project hierarchy, replacing sources in a single constituent project normally requires extending every project along the import path up to the root (so that import redirection keeps the tree consistent). `extends all` eliminates this overhead: it implicitly extends every project in the import closure of the named base, with import redirection applied throughout, so the whole subtree is immediately available for targeted amendment.

```
project Full_Override extends all "original/original.gpr" is
end Full_Override;
```

To replace a specific constituent - for example to substitute instrumented sources for a particular library - import an explicit extending project for that constituent from within the `extends all` project. The explicit extension replaces the corresponding implicit one, and import redirection ensures all other projects in the closure consistently reference the new version instead of the original.

5.4 Restriction

A project may not import, directly or indirectly, both an extending project *P* and any project that *P* extends (directly or indirectly). Without this restriction, two import paths could reach different versions of the same source file.

The standard solution is to introduce an empty intermediate extension so that all import paths go through the extending project:

```
--  b_ext.gpr - bridges the gap so no path reaches the original b.gpr
with "a_ext.gpr";
project B_Ext extends "b.gpr" is
end B_Ext;
```

CHAPTER

SIX

SOURCE RESOLUTION

This chapter describes how the GPR project model resolves source files: how sources are discovered, which basenames are valid, how conflicts are handled, and how project extension overrides inherited sources.

6.1 Source discovery

For each non-abstract, non-aggregate project view, GPR tools scan every directory listed in `Source_Dirs` in declaration order. The resulting candidate set is then filtered by:

- `Source_Files` / `Source_List_File` - restrict to an explicit list.
- `Excluded_Source_Files` / `Excluded_Source_List_File` - remove specific files.
- The `Naming` package - controls which file name extensions and patterns identify sources of each language.

Sources inherited from an extended project are added to this set before filtering.

6.2 Basename uniqueness

Every source file is identified by its *basename* - the simple file name without directory components. When the same basename appears in more than one source directory, the outcome depends on how those directories entered the project:

- **Different values in ``Source_Dirs``** - the directory corresponding to the earlier value takes precedence; no error is reported. For example, if `Source_Dirs` is `("src/a", "src/b")` and both contain `util.adb`, the file from `src/a` is used.
- **Same value in ``Source_Dirs``** - an error is reported. This can occur when a single value is a recursive pattern such as `"src/**"`: if two subdirectories matched by that pattern both contain a file with the same basename, the conflict cannot be resolved by ordering and is flagged as an error.

6.3 Extending projects and source shadowing

When project *B* extends project *A*, *B* inherits all of *A*'s sources. If *B* also declares a source whose basename matches one inherited from *A*, *B*'s version shadows the inherited one - the extending project's source always takes precedence. This is not treated as a basename clash: overloading an inherited source is the intended mechanism for selectively replacing individual source files in an extension without copying the entire project.

To remove an inherited source without providing a replacement, list it in `Excluded_Source_Files` or `Excluded_Source_List_File`.

6.4 `extends all`

With `extends all`, the extending project inherits sources from the full transitive import closure of the extended project, not only from the immediate extended project. The same shadowing rules apply: a source declared in the extending project overrides the corresponding inherited source from any project in the closure.

See *Project Extension* for a full description of `extends all` semantics.

6.5 Multi-unit Ada sources

A single Ada source file may contain more than one compilation unit (the `separate` construct, or an explicit `Naming` mapping via `Spec` / `Body`). The project model tracks sources at the basename level; unit-to-file mappings are resolved during source analysis. Basename uniqueness rules still apply at the file level regardless of how many units a file contains.

CHAPTER

SEVEN

ATTRIBUTES

Attributes communicate build properties to GPR tools. They are declared with the `for ... use` syntax described in *Project File Language*; this chapter lists all predefined attributes and their semantics.

Default values

Every attribute has a default value that applies when no declaration is present. The default is indicated in each attribute's description below.

Interaction with the configuration project

When an attribute is declared in the configuration project but not in the user project, the user project inherits the configuration value.

When a single-value attribute is declared in both, the user project's declaration takes precedence.

For list attributes marked **configuration concatenable**, the final value is the concatenation of the configuration project's value followed by the user project's value.

Reading attribute values

Attribute values may be referenced in expressions anywhere in a project file. If an attribute has not been set, its value defaults as described in each attribute's entry. For the reference syntax see *Project File Language*.

How to read the attribute descriptions

Each attribute entry below indicates:

- **Type** - `single` (string) or `list` (string list).
- **Read-only** - the attribute is set by the build system; user declarations are forbidden.
- **Indexed by** - the kind of index accepted; see *Project File Language* for the full description of each index kind. Possible values:
  - *language* - case-insensitive language identifier
  - *file name* - simple file name without directory components
  - *source glob* - simple file name or glob pattern; case sensitivity is host-dependent
  - *source glob or language* - either a source glob or a language identifier; resolution priority: exact file name, glob, language, `others`
  - *unit* - Ada unit name, case-insensitive
  - *string* - arbitrary string key
  - *external reference* - name of an external variable
- **Others index allowed** - the `others` index is accepted as a catch-all for this attribute.

- **Configuration concatenable** - for list attributes, the final value is the configuration value concatenated with the user value.
- **Inheritance** - by default, attributes are inherited from extended projects. Deviations are noted as *not inherited from extended* or *concatenated from extended*.

In addition to the attributes listed here, individual tools may define their own attributes in standard or tool-specific packages; refer to each tool's documentation.

7.1 Project Level Attributes

- **Configuration - Archives**
  - **Archive_Builder**: list value, not inherited from extended project

    Warning

    Empty value is not supported yet in non GNATcoll.Project-based tools

    Name of the application used to create a static library (archive), followed by its required options. If the value is empty, the object files listed in the archive recipe are copied to the library directory instead.
  - **Archive_Builder_Append_Option**: list value, not inherited from extended project

    Options passed to the archive builder when appending files to an existing archive.
  - **Archive_Indexer**: list value, not inherited from extended project

    Name of the archive indexer executable, followed by its required options.
  - **Archive_Prefix**: single value, not inherited from extended project

    Prefix for archive file names. Defaults to `"lib"`.
  - **Archive_Suffix**: single value, not inherited from extended project

    Extension for archive file names. Defaults to `".a"`.
  - **Library_Partial_Linker**: list value, not inherited from extended project

    Name of the partial linker executable, followed by its required options. An empty list disables partial linking.
- **Directories**
  - **Create_Missing_Dirs**: single value

    Applies to the main project only. When set to `"true"`, automatically creates missing object, library, and executable directories. Accepted values (case-insensitive): `"true"` or `"false"`.
  - **Exec_Dir**: single value, not inherited from extended project

    Directory where executables are placed.
  - **Gpr_Registry_Dirs**: list value, not inherited from extended project

    List of directories containing JSON files with external package and attribute definitions. All JSON files found in the listed directories are loaded at project tree initialization.
  - **Ignore_Source_Sub_Dirs**: list value, not inherited from extended project

    List of simple names or patterns for subdirectories to exclude from the source directory list, including their own subdirectories.

  - **Inherit_Source_Path**: list value, indexed by a language

    Indexed by language name. Lists additional languages whose source directories are included in the source search path of the indexed language.

  - **Object_Dir**: single value, not inherited from extended project

    Directory where the compiler places object files.

  - **Source_Dirs**: list value, not inherited from extended project

    List of source directories for the project.

- **Configuration - General**
  - **Config_Prj_File**: single value

    The main configuration project file.

  - **Default_Language**: single value, not inherited from extended project

    Default language for the project when `Languages` is not declared. The value is a case-insensitive language name.

  - **Disable_Linking**: single value, not inherited from extended project

    Indicates whether linking is disabled on the platform. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Object_Generated**: single value, indexed by a language

    Indexed by language name. Indicates whether compiling a source of the language produces an object file. Accepted values (case-insensitive): `"true"` (default) or `"false"`.

  - **Objects_Linked**: single value, indexed by a language

    Indexed by language name. Indicates whether object files produced for the language are linked into executables. Accepted values (case-insensitive): `"true"` (default) or `"false"`.

  - **Required_Toolchain_Version**: single value, indexed by a language

    Indexed by language name. Expected value for `Toolchain_Version` for the language, typically set in an auto-generated configuration project. Project processing aborts if `Required_Toolchain_Version` and `Toolchain_Version` do not match.

  - **Run_Path_Option**: list value, not inherited from extended project

    Switches used to specify the run-path option when linking an executable.

  - **Run_Path_Origin**: single value, not inherited from extended project

    String that may substitute the executable directory path in run-path options.

  - **Runtime**: single value, indexed by a language

    Indexed by language name. Applies to the main project and any extended projects. Specifies the runtime directory for the language's compiler. When `--RTS` is specified on the command line, `'Runtime` for that language reflects the command-line value instead.

  - **Runtime_Dir**: single value, indexed by a language

    Indexed by language name. Path of the runtime directory for the language.

  - **Runtime_Library_Dir**: single value, indexed by a language, not inherited from extended project

    Indexed by language name. Path of the directory containing runtime libraries. Obsolete.

  - **Runtime_Source_Dir**: single value, indexed by a language

    Indexed by language name. Path of the directory containing runtime library sources. Obsolete.

  - **Runtime_Source_Dirs**: list value, indexed by a language

    Indexed by language name. Paths of the directories containing runtime library sources. Not normally declared directly.

  - **Separate_Run_Path_Options**: single value, not inherited from extended project

    Indicates whether multiple separate run-path options may be passed to the linker. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Target**: single value

    Applies to the main project only. Name of the target platform. When `--target=` is specified on the command line, `'Target` reflects that value instead.

  - **Toolchain_Version**: single value, indexed by a language

    Indexed by language name. Records the version of the toolchain used for the language.

  - **Toolchain_Name**: single value, indexed by a language

    Indexed by language name. Applies to the main project and any extended projects. Identifies the toolchain used for the language.

  - **Toolchain_Description**: single value, indexed by a language

    Obsolescent. No longer used.

- **Source Files**

  - **Excluded_Source_Files**: list value, not inherited from extended project

    List of simple file names excluded from the project's sources. Use this to remove sources that are inherited or found in source directories but should not be part of this project.

  - **Excluded_Source_List_File**: single value, not inherited from extended project

    Name of a text file listing file simple names to exclude from the project's sources.

  - **Interfaces**: set value, case-sensitive

    List of file names that form the interface of the project.

  - **Locally_Removed_Files**: list value, not inherited from extended project

    Obsolescent. Equivalent to `Excluded_Source_Files`.

  - **Source_Files**: list value, not inherited from extended project

    List of source file simple names for the project.

  - **Source_List_File**: single value, not inherited from extended project

    Name of a text file listing source file simple names, one per line.

- **Aggregate Projects**

  - **External**: single value, indexed by an external reference

    Indexed by external reference name. Sets the value of the external reference to use when parsing the aggregated projects.

