# GPRbuild User's Guide

Document revision level <<TBD>>
Date: <<TBD>>

AdaCore

Printed November 2007

# Introduction

GPRbuild is a generic build tool designed for the construction of large multi-language systems organized into subsystems and libraries. It is well-suited for compiled languages supporting separate compilation, such as Ada, C, C++ or Fortran.

GPRbuild manages a three step build process

- compilation phase:

  Each compilation unit of each subsystem is examined in turn, checked for consistency, and compiled or recompiled when necessary by the appropriate compiler. The recompilation decision is based on dependency information that is typically produced by a previous compilation.
- post-compilation phase (or binding):

  Compiled units from a given language are passed to a language-specific post-compilation tool if any. Also during this phase objects are grouped into static or dynamic libraries as specified.
- linking phase:

  All units or libraries from all subsystems are passed to a linker tool specific to the set of toolchains being used.

GPRbuild takes as its main input a project file '<file>.gpr' defining the build characteristics of the system under construction such as:

- which sources to use,
- where to find them,
- where to store the objects produced by the compiler,
- which options should be passed to the compiler.

The tool is generic in that it provides, when possible, equivalent build capabilities for all supported languages. For this, it uses a configuration file '<file>.cgpr' that has a syntax and structure very similar to a project file, but which defines the characteristics of the supported languages and toolchains. The configuration file contains information such as:

- the default source naming conventions for each language,
- the compiler name, location and required options,
- how to compute inter-unit dependency,
- how to build static or dynamic libraries,
- which post-compilation actions are needed,
- how to link together units from different languages.

On the other hand, GPRbuild is not a replacement for general-purpose build tools such as make or ant which give the user a high level of control over the

build process itself. When building a system requires complex actions that do not fit well in the three-phase process described above, `GPRbuild` might not be sufficient. In such situations, `GPRbuild` can still be used to manage the appropriate part of the build. For instance it can be called from within a Makefile.

# 1 Guided Tour

This chapter presents examples ranging from simple multi-language builds to some of the most advanced scenarios. All the examples shown in the text are included in the GPRbuild package, which is installed on your system in '`<prefix>/share/examples/gprbuild/`'.

## 1.1 Configuration

GPRbuild requires one configuration file describing the languages and toolchains to be used, and project files describing the characteristics of the user project. Typically the configuration file can be created automatically by `GPRbuild` based on the languages defined in your projects and the compilers on your path. In more involved situations – such as cross compilation, or environments with several compilers for the same language – you may need to control more precisely the generation of the desired configuration of toolsets. A tool, GPRconfig, described in Chapter 5 [Configuring with GPRconfig], page 29), offers this capability. In this chapter most of the examples can use autoconfiguration.

GPRbuild will start its build process by trying to locate a configuration file called '`default.cgpr`'. If such a file is not available or if the option `--autoconf=xxx` is used, GPRbuild will create its own configuration file suitable for native development and assuming that there are known compilers on your path for each of the necessary languages. It is preferable and often necessary to manually generate your own configuration file when:

- using cross compilers (in which case you need to use gprconfig's '`--target=`') option,
- using a specific Ada runtime (e.g. '`--RTS=sjlj`'),
- working with compilers not in the path or not first in the path, or
- autoconfiguration does not give the expected results.

GPRconfig provides several ways of generating configuration files. By default, a simple interactive mode lists all the known compilers for all known languages. You can then select a compiler for each of the languages; once a compiler has been selected, only compatible compilers for other languages are proposed. Here are a few examples of GPRconfig invocation:

- The following command triggers interactive mode. The configuration will be generated in GPRbuild's default location, `<gprbuild_install_root>/share/gpr/default.cgpr`)

```
gprconfig
```

- The first command below also triggers interactive mode, but the resulting configuration file has the name and path selected by the user. The second

command shows how GPRbuild can make use of this specific configuration file instead of the default one.

```
gprconfig -o path/my_config.cgpr
gprbuild --config=path/my_config.cgpr
```

- The following command again triggers interactive mode, and only the relevant cross compilers for target ppc-elf will be proposed.

```
gprconfig --target=ppc-elf
```

- The next command triggers batch mode and generates at the default location a configuration file using the first native Ada and C compilers on the path.

```
gprconfig --config=Ada --config=C --batch
```

- The next command, a combination of the previous examples, creates in batch mode a configuration file named 'x.cgpr' for cross-compiling Ada with a run-time called hi and using C for the LEON processor.

```
gprconfig --target=leon-elf --config=Ada,,hi --config=C --batch -o x.cgpr
```

## 1.2 First Steps

Assume a simple case of interfacing between Ada and C where the main program is first in Ada and then in C. For a main program in Ada the following project may be used:

```
project Ada_Main is
   for Languages    use ("Ada", "C");
   for Source_Dirs  use ("ada_src", "util_src");
   for Object_Dir   use  "obj";
   for Exec_Dir     use ".";
   for Source_Files use ("ada_main.adb", "c_lib.ads", "lib.h", "lib.c");
   for Main         use ("ada_main.adb");
end Ada_Main;
```

This project indicates that:

- the sources files of this subsystem are written in Ada and C.
- the directories containing the sources are ada_src and util_src.
- the directory for the objects is obj.
- the directory for the executables is "."; i.e., the directory containing the project file itself.
- the complete list of source files is ada_main.adb, c_lib.ads, lib.h, lib.c. They can be found anywhere in the source directories mentioned above.
- the main entry point of the system is in the source file ada_main.adb.

This information is sufficient for GPRbuild to build an executable program from the sources. Note that no direct indication on how to build the various elements is given in the project file, which describes the project properties rather than a set of actions to be executed. Here is the invocation of GPRbuild that allows building the multi-language program:

```
$ gprbuild -Pada_main
gcc -c ada_main.adb -o ada_main.o
gcc -c c_lib.ads -o c_lib.o
gcc -c lib.c -o lib.o
gprbind ada_main
...
gcc ada_main.o -o ada_main.exe
```

Notice the three steps described in the [Introduction], page 1:

- The first three gcc commands correspond to the compilation phase.
- The gprbind command corresponds to the post-compilation phase.
- The last gcc command corresponds to the final link.

The default output of GPRbuild's execution is kept reasonably simple and easy to understand. In particular, some of the less frequently used commands are not shown, and some parameters are abbreviated. GPRbuild's option `-v` provides a much more verbose output which includes, among other information, more complete compilation, post-compilation and link commands.

To illustrate some other GPRbuild capabilities, here is a slightly different project using similar sources and a main program in C:

```
project C_Main is
   for Languages    use ("Ada", "C");
   for Source_Dirs  use ("c_src", "util_src");
   for Object_Dir   use  "obj1";
   for Main         use ("c_main.c");
   package Compiler is
      C_Switches := ("-pedantic");
      for Default_Switches ("C")   use C_Switches;
      for Default_Switches ("Ada") use ("-gnaty");
      for Switches ("c_main.c")    use C_Switches & ("-g");
   end Compiler;
end C_Main;
```

This project has many similarities with the previous one, as evident in the `Languages`, `Source_Dirs` and `Object_Dirs` attributes. As expected, its `Main` attribute now refers to a C source. The first noticeable difference is the lack of a `Source_Files` attribute. When not specified explicitly, this attribute has an implicit value which is the complete set of sources of the given `Languages` to be found in `Source_Dirs`. Many attributes can be left implicit and will be given reasonable default values. For instance, `Source_Dirs` and `Object_Dir` default to the current directory (where the project file resides). The `Exec_Dir` attribute defaults to the value of `Object_Dir`.