  - **Project_Files**: list value, not inherited from extended project

    List of project files aggregated by this project.

  - **Project_Path**: list value, not inherited from extended project

    Additional directories added to the project search path when locating aggregated projects.

- **General**
  - **Externally_Built**: single value, not inherited from extended project

    Marks the project as externally built. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Languages**: set value, case-insensitive

    List of languages used in the project's sources.

  - **Main**: list value

    List of main sources for the executables.

  - **Name**: single value, read-only, not inherited from extended project

    The name of the project.

  - **Project_Dir**: single value, read-only, not inherited from extended project

    Path of the project directory.

  - **Roots**: list value, indexed by a source glob or language

    Indexed by source file name, language name, or `"*"`. Lists units from the main project that must be bound and linked, together with their closures, into the indexed executable. Resolution order: exact source file name, then language name, then `"*"`.

  - **Warning_Message**: single value

    Emits a user-defined warning message during project processing.

- **Libraries**
  - **Leading_Library_Options**: list value, configuration concatenable, not inherited from extended project

    Options placed at the beginning of the linker command line when building a shared library.

  - **Library_Auto_Init**: single value, not inherited from extended project

    Controls whether a Stand-Alone Library is automatically initialized at load time. Accepted values (case-insensitive): `"true"` or `"false"`. Defaults to the value of `Library_Auto_Init_Supported`. Cannot be set to `"true"` when `Library_Auto_Init_Supported` is `"false"`.

  - **Library_Dir**: single value

    Directory where the library is placed. Must be declared in every library project.

  - **Library_Encapsulated_Options**: list value, configuration concatenable, not inherited from extended project

    Additional link options required when building an encapsulated Stand-Alone Library.

  - **Library_Encapsulated_Supported**: single value, not inherited from extended project

    Indicates whether encapsulated Stand-Alone Libraries are supported on the platform. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Library_Interface**: set value, case-insensitive

    List of unit names that form the interface of a Stand-Alone Library.

  - **Library_Kind**: single value, not inherited from extended project

    Accepted values (case-insensitive): `"static"` for archives (default), `"static-pic"` for archives of position-independent code, or `"dynamic"` / `"relocatable"` for shared libraries.

  - **Library_Name**: single value

    Name of the library. Must be declared or inherited in every library project.

  - **Library_Options**: list value, configuration concatenable, not inherited from extended project

    Additional switches ("last switches") passed when linking a shared library or a static standalone library. For a simple static library or when partial linking is disabled, values are restricted to paths to object files (absolute or relative to the object directory).

  - **Library_Reference_Symbol_File**: single value, not inherited from extended project

    Name of the reference symbol file.

  - **Library_Rpath_Options**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Compiler options used to determine the run-path entry for a shared library built from sources of the language. The compiler is invoked with these options; its output is a shared library path whose directory is added to the run-path option when linking.

  - **Library_Src_Dir**: single value, not inherited from extended project

    Directory where copies of the interface sources of a Stand-Alone Library are placed.

  - **Library_Standalone**: single value, not inherited from extended project

    Accepted values (case-insensitive): `"standard"` for non-encapsulated Stand-Alone Libraries, `"encapsulated"` for encapsulated Stand-Alone Libraries, or `"no"` for a regular (non-SAL) library project.

  - **Library_Symbol_File**: single value, not inherited from extended project

    Name of the library symbol file.

  - **Library_Symbol_Policy**: single value, not inherited from extended project

    Accepted values (case-insensitive): `"restricted"` or `"unrestricted"`.

  - **Library_Version**: single value, not inherited from extended project

    Internal version string of the library file (for shared library versioning).

    For more details see the *attribute semantics*.

- **Configuration - Shared Libraries**

  - **Library_Auto_Init_Supported**: single value, not inherited from extended project

    Indicates whether automatic initialization of Stand-Alone Libraries is supported on the platform. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Library_Install_Name_Option**: single value, not inherited from extended project

    Option prefix that, concatenated with the library file path, sets the install name of a shared library at link time.

  - **Library_Major_Minor_Id_Supported**: single value, not inherited from extended project

    Indicates whether major/minor version identifiers in shared library names are supported on the platform. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Library_Version_Switches**: list value, configuration concatenable, not inherited from extended project

    Switches used to set the internal (soname) name of a shared library.

  - **Shared_Library_Minimum_Switches**: list value, not inherited from extended project

    Minimum required switches when linking a shared library.

  - **Shared_Library_Prefix**: single value, not inherited from extended project

    Prefix for shared library file names. Defaults to `"lib"`.

  - **Shared_Library_Suffix**: single value, not inherited from extended project

    Extension for shared library file names. Defaults to `".so"`.

  - **Symbolic_Link_Supported**: single value, not inherited from extended project

    Indicates whether symbolic links are supported on the platform. Accepted values (case-insensitive): `"true"` or `"false"` (default).

- **Configuration - Libraries**
  - **Library_Builder**: single value, not inherited from extended project

    Path of the application used to build libraries (typically `gprlib`).

  - **Library_Support**: single value, not inherited from extended project

    Accepted values (case-insensitive): `"none"` (default), `"static_only"`, or `"full"`.

  - **Linker_Lib_Dir_Option**: single value, not inherited from extended project

    Option used to add a library directory to the linker search path.

7.2 Package Binder Attributes

- **General**
  - **Default_Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by language name. Switches passed to the binder for the language when no `Switches` entry matches.

  - **Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by source file name or language name. Switches passed to the binder for the matching executable or language.

- **Configuration - Binding**
  - **Driver**: single value, indexed by a language

    Indexed by language name. Name of the binder executable for the language.

  - **Objects_Path**: single value, indexed by a language

    Indexed by language name. Name of the environment variable that holds the object directory search path for the binder.

  - **Prefix**: single value, indexed by a language

    Indexed by language name. Prefix applied to binder exchange file names for the language, allowing distinct exchange files when binding multiple languages.

  - **Required_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Mandatory switches always passed to the binder for the language.

7.3 Package Builder Attributes

- **Default_Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

  Indexed by language name. Builder switches used when building an executable for the language and no `Switches` entry applies.

- **Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

  Indexed by source file name or language name. Builder switches applied when building the matching executable.

- **Executable**: single value, indexed by a file name

  Indexed by executable source file name. Simple file name of the resulting executable.

- **Executable_Suffix**: single value

  Extension appended to executable file names. Defaults to `.exe` on Windows, empty on other platforms.

- **Global_Compilation_Switches**: list value, indexed by a language, "others" index allowed, configuration concatenable

  Indexed by language name. Compilation switches applied globally when building an executable.

- **Global_Config_File**: single value, indexed by a language

  Indexed by language name. File name of a configuration file passed to the compiler for every source of the language in the project tree.

- **Global_Configuration_Pragmas**: single value

  File name of a configuration pragmas file passed to the Ada compiler for every Ada source in the project tree.

7.4 Package Clean Attributes

- **Switches**: list value, configuration concatenable

  Applies to the main project only. Switches passed to the cleaning application.

- **Artifacts_In_Exec_Dir**: list value

  List of file name patterns (regular expressions) deleted by `gprclean` in the main project's executable directory.

- **Artifacts_In_Object_Dir**: list value

  List of file name patterns (regular expressions) deleted by `gprclean` in the project's object directory.

- **Object_Artifact_Extensions**: list value, indexed by a language

  Indexed by language name. Extensions of files derived from source file names that `gprclean` removes from the object directory.

- **Source_Artifact_Extensions**: list value, indexed by a language

  Indexed by language name. Extensions of files derived from object file names that `gprclean` removes from the object directory.

7.5 Package Compiler Attributes

- **General**
  - **Default_Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by language name. Switches passed to the compiler for the language when no `Switches` entry applies.
  - **Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by source file name or language name. Switches passed to the compiler for the matching source or language.
  - **Local_Config_File**: single value, indexed by a language

    Indexed by language name. File name of a configuration file passed to the compiler for every source of the language in this project.
  - **Local_Configuration_Pragmas**: single value

    File name of a configuration pragmas file passed to the Ada compiler for every Ada source in this project.
- **Configuration - Compiling**
  - **Driver**: single value, indexed by a language

    Indexed by language name. Name of the compiler executable for the language.
  - **Required_Switches**: list value, indexed by a language, configuration concatenable

    Equivalent to `Leading_Required_Switches`.
  - **Dependency_Kind**: single value, indexed by a language

    Indexed by language name. Accepted values (case-insensitive): `"none"` (default), `"makefile"`, `"ali_file"`, or `"ali_closure"`.
  - **Language_Kind**: single value, indexed by a language

    Indexed by language name. Accepted values (case-insensitive): `"file_based"` (default) or `"unit_based"`.
  - **Leading_Required_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Mandatory switches placed at the beginning of the compiler command line for the language.
  - **Multi_Unit_Object_Separator**: single value, indexed by a language

    Indexed by language name. String inserted in the object file name before the unit index when compiling a unit from a multi-unit source.
  - **Multi_Unit_Switches**: list value, indexed by a language

    Indexed by language name. Switches used to identify the unit to compile in a multi-unit source. The unit's index within the source is appended to the last switch in the list.
  - **Object_File_Suffix**: single value, indexed by a language

    Indexed by language name. Extension of object files produced by the compiler. Defaults to the platform's standard object file extension.

  - **Object_File_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass the output object file path to the compiler. Defaults to `"-o"`.

  - **Source_File_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches placed immediately before the source file path when invoking the compiler.

  - **Trailing_Required_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Mandatory switches placed at the end of the compiler command line for the language.

- **Configuration - Config Files**
  - **Config_Body_File_Name**: single value, indexed by a language

    Indexed by language name. Template for identifying a body-specific configuration entry in a configuration file.

  - **Config_Body_File_Name_Index**: single value, indexed by a language

    Indexed by language name. Template for identifying a body-specific configuration entry for a unit within a multi-unit source in a configuration file.

  - **Config_Body_File_Name_Pattern**: single value, indexed by a language

    Indexed by language name. Template matching all body configuration entries for the language in a configuration file.

  - **Config_File_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass a configuration file to the compiler.

  - **Config_File_Unique**: single value, indexed by a language

    Indexed by language name. When `"true"`, only one configuration file is passed to the compiler. Accepted values (case-insensitive): `"true"` or `"false"` (default).

  - **Config_Spec_File_Name**: single value, indexed by a language

    Indexed by language name. Template for identifying a spec-specific configuration entry in a configuration file.

  - **Config_Spec_File_Name_Index**: single value, indexed by a language

    Indexed by language name. Template for identifying a spec-specific configuration entry for a unit within a multi-unit source in a configuration file.

  - **Config_Spec_File_Name_Pattern**: single value, indexed by a language

    Indexed by language name. Template matching all spec configuration entries for the language in a configuration file.

- **Configuration - Dependencies**
  - **Dependency_Driver**: list value, indexed by a language

    Indexed by language name. Executable, followed by required switches, used to generate dependency files for sources of the language.