The other notable difference in this new project is the presence of the package `Compiler`, which groups the attributes specific to the compilation phases. The `Default_Switches` attribute provides the list of compilation switches to be used for any source of a given `Languages` value unless the source has its own set of

compilation switches specified by the attribute `Switches`. Note also the use of a the variable `C_Switches`. A project variable can be useful to avoid duplication of information and here ensures that the file '`c_main.c`' is always compiled with the default switches (whatever they may be), plus '`-g`'. In this specific situation the use of a variable could have been replaced by a reference to the `Default_Switches` attribute:

```
for Switches ("c_main.c") use Compiler'Default_Switches ("C") & ("-g");
```

Here is the output of the GPRbuild command using this project:

```
$gprbuild -Pc_main
gcc -c -pedantic -g c_main.c -o c_main.o
gcc -c -gnaty ada_lib.adb -o ada_lib.o
gcc -c -gnaty c_lib.ads -o c_lib.o
gcc -c -pedantic lib.c -o lib.o
gprbind c_main
...
gcc c_main.o -o c_main.exe
```

The switches for compiling Ada sources, the default switches for C sources in the compilation of '`lib.c`', and the specific switches for '`c_main.c`' have all been taken into account. When comparing this GPRbuild output with the previous one, notice that there are common sources between the two projects, namely '`c_lib.ads`' and '`lib.c`'. Those sources were compiled twice, once for each project. This is not surprising, since the two projects are independent and have different `Object_Dirs` even though they share some sources. It is possible to share more than the sources, and thus avoid unnecessary recompilations when compilation units are used in several different projects, by splitting a system into subsystems.

## 1.3 Subsystems

The common compilation units may be grouped in a separate subsystem with its own project file and a specified `Object_Dir` attribute:

```
project Common_Subsystem is
   for Source_Dirs  use ("util_src");
   for Object_Dir   use  "obj_util";
end Common_Subsystem;
```

By default, all the sources in directory `util_src` belong to project `Common_Subsystem`, and when compiled, their objects go in `obj_dir`. Other projects, whose sources depend on sources in `util_src`, can create a dependency relationship using a project `with` clause:

```
with "Common_Subsystem";
project Ada_Main is
   for Languages    use ("Ada");
   for Source_Dirs  use ("ada_src");
   ...
```

```
end Ada_Main;
```

The `Ada_Main.Source_Dirs` no longer includes `util_src`. The main program in `ada_src` still needs sources from `util_src` to be compiled. It also needs those units built. Now they will be compiled using the build properties defined in `Common_Subsystem` instead of those of `Ada_Main`.

The project's `with` clause has several effects. It provides visibility of sources during the compilation process as if the units from `Ada_Main` were compiled with an implicit `-Iutil_src` option. It also guarantees that the necessary units from `Common_Subsystem` are available for `Ada_Main`'s linking phase.

It is also possible to use the `with` relationship to define a project file that allows sharing common attributes or packages. For instance, you can define a project whose only purpose is to centralize the default compilation options for various languages:

```
abstract project Attribute_Sharing is
   for Source_Files use ();
   package Compiler is
      for Default_Switches ("C")   use ("-pedantic");
      for Default_Switches ("Ada") use ("-gnaty");
   end Compiler;
end Attribute_Sharing;
```

This package does not correspond to a subsystem with proper source files, and this is conveyed by the modifier `abstract` just before the keyword `project`. An abstract project cannot have associated sources, hence the definition of `Source_Files` as the empty list. It informs the build manager that no compilation is to take place directly from this project. Defining an empty list of sources is essential for such utility projects because otherwise the default rules would apply and all sources in the current directory would be associated with this project. Now other projects can share this project's attributes and packages. For instance, `Ada_Main` can share all the `Compiler` package's attributes at once using a renaming declaration:

```
with "Attribute_Sharing";
...
project Ada_Main is
...
   package Compiler renames Attribute_Sharing.Compiler;
end Ada_Main;
```

or if other aspects of the `Compiler` package need to be defined locally, such as adding `-g` for the 'c_main.c' source only, sharing can be done at the attribute level directly as in this new version of `C_Main.Compiler`:

```
   package Compiler is
      for Default_Switches ("C") use
        Attribute_Sharing.Compiler'Default_Switches ("C");
      for Switches ("c_main.c")  use
        Compiler'Default_Switches ("C") & ("-g");
```

```
   end Compiler;
```

In order to change the compilation options of all C files used for building both `C_Main` and `Ada_Main`, it is sufficient to edit a single project file.