  - **Dependency_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to specify the dependency file to the compiler, when the dependency kind is file-based and `Dependency_Driver` is not set.

- **Configuration - Search Paths**
  - **Include_Path**: single value, indexed by a language

    Indexed by language name. Name of the environment variable that holds all source search directories for the compiler.
  - **Include_Path_File**: single value, indexed by a language

    Indexed by language name. Name of the environment variable whose value is a text file listing the source search directories for the compiler.
  - **Include_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass a source search directory to the compiler.
  - **Object_Path_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass a text file listing object directories to the compiler. When not declared, no such file is created.
- **Configuration - Mapping Files**
  - **Mapping_Body_Suffix**: single value, indexed by a language

    Indexed by language name. Suffix used in mapping files to mark a source as a body.
  - **Mapping_File_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass a mapping file to the compiler.
  - **Mapping_Spec_Suffix**: single value, indexed by a language

    Indexed by language name. Suffix used in mapping files to mark a source as a spec.
- **Configuration - Response Files**
  - **Max_Command_Line_Length**: single value

    Maximum number of characters in the compiler command line before a response file is used.
  - **Response_File_Format**: single value, indexed by a language

    Indexed by language name. Format of the response file generated when the compiler command line exceeds `Max_Command_Line_Length`. Accepted values (case-insensitive): `"none"`, `"gnu"`, `"object_list"`, `"gcc_gnu"`, `"gcc_option_list"`, or `"gcc_object_list"`.
  - **Response_File_Switches**: list value, indexed by a language, configuration concatenable

    Indexed by language name. Switches used to pass a response file to the compiler.

7.6 Package Gnatls Attributes

- **Switches**: list value

  Applies to the main project only. Switches passed to `gnatls`.

7.7 Package Install Attributes

- **Prefix**: single value

  Installation destination directory. A relative path is resolved against the global prefix (`--prefix` option value, or the default installation prefix).

- **Active**: single value

  Controls whether the project is installed. Setting this to `"false"` (case-insensitive) skips installation; all other values enable it.

- **Artifacts**: list value, indexed by a file name

  Indexed by installation directory. Lists non-source files to install. Relative directory indexes are resolved against `Prefix`. If the same file name appears more than once, the last entry wins. A missing artifact produces a warning.

- **Exec_Subdir**: single value

  Subdirectory under `Prefix` where executables are installed. Default: `bin/`.

- **Install_Name**: single value

  Name used to record the installation. Defaults to the project name without extension.

- **Install_Project**: single value

  Controls whether a relocatable project file is generated and installed. Accepted values: `"true"` (default) or `"false"`.

- **Lib_Subdir**: single value

  Subdirectory under `Prefix` where library files are installed. Default: `lib/`.

- **Mode**: single value

  Installation mode. Accepted values (case-insensitive): `"dev"` (default) - full developer installation including sources, ALI files, and libraries; `"usage"` - end-user installation including only shared libraries and executables.

- **Project_Subdir**: single value

  Subdirectory under `Prefix` where the generated GPR project file and installation manifests are placed. Default: `share/gpr/`.

- **Required_Artifacts**: list value, indexed by a file name

  Same as `Artifacts`, but a missing file raises an error instead of a warning.

- **Side_Debug**: single value

  When set to `"true"`, strips debug symbols from executables and shared libraries and writes them to a side file with a `.debug` extension. Accepted values (case-insensitive): `"false"` (default) or `"true"`.

- **Sources_Subdir**: single value

  Subdirectory under `Prefix` where source files are installed. Default: `include/`.

7.8 Package Linker Attributes

- **General**
  - **Default_Switches**: list value, indexed by a source glob or language, “others” index allowed, configuration concatenable

    Warning

    Index “others” is not supported yet in gprbuild or GNATcoll.Projects-based tools

    Indexed by language name. Linker switches used when no `Switches` entry applies for an executable of the language.

  - **Required_Switches**: list value, configuration concatenable

    Mandatory switches always passed to the linker.

  - **Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by source file name or language name. Switches passed to the linker for the matching executable.

  - **Group_End_Switch**: single value

    Switch that ends a link group.

  - **Group_Start_Switch**: single value

    Switch that begins a link group (a set of libraries linked with recursive symbol resolution).

  - **Leading_Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by source file name or language name. Switches placed at the beginning of the linker command line for the matching executable.

  - **Linker_Options**: list value, configuration concatenable

    Additional linker switches for imported subsystems. Ignored when set in the main project; applied when set in any directly or indirectly imported project. Complements `Linker'Switches` in the main project. Use this to declare per-subsystem external library dependencies instead of repeating them in every top-level project.

  - **Trailing_Switches**: list value, indexed by a source glob or language, "others" index allowed, configuration concatenable

    Indexed by source file name or language name. Switches placed at the end of the linker command line for the matching executable. These may override `Required_Switches`.

  - **Unconditional_Linking**: single value, indexed by a language

    Indexed by language name. When set, the link phase always explicitly includes all object files produced for the language.

- **Configuration - Linking**
  - **Driver**: single value

    Name of the linker executable.

- **Configuration - Response Files**
  - **Max_Command_Line_Length**: single value

    Maximum number of characters in the linker command line before a response file is used.

  - **Response_File_Format**: single value

    Format of the response file generated when the linker command line exceeds `Max_Command_Line_Length`. Accepted values (case-insensitive): `"none"`, `"gnu"`, `"object_list"`, `"gcc_gnu"`, `"gcc_option_list"`, or `"gcc_object_list"`.

  - **Response_File_Switches**: list value, configuration concatenable

    Switches used to pass a response file to the linker.

7.9 Package Naming Attributes

- **Body**: single value, indexed by a unit

  Indexed by unit name. File name of the unit's body.

- **Body_Suffix**: single value, indexed by a language

  Warning

  Also has case-insensitive values in gprbuild and GNATcoll.Project-based tools

  Indexed by language name. File name extension for body files of the language.

- **Casing**: single value

  Accepted values (case-insensitive): `"lowercase"`, `"uppercase"`, or `"mixedcase"`. Specifies the expected casing of Ada source file names.

- **Dot_Replacement**: single value

  String that replaces the dot separator in Ada unit names when forming source file names.

- **Implementation**: single value, indexed by a unit

  Equivalent to `Body`.

- **Implementation_Exceptions**: list value, indexed by a language

  Indexed by language name. List of body files for the language that do not follow the standard naming scheme and may reside outside the declared source directories.

- **Implementation_Suffix**: single value, indexed by a language

  Warning

  Also has case-insensitive values in gprbuild and GNATcoll.Project-based tools

  Equivalent to `Body_Suffix`.

- **Separate_Suffix**: single value

  File name extension for Ada subunit files.

- **Spec**: single value, indexed by a unit

  Indexed by unit name. File name of the unit's spec.

- **Spec_Suffix**: single value, indexed by a language

  Warning

  Also has case-insensitive values in gprbuild and GNATcoll.Project-based tools

  Indexed by language name. File name extension for spec files of the language.

- **Specification**: single value, indexed by a unit

  Equivalent to `Spec`.

- **Specification_Exceptions**: list value, indexed by a language

  Indexed by language name. List of spec files for the language that do not follow the standard naming scheme and may reside outside the declared source directories.

- **Specification_Suffix**: single value, indexed by a language

  Warning

  Also has case-insensitive values in gprbuild and GNATcoll.Project-based tools

  Equivalent to `Spec_Suffix`.

CHAPTER

EIGHT

KNOWLEDGE BASE

The **knowledge base** (KB) is a collection of XML files that describe the compilers, targets, and runtimes that GPR tools can work with. It supplies the toolchain-specific information - compiler driver names, default switches, dependency-file formats, library conventions, target normalization rules - that all GPR tools need to build, install, and inspect projects.

8.1 Embedded KB

In all current GPR tools, the knowledge base is **embedded directly into the tool binary**. Every tool carries a complete, self-contained copy of the KB and can operate with no KB files present on disk.

A reference copy of the KB files is still installed under `<prefix>/share/gprconfig/` alongside the tools, but it is not consulted at run time unless explicitly requested with `--db` (see *Run-time KB selection* below).

8.2 KB structure

The KB is a set of XML files. Each file may contain:

- **Compiler descriptions** (`<compiler_description>`) - identify a compiler by executable name (which may be a regular expression), specify how to extract its version and target, and enumerate the languages and runtime variants it supports.
- **Target sets** (`<targetset>`) - group target strings under a canonical name used throughout the project model.
- **Fallback targets** (`<fallback_targets>`) - define chains of alternative targets tried when no exact match is found.
- **Configuration blocks** (`<configuration>`) - conditional GPR package settings (`Compiler`, `Binder`, `Linker`, `Naming`, etc.) emitted into the generated configuration project when a matching compiler/target/ host combination is selected.

Entity files (`.ent`) factoring out common patterns are referenced by the XML files to avoid duplication across compiler families and platform variants. An XSD schema file is also provided for validation.

8.3 Run-time KB selection

All project-based GPR tools and GPRconfig accept the following switches to control which KB is used at run time:

`--db` *dir*
: Parse *dir* as an additional KB directory and merge its contents with the embedded KB. May be repeated. Use this to add support for a proprietary compiler or to override specific compiler descriptions.

`--db-`
: Disable the embedded KB entirely. Only the directories supplied via `--db` are loaded. Use this when a completely custom KB is required, independent of the built-in one.

With neither switch, only the embedded KB is used.

8.4 Customizing the KB

The most common reason to supply a custom KB chunk is to add a proprietary or in-house compiler that the standard KB does not know about. To do so:

1. Create a new `.xml` file following the KB schema. The installed reference copy under *<prefix>*`/share/gprconfig/` serves as a guide and contains representative examples for many compiler families.
2. Place it in a dedicated directory, e.g. `/opt/mycompiler/gpr/`.
3. Pass `--db /opt/mycompiler/gpr/` to GPRconfig and any other tool that needs to detect the compiler.

To replace a standard compiler description rather than extending it, combine `--db-` with `--db` to load only the custom directory.

8.5 KB validation

GPRconfig accepts `--validate` to check all loaded KB files against the XSD schema before use. This is useful when developing new KB chunks:

```
gprconfig --validate --db /opt/mycompiler/gpr/ --batch \
  --config language:ada
```

CHAPTER

NINE

COMMON COMMAND-LINE OPTIONS

GPRbuild, GPRclean, GPRinstall, GPRname, GPRls, and GPRinspect all operate on a GPR project tree and share a common set of command-line options for project loading, configuration, and diagnostics. Tool-specific options are documented in each tool's own chapter.

GPRconfig is not covered here: it works directly with the compiler knowledge base rather than a project tree and has its own independent option set (see *GPRconfig Reference*).

Tip

When developing a custom project-aware tool, support most if not all of these options to ensure a consistent user experience across the GPR tool suite.

9.1 Project file

Every project-based GPR tool requires a project file. It may be supplied directly with `-P` or located automatically: the tool uses `default.gpr` in the current directory, or the sole `.gpr` file present if there is exactly one. Relative paths are searched against the current directory, then against *GPR_PROJECT_PATH_FILE*, *GPR_PROJECT_PATH*, and *ADA_PROJECT_PATH* (in that order).

GPR_PROJECT_PATH_FILE, when set, names a text file containing one project directory per line. *GPR_PROJECT_PATH* and *ADA_PROJECT_PATH* contain colon-separated directory lists.

9.2 Project and configuration switches

`-P` *proj*
: Specify the main project file. The space between `-P` and the name is optional.

`-aP` *dir*
: Add *dir* to the project search path.

`-X`*NAME=value*
: Set external reference *NAME* to *value* for use in project files.

`--no-project`
: Do not use any project file in the current directory; use the default project from `<prefix>/share/gpr`.

`--implicit-with=`*proj.gpr*
: Add the given project as an implicit `limited with` to every project in the tree.

`--config=`*file.cgpr*
: Use *file.cgpr* as the configuration project. The file must exist.

--autoconf=*file.cgpr*
Use *file.cgpr* as the configuration project, invoking GPRconfig to create it if it does not yet exist.

--target=*name*
Specify the target for cross-platform builds. Mutually exclusive with `--config` and `--autoconf`.

--RTS[:*lang*]=*runtime*
Specify the runtime directory for *lang* (or Ada if *lang* is omitted).

--db *dir*
Parse *dir* as an additional knowledge base directory.

--db-
Do not load the standard knowledge base.

--relocate-build-tree[=*dir*]
Relocate all object, library, and executable directories under the current working directory (or *dir*). See *Out-of-Tree Builds*.

--root-dir=*dir*
Root directory for `--relocate-build-tree` relocation. Defaults to the main project directory; override when artifact directories lie outside it. See *Out-of-Tree Builds*.

--subdirs=*dir*
Append *dir* to all object, library, and executable directories, creating them as needed.

--src-subdirs=*dir*
For each project, prepend *obj-dir/dir* to the source directories, where *obj-dir* is the project's object directory. Useful for temporary source overrides such as instrumentation.

9.3 Output and diagnostics

-q
Quiet: display only errors.

-v
Verbose: display full paths and all spawned-process options.

-F
Use full project path names in brief error messages.

-we
Treat project-file warnings as errors.

-ws
Suppress project-file warnings. Does not affect compiler warnings.

-wn
Restore default warning behavior (cancels `-we` or `-ws`).

Note

All switches documented in this chapter must be given on the command line. They are not accepted in the `Switches` attribute of any tool's project package (`Builder'Switches`, `Clean'Switches`, `Install'Switches`, etc.).

CHAPTER

TEN

OUT-OF-TREE BUILDS

By default, GPR tools write build artifacts (object files, libraries, executables) to the directories declared in the project file (`Object_Dir`, `Library_Dir`, `Exec_Dir`). An *out-of-tree build* places all artifacts under a single external directory without modifying the project files, using `--relocate-build-tree`.

When `--relocate-build-tree=`*dir* is given, each artifact directory is mirrored under *dir*. For a project file at `/src/workspace/app/app.gpr` with `Object_Dir = "obj"`, the relocated object directory becomes *dir*`/src/workspace/app/obj`.

This is useful to keep build artifacts separate from source files: to maintain a clean version-controlled source tree, to build a read-only source tree (third-party code, mounted file systems), or to run multiple independent builds of the same sources in parallel without interference.

Note

`--relocate-build-tree` has no effect on artifact directories declared with absolute paths in the project file. Only relative `Object_Dir`, `Library_Dir`, and `Exec_Dir` values are relocated.

When the project tree contains `with` clauses that reference projects outside the main project's directory (for example `with "../lib/lib.gpr"`), their artifact directories would by default be relocated outside *dir*. Use `--root-dir=`*root* to declare the common ancestor of all project directories in the tree; the relocation then maps every artifact directory to a path strictly under *dir*.

Example: a tree rooted at `/src/workspace` with two projects:

```
/src/workspace/
├── app/
│   └── app.gpr      (with "../lib/lib.gpr")
└── lib/
    └── lib.gpr
```

Building out-of-tree:

```
$ gprbuild -P /src/workspace/app/app.gpr \
           --relocate-build-tree=/tmp/build \
           --root-dir=/src/workspace
```

Artifact directories are created under `/tmp/build/app/` and `/tmp/build/lib/`, leaving the source tree untouched. Other GPR tools (`gprclean`, `gprls`, `gprinstall`) accept the same switches and operate on the relocated artifacts.

The `--relocate-build-tree` and `--root-dir` switches are documented in *Common Command-Line Options*.

CHAPTER

ELEVEN

GPRBUILD REFERENCE

GPRbuild is a multi-language build tool for GPR project trees. It compiles sources, binds Ada programs, builds libraries, and links executables, using a persistent build database and checksum-based change detection for reliable incremental builds.

GPRbuild2, the current build engine, models the build as a directed acyclic graph (DAG) of actions and is the recommended engine.

11.1 Build Engine Selection

Two build engines are available. Set the *GNAT_GPR_ENGINE* environment variable, or pass `--gpr=`*n* on the command line (takes precedence):

1 / legacy
: The original GPRbuild engine. Current default.

2 / new
: GPRbuild2: DAG-based execution, checksum-based change detection, persistent build database. The recommended engine.

```
GNAT_GPR_ENGINE=2 gprbuild -P my_proj.gpr
gprbuild --gpr=2 -P my_proj.gpr
```

The rest of this chapter documents GPRbuild2 behavior and switches.

11.2 Command Line

11.2.1 Syntax

```
gprbuild [<proj>.gpr] [switches] [mains]
 {[-cargs[:<lang>] opts] [-bargs[:<lang>] opts]
  [-largs opts] [-kargs opts] [-gargs opts]}
```

11.2.2 Project file

See *Common Command-Line Options* for project file discovery rules and the project search path environment variables.

11.2.3 Main sources

Main source files to build may be listed on the command line by simple file name or full path. When a simple name is given, GPRbuild searches the entire project tree for a source whose basename matches. If the name matches sources in more than one project, GPRbuild reports an error. If no mains are given on the command line, GPRbuild uses the `Main` attribute of the root project. If that is also absent, no executable is built.

11.2.4 Pass-through option groups

Options may be forwarded to individual tools by preceding them with a section switch. Each section switch introduces options that apply up to the next section switch or end of the command line.

`-cargs`
All compilers (all languages).

`-cargs:`*lang*
The compiler for *lang* only.

`-bargs`
All binder drivers.

`-bargs:`*lang*
The binder driver for *lang* only.

`-largs`
The linker.

`-kargs`
GPRconfig (auto-configuration invocation).

`-gargs` / `-margs`
GPRbuild itself (useful after other section switches).

11.3 Switches

Project, configuration, and common diagnostics switches are documented in *Common Command-Line Options*. The sections below cover GPRbuild-specific switches.

11.3.1 Phase selection

`-c`
Include only compile actions in the DAG.

`-b`
Include only bind actions in the DAG.

`-l`
Include only link actions in the DAG.

Note

These switches are compatibility filters over GPRbuild2's action DAG. `-c` maps cleanly to compile actions. `-b` includes not only the binder invocation but also the compilation of the binder-generated Ada source, so it involves actual compile steps. `-l` may cover a range of intermediate actions beyond the final linker call: intermediate archive creation, partial linking, linker-options section generation, and similar housekeeping. In the legacy engine these stages were driven by separate tools (`gprbind` for `-b`, `gprlib` for `-l`); GPRbuild2 models them as ordinary actions in the DAG with no distinct stage boundaries, and `-b` / `-l` simply select the corresponding subsets.

-u
Compile only the source files named on the command line (or all sources of the main project if none named). No binding or linking.

-U
Compile only the source files named on the command line (or all sources of all projects if none named). No binding or linking.

-r
With `-c` or `-u` and no entry points: compile all sources of all projects recursively.

-z
Build without generating an OS-level entry point. Normally GPRbuild produces a `main` symbol that the operating system calls to start the program, driving Ada elaboration and invoking the main subprogram. With `-z`, that symbol is omitted so the resulting objects can be linked with a main subprogram written in another language — for example a C `main` that handles startup and then calls into Ada code.

11.3.2 Build behavior

-f
Force re-execution of all actions, ignoring stored signatures.

-j*num*
Maximum number of actions to run in parallel. `-j0` uses one job per CPU core. If GPRbuild is invoked from a Makefile and the invoking make provides a jobserver, GPRbuild2 acquires job slots from it rather than running freely: `-j` then acts as an upper bound on the slots requested, with actual parallelism capped by what the jobserver makes available.

-k
Keep executing independent actions after a failure. Actions that depend on a failed action are skipped.

-p / --create-missing-dirs
Force creation of object, library, and executable directories that lie outside the project directory. Directories relative to the project directory (including relocated ones via `--relocate-build-tree`) are created automatically without this switch. GPRbuild refuses by default to create directories outside the project tree — whether specified as absolute paths or as relative paths that escape the project directory (e.g. `../other`) — and `-p` overrides that restriction.

-R
Do not pass a run-path option to the linker, even when `Run_Path_Option` is declared.

-o *name*
Name the output executable *name*. Requires exactly one entry point.

-eI*nn*
Index of the main unit in a multi-unit source file.

--no-indirect-imports
After each successful compilation, verify that the source only uses files from directly imported projects. A violation invalidates the compilation artifacts.

--indirect-imports
Allow use of files from directly or indirectly imported projects (default). Cancels a preceding `--no-indirect-imports`.

--no-object-check
Do not verify that an object file was produced after compilation.

--no-split-units
Require all parts of an Ada compilation unit to come from the same project view.

`--no-sal-binding`
: Reuse binder artifacts from a previous build for Stand-Alone Libraries. Unsafe; may produce incorrect results if sources have changed.

`--restricted-to-languages=`*lang1*`,`*lang2*
: Include only compile actions for the listed languages in the DAG.

`--compiler-subst=`*lang*`,`*tool*
: Use *tool* instead of the configured compiler for *lang*.

`--temp-dir=[os|obj]`
: Control where GPRbuild places temporary files: `os` uses the system temporary directory; `obj` uses the object directory of the owning project.

`--unchecked-shared-lib-imports`
: Allow shared library projects to import non-shared-library projects.

11.3.3 Output and diagnostics

See *Common Command-Line Options* for the `-q`, `-v`, `-F`, `-we`, `-ws`, and `-wn` switches.

`-d`
: Display per-action progress lines.

`--no-warnings-replay` / `-n`
: Do not replay the warnings of actions that were skipped because they were up to date. By default, GPRbuild2 replays such warnings.