## 1.4 Project Extensions

Previous sections have shown how to share both compilation units and project properties among different systems. Sharing is an essential aspect of large system development since it avoids unnecessary duplication, reduces compilation time and lessens the maintenance burden. Unfortunately, it sometimes gets in the way. In the above example, `Ada_Main` and `C_Main` both depend on `Common_Subsystems` because they have precisely the same needs, but the requirements of the two systems can diverge over time in which case sharing becomes a problem rather than a solution. For instance, suppose that `C_Main` is now baselined and should not be changed, while `Ada_Main` is still under active development (which requires changes in the sources of the common subsystem). The issue is how to satisfy `Ada_Main`'s needs without taking any risk of perturbing `C_Main`. One possibility is to duplicate the common subsystem completely so that one version can evolve while the other one can remain frozen. That will force duplication and recompilation of the whole project. Extensions provide a more flexible mechanism, making it possible to provide a new version of a subsystem without having to duplicate the sources that do not change. A project extension inherits all the sources and objects from the project it extends and allows you to redefine some of the sources which hide the original versions. You can also add new sources or remove existing ones. Here is an example of extending project `Common_Subsystem`:

```
project New_Common_Subsystem
        extends "../subsystems/common_subsystem.gpr" is
   for Source_Dirs  use ("new_util_src");
   for Object_Dir   use  "new_obj_util";
end New_Common_Subsystem;
```

The project sources are to be found in `Common_Subsystem.Source_Dirs` and `New_Common_Subsystem.Source_Dirs`, the latter hiding the former when the same source filenames are found in both. New sources can be added to the project extension by simply placing them in its `Source_Dirs`. Original sources can be removed, because of the attribute `Excluded_Source_Files`. When building a project extension, all units depending on new versions of sources are rebuilt in the extension's `Object_Dir`. Other objects remain in their original location and will be used by the builder whenever necessary.

A project extension can be used in the context of a large program composed of many subsystems. Extending one subsystem may implicitly affect many other subsystems depending on it, even though there are no source changes in those dependent projects. In order to avoid the need to generate a project extension

for each of the affected subsystems, you can use the notion of “extending all”, which basically means “extend all the projects necessary so that this specific project extension can be used instead of its original version”. Here is an example of an extending-all project:

```
with "new_common_subsystem.gpr";
project New_Ada_Main extends all "../subsystems/ada_main.gpr" is
   for Source_Dirs  use ("new_ada_src");
   for Object_Dir   use  "new_obj";
   for Exec_Dir     use ".";
end New_Ada_Main;
```

## 1.5 Libraries

A library is a subsystem packaged in a specific way. There are two major kinds of libraries: *static* and *dynamic*. A project file representing a library is very similar to a project file representing a subsystem. You just need to give the library a name, through the attribute `Library_Name`. It is also possible to define a `Library_Dir` which allows you to separate the final library components (e.g. archive files) from the compilation byproducts (e.g. object files) that may be needed to efficiently rebuild a new version of the library but which are not of any interest to library users. Here is a simple project file for a static library:

```
library project Static_Lib is
   for Languages    use ("Ada", "C");
   for Source_Dirs  use ("lib_src");
   for Object_Dir   use "obj";
   for Library_Dir  use "lib";
   for Library_Kind use "static";
   for Library_Name use "l1";
end Static_Lib;
```

The library can be built on its own using standard GPRbuild commands, for example:

```
$gprbuild static_lib.gpr
```

It can also be built as a by product of building a main project using this library through a "with" as it was the case for simple subsystems:

```
with "static_lib.gpr";
project Main is
   for Main  use ("ada_main.adb");
end Main;
```

in which case

```
$gprbuild main.gpr
```

will rebuild the library `l1` if it is not up to date.