11.3.4 Build artifacts

`--build-script=`*file*
: Write a shell script to *file* that reproduces the build. This is a best-effort output: the generated script may not be portable and will only work reliably for simple cases. It nonetheless gives a useful overview of the commands GPRbuild executed.

`--create-map-file[=`*file*`]`
: Ask the linker to generate a map file. Defaults to the executable name with a `.map` extension.

`--keep-temp-files`
: Do not delete temporary files created during the build.

11.3.5 Compiler compatibility switches

The following switches are accepted for compatibility with Ada compilers and are forwarded to the Ada compiler:

`-g[`*opt*`]`, `-O[`*level*`]`, `-fno-inline`, `-fstack-check`, `-nostdinc`, `-nostdlib`

11.4 Build Execution

GPRbuild2 models the build as a directed acyclic graph (DAG) of *actions*. Each action is an atomic invocation of an external tool (compiler, binder, archiver, linker, etc.) with a defined set of inputs and outputs. Dependency edges encode execution order: for example, a link action depends on all compile actions whose objects it consumes.

After loading the project tree, GPRbuild2 populates the DAG from the project configuration and the set of *entry points* - executables declared via the `Main` attribute or on the command line, libraries, and any additional units specified via the `Roots` attribute. The process manager then traverses the DAG in topological order, running independent actions in parallel up to the `-j` limit.

Each action's up-to-dateness is determined by a *signature*: checksums over all inputs (source files, dependency closures, switches, configuration) and expected outputs. An action is re-executed only when its signature has changed since the last successful run. Signatures are persisted in a build database across invocations.

Action kinds include (non-exhaustive):

- **Compile** - invoke the language compiler on a single source file.
- **Ada bind** - invoke `gprbind` / `gnatbind` for an Ada entry point to produce the bind artifacts.
- **Post-bind** - compile the binder-generated Ada source.
- **Link** - invoke the linker to produce an executable or shared library.
- **Archive** - invoke the archive builder (`gprlib`) to produce a static library.

The `-c`, `-b`, and `-l` switches restrict which action kinds are included in the DAG for a given invocation.

11.5 Project package

The `Builder` package of the main project may provide default switches and global compilation switches that apply to every invocation. See *Package Builder Attributes* in the Attributes Reference for the full attribute list and descriptions.

The following switches are not accepted in `Builder'Switches` and must be given on the command line: `-o`, `-r`, `-u`, `-U`, `-z`, `--build-script`, `--compiler-subst`, `--no-sal-binding`, `--restricted-to-languages`, and `--temp-dir`. The common switches documented in *Common Command-Line Options* are likewise command-line only.

```
project My_App is
   package Builder is
      for Switches ("Ada") use ("-j4", "-k");
      for Global_Compilation_Switches ("Ada") use ("-O2");
   end Builder;
end My_App;
```

11.6 Exit Codes

0
: Success (warnings may have been issued).

1
: General error (invalid option, missing file, etc.).

4
: Underlying tool error (compiler, linker, etc.).

5
: Project parsing error.

7
: Critical internal error.

CHAPTER

TWELVE

GPRCONFIG REFERENCE

GPRconfig probes the host for available compilers and generates a configuration project (`.cgpr`) that describes the selected toolchains to all GPR tools. It can be used interactively or in batch mode from a script.

GPRconfig is independent of any GPR project tree. It does not accept the common project and configuration switches described in *Common Command-Line Options*; its option set is documented entirely in this chapter.

12.1 Command Line

12.1.1 Syntax

```
gprconfig [switches]
```

12.1.2 Compiler selection

GPRconfig scans `PATH` (and any directories added by `--config`) for compiler executables, cross-references them against the knowledge base, and builds a list of available compilers. In interactive mode the user selects which compilers to include; in batch mode (`--batch`) the selection is driven entirely by `--config` options on the command line.

At most one compiler per language may be selected. The output configuration project contains one `package Compiler` section per selected language.

12.2 Switches

12.2.1 Target and output

`--target=`*name*
: Select compilers for the given target. Use `all` to list compilers for every available target. Defaults to the native target. In batch mode, `--target=all` is silently ignored.

`-o`*file*
: Write the generated configuration project to *file*. Defaults to `default.cgpr` when no target is specified, or `<target>.cgpr` when a target is given.

`--show-targets`
: Print the list of targets for which at least one compiler is available in the knowledge base, then exit. The native target is marked `(native target)`.

`--mi-show-compilers`
: Print a machine-readable list of all available compilers and exit. Each compiler is described as a set of `key=value` pairs; compilers already selected for output are prefixed with `*`. Intended for IDE integration.

--show-known-compilers
Print the names of all compiler blocks defined in the knowledge base, then exit.

12.2.2 Compiler selection (batch mode)

--batch
Run in batch (non-interactive) mode. No prompts are displayed. The compilers to include must be specified entirely via `--config` options.

--config=*selector*
Pre-select a compiler. May be repeated, at most once per language.

The selector has two equivalent forms:

Positional: `language[,version[,runtime[,path[,name]]]]`

```
ada
ada,12.1
ada,,zfp
ada,,zfp,/opt/gnat/bin,gnat
```

Named: `key:value[,key:value...]`

```
language:ada
language:ada,version:12.1
language:ada,runtime:zfp
language:ada,runtime:zfp,path:/opt/gnat/bin,name:gnat
```

Available keys/fields:

language
Required. Language identifier (`ada`, `c`, `c++`, ...).

version
Optional. Compiler version string (e.g. `12.1`).

runtime
Optional. Runtime variant (e.g. `sjlj`, `zfp`, `ravenscar`).

path
Optional. Directory containing the compiler executable.

name
Optional. Compiler name as defined in the knowledge base (e.g. `GCC`, `GNAT`) or the base name of the compiler executable (e.g. `gcc`, `arm-elf-gnatmake`).

Omitted optional fields match any value. An empty string also matches any value, so `ada,,zfp` is equivalent to `language:ada,runtime:zfp`.

--fallback-targets
When no compiler is found for the requested target, also search for toolchains recorded as fallbacks, such as a compatible native target of a different architecture.

12.2.3 Knowledge base

--db *dir*
Parse *dir* as an additional knowledge base directory (may be repeated).

--db-
Do not load the standard knowledge base. Use together with `--db` to load a completely custom knowledge base.

`--validate`
: Validate the knowledge base before use and report any schema errors, then proceed normally.

12.2.4 Diagnostics

`-v`
: Verbose output. Repeat (`-v -v`) for even more detail, including internal knowledge base traces.

`-q`
: Quiet: suppress all output except errors.

12.3 Interactive mode

When run without `--batch`, GPRconfig displays the list of compilers found for the selected target (one per line, with an index), prompts the user to select or deselect entries by typing their index number, and writes the configuration when the user types `s`. Selecting a compiler for a language automatically deselects any previously selected compiler for the same language.

After saving, GPRconfig prints the equivalent `--batch` command line so that the same selection can be reproduced in scripts.

12.4 Configuration file format

The generated file is a valid GPR configuration project (`configuration` qualifier, `.cgpr` extension). It defines:

- `package Compiler` - compiler driver, default switches, dependency file format, and naming conventions for each selected language.
- `package Linker` - linker driver and default flags.
- `package Binder` - binder driver (for Ada).
- `package Archive_Builder` - archiver command.
- Top-level attributes - `Target`, `Canonical_Target`, `Runtime_Library_Dir`, `Object_Generated`, `Library_Support`, `Shared_Library_Prefix`, `Shared_Library_Suffix`, and related platform attributes.

The file is consumed by GPRbuild and the other project-based tools via `--config` or `--autoconf`. It is not a user project file and is not referenced by `with` clauses.

12.5 Exit Codes

`0`
: Success; configuration file written (or information displayed).

`1`
: No compiler found for the requested target or language, knowledge base error, or configuration generation failure.

`7`
: Invalid command-line option.

CHAPTER

THIRTEEN

GPRCLEAN REFERENCE

GPRclean removes the build artifacts produced by GPRbuild: object files, ALI files, libraries, executables, and binder-generated files. It reads the same project tree as GPRbuild so it knows exactly what was produced and where.

13.1 Command Line

13.1.1 Syntax

```
gprclean [-P<proj>.gpr] [switches] [mains]
```

13.1.2 Project file and common switches

See *Common Command-Line Options* for project file discovery rules, project and configuration switches, and common diagnostic switches.

13.1.3 Main sources

Zero or more main file names may be listed on the command line to restrict cleaning to the artifacts of those entry points only.

GPRclean applies the same source resolution rules as GPRbuild (see *GPRbuild Reference*): a simple name is looked up across the entire project tree, and an error is reported if it matches sources in more than one project. This ensures GPRclean removes exactly the artifacts that GPRbuild produced.

13.2 Switches

-r
: Clean all projects in the project tree, not only the main project.

-c
: Delete only compiler-generated files (object files, ALI files, and their associated artifacts). Skip link outputs (executables, libraries).

-p
: After removing artifacts, delete any object, library, or executable directories that are now empty.

-f
: Force deletion of files that are not owner-writable. GPRclean will attempt to set write permission on the file and retry the deletion.

-n
: Dry-run mode. List the files that would be deleted without actually removing them.

--autoconf=*file.cgpr*
: In addition to the common `--autoconf` behavior, also schedule *file.cgpr* itself for deletion after cleaning.

13.3 Project package

The `Clean` package of the main project may declare a `Switches` attribute whose value is a list of GPRclean switches. These are processed before command-line switches, so command-line switches take precedence. See *Package Clean Attributes* in the Attributes Reference for the full attribute list and descriptions.

The `--autoconf` switch and all switches from *Common Command-Line Options* must be given on the command line and are not accepted in `Clean'Switches`.

```
project My_App is
   package Clean is
      for Switches use ("-r", "-p");
   end Clean;
end My_App;
```

13.4 What is cleaned

For each project in scope (the main project, or all projects with `-r`), GPRclean deletes:

- Object files and dependency files (`*.o`, `*.d`, etc.) from the object directory.
- Language-specific compilation artifacts, as defined by `Source_Artifact_Extensions` and `Object_Artifact_Extensions` in the active configuration.
- ALI files and other Ada-specific artifacts produced by the compiler and binder.
- The persistent build database files written by GPRbuild2.
- Files matching the `Artifacts_In_Object_Dir` and `Artifacts_In_Exec_Dir` project attributes.
- Library files from the library directory (`Library_Dir`) and, for stand-alone libraries, the generated source directory (`Library_Src_Dir`).
- Executables listed in the `Main` attribute (or on the command line) from the executable directory.

With `--autoconf`, the specified configuration project (`.cgpr`) file is also deleted.

13.5 Exit Codes

0
: Success.

1
: General error (invalid option, missing file, etc.).

5
: Project parsing error.