To construct a dynamic library instead of a static library, simply replace

```
   for Library_Kind use "static";
```

by

```
   for Library_Kind use "dynamic";
```

Library projects can also be useful to describe a library that you may want to use but that, for some reason, cannot be rebuilt, for instance when the sources to rebuild the library are not available. Such library projects need simply to use the `externally_built` attribute as in the example below:

```
library project Extern_Lib is
   for Languages     use ("Ada", "C");
   for Source_Dirs   use ("lib_src");
   for Library_Dir   use "lib2";
   for Library_Kind  use "dynamic";
   for Library_Name  use "l2";
   for Externally_Built use "true";
end Extern_Lib;
```

In the case of externally built libraries, the `Object_Dir` attribute does not need to be specified in this case because it will never be used.

## 1.6 Scenarios and conditional source files

In the previous section we have seen how to create libraries from simple subsystems by using specific attributes. Rather than having several almost identical project files defining a static or a dynamic version of the library representing a subsystem, one can write a more generic version of the project file thanks to the notion of scenarios. Scenarios are defined as types with a known set of string values as shown by the type `Lib_Kind` below. A scenario variable,`Kind` below can get its initial value from the environment thanks to an `external` call. This variable can then be used in a case statement as shown below. Note that only scenario variables can be used to control case statements. There is another kind of untyped string variables illustrated by `Prefix` below that can be used in general string expressions but cannot control case statements and thus are not scenarios variables.

```
library project General_Lib is
   type Lib_Kind is ("static", "dynamic", "extern");
   Kind : Lib_Kind := external ("LIB", "static");
   Prefix := "../libraries/";
   for Languages    use ("Ada", "C");
   for Source_Dirs  use (Prefix & "lib_src");
   case Kind is
      when "static" =>
         for Object_Dir   use  Prefix & "obj";
         for Library_Dir  use  Prefix & "lib";
         for Library_Kind use  "static";
         for Library_Name use "l1";
      when "dynamic" =>
         for Object_Dir   use  Prefix & "obj2";
         for Library_Dir  use  Prefix & "lib2";
         for Library_Kind use  "dynamic";
```

```
            for Library_Name use "l2";
         when "extern" =>
            for Library_Dir  use  Prefix & "lib2";
            for Library_Kind use "dynamic";
            for Library_Name use "l2";
            for Externally_Built use "true";
      end case;
   end General_Lib;
```

# 2 Important Concepts

The following concepts are the foundation of GNAT Project files and the GPRbuild process.

- Source files and source directories

  A source file is associated with a language through a naming convention. For instance, `foo.c` is typically the name of a C source file; `bar.ads` or `bar.1.ada` are two common naming conventions for a file containing an Ada spec. A compilation unit is often composed of a main source file and potentially several auxiliary ones, such as header files in C. You can define or modify the naming conventions, which are used by `GPRbuild` to invoke the appropriate compiler. Source files are looked up in the source directories associated with the project through the `Source_Dirs` attribute. By default, all the files (in these source directories) following the naming conventions associated with the declared languages are considered to be part of the project. It is also possible to limit the list of source files using the `Source_Files` or `Source_List_File` attributes.

- Object files and object directory

  An object file is an intermediate file produced by the compiler from a compilation unit. It is used by the post-compilation phases to produce final executables or libraries. The object files produced in the context of a given project are stored in a single directory, the `Object_Dir` that can be specified in the project file. A need for storing objects in different directories corresponds to the need to split the system into distinct subsystems.

- Project file

  A text file using an Ada-like syntax. It defines build-related characteristics of an application, or a part of it. The characteristics include the list of sources, the location of those sources, the location for the generated object files, the name of the main program, and the default options for the various tools involved in the build process. These characteristics can be conditionalized through scenario variables and case statements.

- Configuration file

  A text file using the project file syntax. It defines languages and their characteristics as well as toolchains for those languages and their characteristics.

- Project attribute

  A specific characteristic as defined by a project attribute clause. Its value is a string or a sequence of strings. For instance, the `Source_Dirs` attribute is a sequence of strings representing the actual names of the directories containing the sources of the project. The attribute itself can be param-

eterized. For instance `Compiler'Default_Options ("Ada")` refers to the default compiler options for the Ada language.