CHAPTER

FOURTEEN

GPRINSTALL REFERENCE

GPRinstall copies the build results of a project tree - libraries, ALI files, sources, executables, and generated project files - into a target prefix directory. It records every installed file in a manifest so that `--uninstall` can later remove them precisely.

14.1 Command Line

14.1.1 Syntax

```
gprinstall [-P<proj>.gpr] [switches]
```

14.1.2 Project file and common switches

See *Common Command-Line Options* for project file discovery rules, project and configuration switches, and common diagnostic switches.

14.2 Operating modes

GPRinstall operates in one of three modes, controlled by the presence (or absence) of `--uninstall` and `--list`:

Install (default)
: Build results from the project tree are copied to the prefix. A manifest recording every installed file is written to *<prefix>/project-subdir/*`manifests/`*install-name*.

Uninstall (`--uninstall`)
: Files listed in the named manifest are removed. If any file has been modified since installation its checksum will differ and GPRinstall will refuse to delete it unless `-f` is given.

List (`--list`)
: Scan the manifests directory and print the name of every installed package. With `--stat`, also show the file count, total size, and number of missing files for each package.

14.3 Switches

14.3.1 Installation paths

`--prefix=`*dir*
: Root directory for the installation. Defaults to the prefix of the active toolchain.

`--exec-subdir=`*dir*
: Subdirectory under the prefix for executables. Default: `bin/`.

--lib-subdir=*dir*
Subdirectory for libraries. Default: `lib/`. When `--ali-subdir` is not given separately, it defaults to the same value as `--lib-subdir`.

--ali-subdir=*dir*
Subdirectory for Ada ALI files. Default: `lib/`.

--link-lib-subdir=*dir*
Subdirectory for shared library compatibility symlinks on Unix. Default: `lib/`.

--sources-subdir=*dir*
Subdirectory for installed source files. Default: `include/`.

--project-subdir=*dir*
Subdirectory for the generated GPR project file and manifests. Default: `share/gpr/`.

--cross-install
Adjust the prefix for cross-compilation: results are placed under *<prefix>/target/runtime/*.

14.3.2 Install behavior

-r / --recursive
Install all projects imported by the main project, not only the main project itself.

-f / --force
Overwrite files that already exist at the destination. During uninstall, delete files even if their checksum has changed.

-p / --create-missing-dirs
Create destination directories if they do not exist.

-m
Minimal source copy: install only the spec files and bodies strictly required to use the library (relevant for stand-alone libraries). Without `-m` all sources are installed in `dev` mode.

-d / --dry-run
Print the operations that would be performed (`cp`, `ln -s`, etc.) without executing them.

--mode=*value*
Installation mode. Accepted values:

dev (default)
Full developer installation: sources, ALI files, and libraries.

usage
End-user installation: shared libraries and executables only; no sources or static libraries.

--install-name=*name*
Name used for the manifest file and the generated project file. Defaults to the project file name without extension.

--build-name=*name*
Tag this installation with a build name (e.g. `debug`, `production`). Results are placed in a subdirectory named *install-name.build-name* under each subdirectory. Default: `default`.

--build-var=*name*
Name of an external variable whose value selects the build at load time. May be repeated. The generated project file contains a case statement keyed on this variable.

--no-build-var
Do not generate a build selection variable in the generated project file.

14.3.3 Source and artifact handling

--sources-only
Copy only source files; skip libraries, ALI files, and executables.

--no-project
Do not generate or install a GPR project file.

--minimal-project
Generate a project file that records only the minimum required metadata.

--side-debug
For executables and shared libraries, extract debug information into a separate `.debug` file (using `objcopy` and `strip`). The installed binary is stripped; the `.debug` file is placed alongside it and linked via a GNU debug link.

--no-lib-link
Do not create symlinks for shared libraries in the executable/lib directory.

14.3.4 Manifest

--no-manifest
Do not generate a manifest file. Without a manifest, `--uninstall` is not available for the installation.

14.3.5 Uninstall mode

--uninstall
Remove a previously installed package. The install name to remove may be given as a trailing argument or inferred from the project file name.

14.3.6 List mode

--list
List all packages recorded in the manifests directory.

--stat
Used with `--list`: display the file count, total installed size, and number of missing files for each package.

14.4 Installation layout

The default layout under the prefix (e.g. `/usr/local`) is:

```
<prefix>/
├── bin/                          # executables
├── lib/                          # libraries and ALI files
├── include/                      # sources
└── share/gpr/
    ├── <install-name>.gpr        # generated project file
    └── manifests/
        └── <install-name>        # manifest (MD5 hash per file)
```

When `--build-name` is used, results are placed in a build-specific subdirectory: *`install-name.build-name`* under each subdirectory.

14.5 Project package

The `Install` package sets installation defaults for a project. Command-line switches take precedence over attribute values. See *Package Install Attributes* in the Attributes Reference for the full attribute list and descriptions.

```
library project My_Lib is
   package Install is
      for Active      use "true";
      for Mode        use "dev";
      for Artifacts ("share/doc") use ("README.md");
   end Install;
end My_Lib;
```

14.6 Exit Codes

0
: Success.

1
: General error (missing file, uninstall checksum mismatch, etc.).

5
: Project parsing error.

CHAPTER

FIFTEEN

GPRNAME REFERENCE

GPRname scans source directories for Ada and non-Ada source files, identifies the Ada units they contain, and generates a GPR project file with a `Naming` package that maps every discovered unit to its source file. It is the standard tool for projects where source file names do not follow the default GNAT naming conventions.

15.1 Command Line

15.1.1 Syntax

```
gprname -P<proj>.gpr [switches] [patterns]
```

GPRname requires a project file name (`-P`). It does not use the common project-loading infrastructure - it creates or updates the named file rather than loading it. The common switches documented in *Common Command-Line Options* do not apply, with the exception of `--target` and `--RTS`.

15.1.2 Patterns

One or more filename patterns may be given on the command line. Each pattern specifies the set of files that GPRname should scan for Ada units. Patterns use glob syntax (`*`, `?`, `[...]`). If no pattern is given, GPRname scans for nothing and produces an empty naming project.

Multiple independent sets of patterns with different source directories can be defined using `--and` to separate them into sections (see below).

15.2 Switches

15.2.1 Project

`-P` *proj*
: Create or update *proj* as the main project file. Required.

`--no-backup`
: Do not create a numbered backup (`.saved_0`, `.saved_1`, ...) of the existing project file before overwriting it.

15.2.2 Source directories

`-d` *dir*
: Add *dir* as a source directory for the current section. May be repeated. Append `/**` to search *dir* recursively.

`-D` *file*
: Read source directories from *file* (one directory per line).

--minimal-dirs
After scanning, retain in `Source_Dirs` only the directories that contain at least one source file. Directories specified with `/**` are expanded to an explicit list of matching subdirectories.

15.2.3 Naming patterns

-f *pattern*
Add *pattern* as a C-language filename pattern for the current section.

-f:*lang pattern*
Add *pattern* as a filename pattern for language *lang*.

-x *pattern*
Exclude files matching *pattern* from the Ada patterns of the current section.

-xf *pattern*
Exclude files matching *pattern* from the C patterns of the current section.

-xf:*lang pattern*
Exclude files matching *pattern* from the *lang* patterns of the current section.

--and
Begin a new section. Each section has its own set of source directories and patterns. Use this when different subtrees of a project have different naming conventions.

15.2.4 Preprocessing

-gnateD*sym*=*val*
Define preprocessor symbol *sym* to *val* when parsing Ada sources.

-gnatep=*file*
Use *file* as a preprocessor data file when parsing Ada sources.

15.2.5 Ada source parser

--libadalang-parser
Use Libadalang to parse Ada source files (default).

--gcc-parser
Use the GCC Ada compiler to parse Ada source files.

15.2.6 Other

--target=*name*
Target for the Ada compiler used to parse sources.

--RTS=*dir*
Runtime for the Ada compiler used to parse sources.

--ignore-duplicate-files
Silently skip a file if its basename has already been seen. Without this switch, a duplicate basename produces a warning.

--ignore-predefined-units
Do not record predefined Ada units (`Ada`, `System`, `Interfaces`, etc.) in the naming project.

-v
Verbose output. Repeat for more detail.

15.3 Generated files

GPRname creates or updates three files named after the project (e.g. for `-P my_proj.gpr`):

my_proj.gpr
: The main project file. GPRname inserts or updates:

  - `with "my_proj_naming.gpr";` - import of the naming project.
  - `for Languages use (...);` - languages discovered.
  - `for Source_Dirs use (...);` - source directories scanned.
  - `for Source_List_File use "my_proj_source_list.txt";`
  - `package Naming renames My_Proj_Naming.Naming;`

my_proj_naming.gpr
: A separate project containing only the `Naming` package, with one `for Spec` / `for Body` entry per discovered Ada unit, and `for Implementation_Exceptions` entries for non-Ada languages.

my_proj_source_list.txt
: A plain-text list of all discovered source files, consumed by `Source_List_File` in the main project.

A numbered backup of any pre-existing project file is created before the file is overwritten (unless `--no-backup` is given).

15.4 Exit Codes

0
: Success.

1
: Error (invalid option, source file not found, etc.).

CHAPTER

SIXTEEN

GPRLS REFERENCE

GPRls lists the sources, units, objects, and dependencies of a GPR project tree. It reads the build database produced by GPRbuild to report the up-to-date status of each artifact.

16.1 Command Line

16.1.1 Syntax

```
gprls [-P<proj>.gpr] [switches] [files]
```

16.1.2 Project file and common switches

See *Common Command-Line Options* for project file discovery rules, project and configuration switches, and common diagnostic switches.

16.1.3 Files

Zero or more source file names may be given on the command line to restrict output to those files only. Simple names (without directory) are matched against source basenames; full paths are also accepted. If no files are given, GPRls lists all sources of the project (or of the entire project tree with `-U`).

16.2 Switches

16.2.1 Output selection

When none of `-u`, `-s`, or `-o` is given, GPRls prints all three categories for each compilation unit.

`-u`
: Print the unit name for each Ada compilation unit.

`-s`
: Print the source file for each compilation unit.

`-o`
: Print the object file for each compilation unit.

`-d`
: For each source file, also list the source files it depends on, together with their status. Superseded by `--closure`.

`--closure`
: Compute the transitive compilation closure of the named files (or of the project's main units if no files are given) and list every source file required to compile them.

-U
: List sources from the entire project tree, not only the root project.

-a
: Include predefined (runtime) units in the output and in dependency lists.

-a0
: Same as `-a`, but print only the simple file name for runtime sources, hiding the runtime directory path.

--hide-status
: Suppress the `OK`/`DIF` status indicator from the output.

--source-parser
: Derive dependency information by parsing Ada source files directly rather than reading ALI files. Useful when ALI files are absent or stale.

-files=*file*
: Read the list of source files to process from *file* (one name per line, blank lines ignored). The resulting list is merged with any files given on the command line.