- Project subpackage

  Global characteristics are defined at the top level of a project. Specific characteristics affecting a given tool are grouped in a subpackage with the same name as the tool. The most common subpackages are `Builder`, `Compiler`, `Binder`, and `Linker`.

- Project variables

  Two kinds of variables are available with projects: simple variables and scenario variables. A simple variable can hold any string, or sequence of string values; this is useful as a shortcut for a complex expression. Scenario variables are a restricted version of simple variables; they can only take their values from a static enumeration of strings representing the various supported scenarios. These scenario variables are used in conjunction with the `case` construct to offer limited conditionalization of project characteristics. Both types of variables are local to the project defining them and can get initialized with an external value, such as an environment variable, through the `external` directive.

- Subsystem

  A subsystem is a coherent part of the complete system to be built. It is represented by a set of sources and one object directory. A simple system is composed of a single subsystem. Complex systems are usually composed of a graph of interdependent subsystems. A subsystem is dependent on another subsystem if knowledge of the other one is required to build it, and in particular if visibility on some of the sources of this other subsystem is required. A subsystem is usually represented by one project file. The subsystem dependency relationship is represented by the `with` clause between corresponding project files.

- Library

  A library is a specific type of subsystem where, for convenience, you group the objects together using system-specific means. Project files are a system- and language-independent way of building both static and dynamic libraries. They also support the concept of standalone libraries (SAL) which offers two significant properties: the initialization of the library is either automatic or very simple; a change in the implementation part of the library implies minimal post-compilation actions on the complete system and potentially no action at all for dynamic SALs.

- Project extension

  A project extension is a project that represents an augmentation of the original project. Suppose for instance that a subsystem `S` is shared by two unrelated systems `S1` and `S2`. If `S1` requires a change in `S`, but this change

would adversely affect S2, then a modified version of S can be defined as an extension of S, a "delta" to the original version which only contains the new source files (those that have been changed) and the objects that are affected by these changes.

# 3 Building with GPRbuild

## 3.1 Command Line

Three elements can optionally be specified on GPRbuild's command line:

- the main project file,
- the switches for GPRbuild itself or for the tools it drives, and
- the main source files.

The general syntax is thus:

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

GPRbuild requires a project file, which may be specified on the command line, either directly or through the '`-P`' switch. If not specified, then GPRbuild uses the project file '`default.gpr`' if there is one in the current working directory. Otherwise, if there is only one project file in the current working directory, GPRbuild uses this project file.

Main source files represent the sources to be used as the main programs. If they are not specified on the command line, GPRbuild uses the source files specified with the `Main` attribute in the project file. If none exists, then no executable will be built.

When source files are specified along with the option '`-c`', then recompilation will be considered only for those source files. In all other cases, GPRbuild compiles or recompiles all sources in the project tree that are not up to date, and builds or rebuilds libraries that are not up to date.

If invoked without the '`--config=`' or '`--autoconf=`' options, then GPRbuild will look for a configuration project file '`default.cgpr`', or '`<targetname>.cgpr`' if option '`--terget=<targetname>`' is used. If there is no such file in the default locations expected by GPRbuild (<install>/share/gpr and the current directory) then GPRbuild will invoke GPRconfig with the languages from the project files, and create a configuration project file '`auto.cgpr`' in the object directory of the main project. The project '`auto.cgpr`' will be rebuilt at each GPRbuild invocation unless you use the switch '`--autoconf=path/auto.cgpr`', which will use the configuration project file if it exists and create it otherwise.

Options given on the GPRbuild command line may be passed along to individual tools by preceding them with one of the "command line separators" shown below. Options following the separator, up to the next separator (or end of the command line), are passed along. The different command line separators are:

- '`-cargs`'

  The arguments that follow up to the next command line separator are options for all compilers for all languages. Example: '`-cargs -g`'

- '`-cargs:<language name>`'

  The arguments that follow up to the next command line separator are options for the compiler of the specific language.