16.3 Status indicators

Unless `--hide-status` is given, each artifact is prefixed with a status tag:

OK
: The artifact is up to date: its timestamp matches the build database.

DIF
: The artifact has been modified since the last build.

16.4 Output layout

Entries are grouped by compilation action. Within each group, the default indentation is:

- **Objects** - no indentation.
- **Unit names and sources** - indented by 3 spaces.
- **Dependencies** (with `-d`) - indented by 3 spaces.

Example output (all three categories, with status):

```
OK  obj/pack.o
   pack
   OK  src/pack.ads
   OK  src/pack.adb
```

16.5 Exit Codes

0
: Success.

1
: General error (invalid option, project error, etc.).

CHAPTER

SEVENTEEN

GPRINSPECT REFERENCE

GPRinspect loads a GPR project tree and displays its structure and contents: project relationships, source directories, attributes, packages, variables, and type definitions. It is primarily a diagnostic tool for understanding how a project tree is interpreted by the GPR project model.

17.1 Command Line

17.1.1 Syntax

```
gprinspect [-P<proj>.gpr] [switches]
```

17.1.2 Project file and common switches

See *Common Command-Line Options* for project file discovery rules, project and configuration switches, and common diagnostic switches.

17.2 Switches

17.2.1 Output format

`--display=`*`format`*
: Select the output format. Accepted values:

  `textual` (default)
  : Human-readable indented text.

  `json`
  : Pretty-printed JSON.

  `json-compact`
  : Compact JSON with no extra whitespace. Suitable for machine processing.

17.2.2 Project scope

`-r` / `--recursive`
: Display all projects in the tree, not only the root project.

`--views=`*`name`*`[,`*`name`*`...]`
: Restrict display to the named project views (comma-separated). Implies `-r`. Only available with `--display=textual`.

17.2.3 Content selection

With no content switch, GPRinspect shows only the structural information for each project (directories, relationships, library properties).

`--all`
Display everything: attributes, packages, variables, and type definitions.

`--attributes`
Display project-level and package-level attributes and their values.

`-c` / `--from-config`
Include attributes inherited from the active configuration project in the attribute display. Requires `--attributes` or `--all`.

`--packages`
Display the packages declared in each project and their attributes.

`--variables`
Display variables and type definitions declared in each project.

17.2.4 Registry

`--gpr-registry-file=`*file*
Load additional attribute definitions from *file* before loading the project tree. Use this to recognize attributes defined by external tools (such as GNATcheck or GNATprove) when inspecting projects that use them.

17.3 Output structure

Textual output is divided into sections:

Header
Timestamp and GPR2 library version.

Project tree
Summary of the loaded tree: project count, search paths.

Per-project blocks
One block per project in scope. Each block reports:

- Project name, kind, and file path.
- Object, source, library, and ALI directories.
- `extends` and `with` relationships.
- Attributes (with `--attributes` or `--all`), grouped by package.
- Variables and type definitions (with `--variables` or `--all`).

Messages
Any errors, warnings, or hints produced during loading.

JSON output wraps the same information under `"projects"` and `"messages"` keys.

17.4 Exit Codes

`0`
Success.

1

General error (invalid option, project loading failure, etc.).

5

Project parsing error.

APPENDIX

A

ENVIRONMENT VARIABLES

The following environment variables are read by GPR tools. Variables that affect project file search paths are consulted by all project-based tools (GPRbuild, GPRclean, GPRinstall, GPRls, GPRinspect); tool-specific variables are noted.

A.1 Project search path

GPR_PROJECT_PATH_FILE

Path to a text file containing project search directories, one per line. Directories listed in this file are added to the project search path before *GPR_PROJECT_PATH* and *ADA_PROJECT_PATH*.

GPR_PROJECT_PATH

Colon-separated (Unix) or semicolon-separated (Windows) list of directories to search for project files referenced by `with` clauses. Takes precedence over *ADA_PROJECT_PATH*.

ADA_PROJECT_PATH

Legacy equivalent of *GPR_PROJECT_PATH*, accepted for backward compatibility. Consulted after *GPR_PROJECT_PATH*.

The search order when resolving a project name is: the directory of the importing project, then *GPR_PROJECT_PATH_FILE* directories, then *GPR_PROJECT_PATH*, then *ADA_PROJECT_PATH*, then the default project path of the active toolchain.

A.2 Configuration

GPR_CONFIG

Path to a configuration project (`.cgpr`) file or to a directory containing one. When set to a directory, the tool looks for a file named `<target>.cgpr` (or `default.cgpr` for the native target) inside it. Takes precedence over the default auto-configuration mechanism but is overridden by an explicit `--config` switch on the command line.

GPR_RUNTIME_PATH

Colon-separated list of directories searched when resolving a relative Ada runtime path specified in the configuration project. Used by GPRbuild, GPRclean, GPRinstall, and GPRconfig.

A.3 Build engine

GNAT_GPR_ENGINE

Selects the GPRbuild engine. Accepted values: `1` or `legacy` for the original engine; `2` or `new` for GPRbuild2 (the DAG-based engine). The `--gpr=`*n* command-line switch takes precedence over this variable. See *GPRbuild Reference* for details.

A.4 Compiler discovery

PATH

The standard executable search path. GPRconfig searches *PATH* for compiler executables during auto-configuration.

MAKEFLAGS

Parsed by GPRbuild2 to detect a GNU Make jobserver token pipe when GPRbuild is invoked as a sub-make. If jobserver authentication details are present, GPRbuild2 coordinates its parallel job limit with the enclosing make process automatically. Users do not normally set this variable directly; it is set by make.

APPENDIX

B

GLOSSARY

Action
: An atomic build step in GPRbuild2's DAG: a single invocation of an external tool (compiler, binder, linker, archiver) with a defined set of inputs and outputs. See *GPRbuild Reference*.

Aggregate library project
: A project kind that builds a single library by collecting object files from a set of constituent projects. Unlike plain aggregate projects, aggregate library projects can be imported by other projects. See *Aggregate Library Project*.

Aggregate project
: A project kind whose sole purpose is to group other project files for a single-invocation build. An aggregate project does not contain sources itself. See *Aggregate Project*.

ALI file
: An Ada Library Information file (`.ali`), generated by the GNAT compiler alongside each compiled unit. It records unit dependencies, compilation switches, and cross-reference data. GPRbuild uses ALI files for incremental build decisions; GPRls and GPRinstall use them for dependency listing and installation.

Attribute
: A named project-level or package-level setting that controls how a project or tool behaves. Attributes are declared with `for` statements inside a project file or package. See the *Attributes* chapter.

Build database
: A persistent store written by GPRbuild2 that records the signature of every action executed. On subsequent builds, GPRbuild2 compares current signatures against the stored ones to determine which actions need re-execution.

Configuration project
: A special project (qualifier `configuration`, extension `.cgpr`) that supplies toolchain-specific settings - compiler drivers, default switches, naming conventions - to all other projects in the tree. Generated by GPRconfig and loaded via `--config` or `--autoconf`. See *Configuration Project* and *GPRconfig Reference*.

Entry point
: A build target that anchors the DAG traversal in GPRbuild2: an executable declared via the `Main` attribute or on the command line, a library interface, or additional units specified via the `Roots` attribute.

Extending project
: A project that inherits sources and attributes from a base project using the `extends` clause. The extending project may shadow inherited sources and override attributes. See *Project Extension*.

External variable
: A string variable whose value is supplied from outside the project file, via the `-X` command-line switch or the `External` attribute in an aggregate project. Used with `case` statements to produce configuration-dependent attribute values.

Knowledge base (KB)
A collection of XML files describing compiler executables, target platforms, and runtime variants. The KB is embedded in all GPR tools and consulted during auto-configuration. See *Knowledge Base*.

Library project
A project kind that builds a static or shared library rather than executables. Identified by the `library` qualifier or by the presence of `Library_Name` and `Library_Dir` attributes. See *Library Project*.

Manifest
A file written by GPRinstall that records the path and MD5 checksum of every file installed. Used by `gprinstall --uninstall` to identify and remove previously installed files safely.

Naming package
A `package Naming` declaration inside a project file that maps language unit names to source file names, overriding the default naming conventions. Generated by GPRname for projects with non-standard file naming.

Package
A named group of attributes inside a project file, corresponding to a specific tool (`Compiler`, `Linker`, `Builder`, `Clean`, `Install`, `Naming`, etc.). Packages scope attribute names and may be extended or renamed.

Project file
A text file with a `.gpr` extension that describes a software component: its sources, build settings, dependencies, and tool configuration. Parsed by all GPR tools.

Project tree
The complete graph of project files loaded for a build: the root project, all projects it imports (directly or transitively), and the active configuration project.

Project view
The resolved representation of a single project file within a loaded project tree, after attribute inheritance, variable expansion, and extension redirection have been applied.

Qualifier
One or two keywords placed immediately before the `project` keyword to declare the project's kind explicitly: `abstract`, `library`, `aggregate`, `aggregate library`, `configuration`, or `standard`. See *Project Kinds*.

Root project
The project file named on the command line (or discovered automatically). It is the starting point for project tree loading and, for GPRbuild, the source of the `Main` and `Builder` attributes that drive the build.

Signature
A checksum computed over all inputs and expected outputs of an action (source files, dependency closures, switches, configuration). GPRbuild2 stores signatures in the build database; an action is re-executed only when its signature has changed.

Source directory
A directory listed in the `Source_Dirs` attribute from which the project model collects source files. See *Source Resolution*.

Stand-alone library (SAL)
A library project that bundles its own elaboration code, declared by setting `Library_Interface` or `Interfaces`. A SAL can be loaded independently of the main program's elaboration sequence. See *Library Project*.

Standard project
The default project kind: contains sources and produces executables or object files. The `standard` qualifier may be omitted. See *Standard Project*.

Unit
An Ada compilation unit: a package, subprogram, generic, or instantiation, identified by a dotted name. One or more units may reside in a single source file.

`with` clause
: A declaration at the top of a project file that imports another project, making its attributes and sources visible. The imported project is added to the project tree.