  Examples:

  - '`-cargs:Ada -gnatf`'
  - '`-cargs:C -E`'
- '`-bargs`'

  The arguments that follow up to the next command line separator are options for all binder drivers.
- '`-bargs:<language name>`'

  The arguments that follow up to the next command line separators are options for the binder driver of the specific language.

  Examples:

  - '`-bargs:Ada binder_prefix=ppc-elf`'
  - '`-bargs:C++ c_compiler_name=ccppc`'
- '`-largs`'

  The arguments that follow up to the next command line separator are options for the linker.
- '`-gargs`'

  The arguments that follow up to the next command line separator are options for GPRbuild itself. Usually '`-gargs`' is specified after one or several other command line separators.

## 3.2 Switches

The switches that are interpreted directly by GPRbuild are listed below.

First, the switches that may be specified only on the command line, but not in package Builder of the main project:

- '`--version`'

  Display information about GPRbuild: version, origin and legal status, then exit successfully, ignoring other options.
- '`--help`'

  Display GPRbuild usage, then exit successfully, ignoring other options.
- '`--display-paths`'

  Display two lines: the configuration project file search path and the user project file search path, then exit successfully, ignoring other options.

- '`--config=<config project file name>`'

  This specifies the configuration project file name. By default, the configuration project file name is '`default.cgpr`'. Option '`--config=`' cannot be specified more than once. The configuration project file specified with '`--config=`' must exist.

- '`--autoconf=<config project file name>`'

  This specifies a configuration project file name that already exists or will be created automatically. Option '`--autoconf=`' cannot be specified more than once. If the configuration project file specified with '`--autoconf=`' exists, then it is used. Otherwise, GPRconfig is invoked to create it automatically.

- '`--target=<targetname>`'

  This specifies that the default configuration project file is '`<targetname>.cgpr`'. If no configuration project file with this name is found, then GPRconfig is invoked with option '`--target=<targetname>`'to create a configuration project file '`auto.cgpr`'.

  Note: only one of '`--config`', '`--autoconf`' or '`--target=`' can be specified.

- '`--subdirs=<subdir>`'

  This indicates that the real directories (except the source directories) are subdirectories of the directories specified in the project files. This applies in particular to object directories, library directories and exec directories. If the directories do not exist, they are created automatically.

- '`--direct-import-only`'

  This indicates that sources of a project should import only sources or header files from directly imported projects, that is those projects mentioned in a with clause and the projects they extend directly or indirectly. A check is done in the compilation phase, after a successful compilation, that the sources follow these restrictions. For Ada sources, the check is fully enforced. For non Ada sources, the check is partial, as in the dependency file there is no distinction between header files directly included and those indirectly included. The check will fail if there is no possibility that a header file in a non directly imported project could have been indirectly imported. If the check fails, the compilation artifacts (dependency file, object file, switches file) are deleted.

- '`-aP dir`' (Add directory '`dir`' to project search path)

  Specify to GPRbuild to add directory '`dir`' to the user project file search path, before the default directory.

- '`-b`' (Bind only)

  Specify to GPRbuild that the post-compilation (or binding) phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- '`-c`' (Compile only)

  Specify to GPRbuild that the compilation phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- '`-d`' (Display progress)

  Display progress for each source, up to date or not, as a single line *completed x out of y (zz%)...* If the file needs to be compiled this is displayed after the invocation of the compiler. These lines are displayed even in quiet output mode (switch '`-q`').

- '`-eL`' (Follow symbolic links when processing project files)

  By default, symbolic links on project files are not taken into account when processing project files. Switch '`-eL`' changes this default behavior.

- '`-F`' (Full project path name in brief error messages)

  By default, in non verbose mode, when an error occurs while processing a project file, only the simple name of the project file is displayed in the error message. When switch '`-F`' is used, the full path of the project file is used. This switch has no effect when switch '`-v`' is used.