APPENDIX

C

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

INDEX

Symbols

-P
    switch, 49
-X, pair: switch; --config, pair: switch;
        --autoconf, pair: switch; --target,
        pair: switch; --RTS
    switch, 49
-q, pair: switch; -v, pair: switch; -we,
        pair: switch; -ws
    switch, 50

A

abstract project, 22
Action, **81**
ADA_PROJECT_PATH, 7, 18, 49, 79
    environment variable, 79
Aggregate library project, **81**
aggregate library project, 25
Aggregate project, **81**
aggregate project, 24
ALI file, **81**
Alternative (*built-in*), 10
artifacts cleaned
    GPRclean, 64
Attribute, 30, **81**
attribute declaration, 13
Attributes - Package Binder Attributes
    Default_Switches, 37
    Driver, 37
    Objects_Path, 37
    Prefix, 37
    Required_Switches, 37
    Switches, 37
Attributes - Package Builder Attributes
    Default_Switches, 38
    Executable, 38
    Executable_Suffix, 38
    Global_Compilation_Switches, 38
    Global_Config_File, 38
    Global_Configuration_Pragmas, 38
    Switches, 38
Attributes - Package Clean Attributes
    Artifacts_In_Exec_Dir, 38
    Artifacts_In_Object_Dir, 38
    Object_Artifact_Extensions, 38
    Source_Artifact_Extensions, 38
    Switches, 38
Attributes - Package Compiler Attributes
    Config_Body_File_Name, 40
    Config_Body_File_Name_Index, 40
    Config_Body_File_Name_Pattern, 40
    Config_File_Switches, 40
    Config_File_Unique, 40
    Config_Spec_File_Name, 40
    Config_Spec_File_Name_Index, 40
    Config_Spec_File_Name_Pattern, 40
    Default_Switches, 39
    Dependency_Driver, 40
    Dependency_Kind, 39
    Dependency_Switches, 40
    Driver, 39
    Include_Path, 41
    Include_Path_File, 41
    Include_Switches, 41
    Language_Kind, 39
    Leading_Required_Switches, 39
    Local_Config_File, 39
    Local_Configuration_Pragmas, 39
    Mapping_Body_Suffix, 41
    Mapping_File_Switches, 41
    Mapping_Spec_Suffix, 41
    Max_Command_Line_Length, 41
    Multi_Unit_Object_Separator, 39
    Multi_Unit_Switches, 39
    Object_File_Suffix, 39
    Object_File_Switches, 39
    Object_Path_Switches, 41
    Required_Switches, 39
    Response_File_Format, 41
    Response_File_Switches, 41
    Source_File_Switches, 40
    Switches, 39
    Trailing_Required_Switches, 40
Attributes - Package Gnatls Attributes

Switches, 41
Attributes - Package Install Attributes
Active, 41
Artifacts, 42
Exec_Subdir, 42
Install_Name, 42
Install_Project, 42
Lib_Subdir, 42
Mode, 42
Prefix, 41
Project_Subdir, 42
Required_Artifacts, 42
Side_Debug, 42
Sources_Subdir, 42
Attributes - Package Linker Attributes
Default_Switches, 42
Driver, 43
Group_End_Switch, 43
Group_Start_Switch, 43
Leading_Switches, 43
Linker_Options, 43
Max_Command_Line_Length, 43
Required_Switches, 42
Response_File_Format, 43
Response_File_Switches, 43
Switches, 43
Trailing_Switches, 43
Unconditional_Linking, 43
Attributes - Package Naming Attributes
Body, 44
Body_Suffix, 44
Casing, 44
Dot_Replacement, 44
Implementation, 44
Implementation_Exceptions, 44
Implementation_Suffix, 44
Separate_Suffix, 44
Spec, 44
Spec_Suffix, 44
Specification, 44
Specification_Exceptions, 44
Specification_Suffix, 45
Attributes - Project Level Attributes
Archive_Builder, 32
Archive_Builder_Append_Option, 32
Archive_Indexer, 32
Archive_Prefix, 32
Archive_Suffix, 32
Config_Prj_File, 33
Create_Missing_Dirs, 32
Default_Language, 33
Disable_Linking, 33
Excluded_Source_Files, 34
Excluded_Source_List_File, 34
Exec_Dir, 32
External, 34
Externally_Built, 35
Gpr_Registry_Dirs, 32
Ignore_Source_Sub_Dirs, 32
Inherit_Source_Path, 32
Interfaces, 34
Languages, 35
Leading_Library_Options, 35
Library_Auto_Init, 35
Library_Auto_Init_Supported, 36
Library_Builder, 37
Library_Dir, 35
Library_Encapsulated_Options, 35
Library_Encapsulated_Supported, 35
Library_Install_Name_Option, 36
Library_Interface, 35
Library_Kind, 35
Library_Major_Minor_Id_Supported, 36
Library_Name, 36
Library_Options, 36
Library_Partial_Linker, 32
Library_Reference_Symbol_File, 36
Library_Rpath_Options, 36
Library_Src_Dir, 36
Library_Standalone, 36
Library_Support, 37
Library_Symbol_File, 36
Library_Symbol_Policy, 36
Library_Version, 36
Library_Version_Switches, 36
Linker_Lib_Dir_Option, 37
Locally_Removed_Files, 34
Main, 35
Name, 35
Object_Dir, 33
Object_Generated, 33
Objects_Linked, 33
Project_Dir, 35
Project_Files, 34
Project_Path, 34
Required_Toolchain_Version, 33
Roots, 35
Run_Path_Option, 33
Run_Path_Origin, 33
Runtime, 33
Runtime_Dir, 33
Runtime_Library_Dir, 33
Runtime_Source_Dir, 33
Runtime_Source_Dirs, 34
Separate_Run_Path_Options, 34
Shared_Library_Minimum_Switches, 37
Shared_Library_Prefix, 37
Shared_Library_Suffix, 37

Source_Dirs, 33
Source_Files, 34
Source_List_File, 34
Symbolic_Link_Supported, 37
Target, 34
Toolchain_Description, 34
Toolchain_Name, 34
Toolchain_Version, 34
Warning_Message, 35
auto-initialization, 24

B

basename, 29
batch mode, pair: switch; --batch, pair: switch; --config
GPRconfig, 60
build behavior, pair: switch; -f, pair: switch; -j, pair: switch; -k
GPRbuild, 55
Build database, **81**
build engine selection, GNAT_GPR_ENGINE
GPRbuild, 53
build execution, DAG, incremental build
GPRbuild, 56
built-in function, 9

C

case construction, 15
child project, 7
command-line options, 48
comment, 6
common options, 48
concatenation, 11
configuration file format, cgpr
GPRconfig, 61
Configuration project, **81**
configuration project, 26
content selection, pair: switch; --all, pair: switch; --attributes
GPRinspect, 75
context clause, 7
customizing
knowledge base, 48

D

declaration, 8
Default (*built-in*), 10

E

embedded KB
knowledge base, 47
Entry point, **81**
environment variable, 77
ADA_PROJECT_PATH, 7, 18, 49, 79
GNAT_GPR_ENGINE, 53, 79
GPR_CONFIG, 79
GPR_PROJECT_PATH, 7, 18, 49, 79
GPR_PROJECT_PATH_FILE, 49, 79
GPR_RUNTIME_PATH, 79
MAKEFLAGS, 80
PATH, 80
expression, 11
Extending project, **81**
extends all, 28
extends keyword, 26
External (*built-in*), 9
External variable, **81**
External_As_List (*built-in*), 9

F

File_As_List (*built-in*), 9
Filter_Out (*built-in*), 10

G

generated files
GPRname, 70
GNAT_GPR_ENGINE, 53
environment variable, 79
GPR, 1
GPR_CONFIG
environment variable, 79
GPR_PROJECT_PATH, 7, 18, 49, 79
environment variable, 79
GPR_PROJECT_PATH_FILE, 49, 79
environment variable, 79
GPR_RUNTIME_PATH
environment variable, 79
GPRbuild, 51
build behavior, pair: switch; -f, pair: switch; -j, pair: switch; -k, 55
build engine selection, GNAT_GPR_ENGINE, 53
build execution, DAG, incremental build, 56
phase selection, pair: switch; -c, pair: switch; -b, pair: switch; -l, 54
GPRclean, 61
artifacts cleaned, 64
switches, pair: switch; -r, pair: switch; -n, 63
GPRconfig, 57
batch mode, pair: switch; --batch, pair: switch; --config, 60
configuration file format, cgpr, 61
interactive mode, 61

target, pair: switch; --target, 59
GPRinspect, 74
content selection, pair: switch; --all, pair: switch; --attributes, 75
output format, pair: switch; --display, 75
output structure, 76
GPRinstall, 64
installation layout, 67
installation paths, pair: switch; --prefix, 65
operating modes, pair: switch; --uninstall, pair: switch; --list, 65
GPRls, 71
output selection, pair: switch; --closure, 73
status indicators, 74
GPRname, 68
generated files, 70
naming patterns, naming convention, 70

I

identifier, 5
indexed attribute, 13
installation layout
GPRinstall, 67
installation paths, pair: switch; --prefix
GPRinstall, 65
interactive mode
GPRconfig, 61
Item_At (*built-in*), 10

K

KB structure
knowledge base, 47
knowledge base, 45
customizing, 48
embedded KB, 47
KB structure, 47
run-time selection, --db, 47
validation, 48
Knowledge base (*KB*), **82**

L

Library project, **82**
library project, 22
Library_Dir, 22
Library_Interface, 24
Library_Kind, 22
Library_Name, 22
Library_Standalone, 24
Library_Version, 23
limited with clause, 18
list value, 8
Lower (*built-in*), 10

M

MAKEFLAGS
environment variable, 80
Manifest, **82**
Match (*built-in*), 10

N

naming convention, 29
Naming package, **82**
naming patterns, naming convention
GPRname, 70

O

operating modes, pair: switch; --uninstall, pair: switch; --list
GPRinstall, 65
out-of-tree build, 50
output format, pair: switch; --display
GPRinspect, 75
output selection, pair: switch; --closure
GPRls, 73
output structure
GPRinspect, 76

P

Package, **82**
package declaration, 14
package extension, 15
package renaming, 15
PATH, 80
environment variable, 80
path resolution, 18
phase selection, pair: switch; -c, pair: switch; -b, pair: switch; -l
GPRbuild, 54
project extension, 26
Project file, **82**
project file, 1
structure, 6
Project tree, **82**
project tree, 17
Project view, **82**

Q

Qualifier, **82**

R

Remove_Prefix (*built-in*), 10
Remove_Suffix (*built-in*), 10

reserved words, 6
Root project, **82**
root project, 17
run-time selection, --db
    knowledge base, 47

S

scenario variable, 9
shared library versioning, 23
Signature, **82**
Source directory, **82**
source ownership, 7
source resolution, 28
source shadowing, 29
source shadowing in extension, 27
Split (*built-in*), 10
Stand-alone library (*SAL*), **82**
stand-alone library, 24
Standard project, **82**
standard project, 21
status indicators
    GPRls, 74
string literal, 6
string value, 8
switch
    -P, 49
    -X, pair: switch; --config, pair:
        switch; --autoconf, pair: switch;
        --target, pair: switch; --RTS, 49
    -q, pair: switch; -v, pair: switch;
        -we, pair: switch; -ws, 50
switches, pair: switch; -r, pair: switch;
        -n
    GPRclean, 63

T

target, pair: switch; --target
    GPRconfig, 59
typed string declaration, 12
typed variable, 12

U

Unit, **82**
untyped variable, 12
Upper (*built-in*), 10

V

validation
    knowledge base, 48
variable, 12
variable reference, 12

W

with clause, 7, 17, **83**