- '`-l`' (Link only)

  Specify to GPRbuild that the linking phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- '`-o name`' (Choose an alternate executable name)

  Specify the file name of a single executable. Switch '`-o`' cannot be used with several mains on the command line, nor with no main on the command line and several mains in attribute `Main` of the main project.

- '`-p`' or '`--create-missing-dirs`' (Create missing object, library and exec directories)

  By default, GPRbuild checks that the object, library and exec directories specified in project files exist. Switch '`-p`' instructs GPRbuild to attempt to create missing directories.

- '`-P proj`' (use Project file *proj*)

  Specify the path name of the main project file. The space between '`-P`' and the project file name is optional. Specifying a project file name (with suffix '`.gpr`') may be used in place of option '`-P`'. Exactly one main project file can be specified.

- '`-u`' (Unique compilation, only compile the given files)

  If there are sources specified on the command line, only compile these sources. If there are no sources specified on the command line, compile all the sources of the main project.

  In both cases, do not attempt the binding and the linking phases.

- '`-U`' (Compile all sources of all projects)

  If there are sources specified on the command line, only compile these sources. If there are no sources specified on the command line, compile all the sources of all the projects in the project tree.

  In both cases, do not attempt the binding and the linking phases.

- '`-vPx`' (Specify verbosity when parsing Project Files)

  By default, GPRbuild does not display anything when processing project files, except when there are errors. This default behavior is obtained with switch '`-vP0`'. Switches '`-vP1`' and '`-vP2`' yield increasingly detailed output.

- '`-Xnm=val`' (Specify an external reference for Project Files)

  Specify an external reference that may be queried inside the project files using built-in function `external`. For example, with '`-XBUILD=DEBUG`', `external("BUILD")` inside a project file will have the value `"DEBUG"`.

Then, the switches that may be specified on the command line as well as in package Builder of the main project (attribute Switches):

- '`-f`' (Force recompilations)

  Force the complete processing of all phases (or of those explicitly specified) even when up to date.

- '`-j<num>`' (use *num* simultaneous compilation jobs)

  By default, GPRbuild invokes one compiler at a time. With switch '`-j`', it is possible to instruct GPRbuild to spawn several simultaneous compilation jobs if needed. For example, '`-j2`' for two simultaneous compilation jobs or '`-j4`' for four.

- '`-k`' (Keep going after compilation errors)

  By default, GPRbuild stops spawning new compilation jobs at the first compilation failure. Using switch '`-k`', it is possible to attempt to compile/recompile all the sources that are not up to date, even when some compilations failed. The post-compilation phase and the linking phase are never attempted if there are compilation failures, even when switch '`-k`' is used.

- '`-q`' (Quiet output)

  Do not display anything except errors and progress (switch '`-d`'). Cancel any previous switch '`-v`'.

- '`-s`' (recompile if compilation switches have changed)

  By default, GPRbuild will not recompile a source if all dependencies are satisfied. Switch '`-s`' instructs GPRbuild to recompile sources when a different set of compilation switches has been used in the previous compilation, even if all dependencies are satisfied. Each time GPRbuild invokes a compiler,

it writes a text file that lists the switches used in the invocation of the compiler, so that it can retrieve these switches if '`-s`' is used later.

- '`-v`' (Verbose output)
  Display full paths, all options used in spawned processes, and reasons why these processes are spawned. Cancel any previous switch '`-q`'.
- '`-we`' (Treat all warnings as errors)
  When '`-we`' is used, any warning during the processing of the project files becomes an error and GPRbuild does not attempt any of the phases.
- '`-wn`' (Treat warnings as warnings)
  Switch '`-wn`' may be used to restore the default after '`-we`' or '`-ws`'.
- '`-ws`' (Suppress all warnings)
  Do not generate any warnings while processing the project files.

## 3.3 Initialization

Before performing one or several of its three phases, GPRbuild has to read the command line, obtain its configuration, and process the project files.

If GPRbuild is invoked with an invalid switch or without any project file on the command line, it will fail immediately.

Examples:

```
$ gprbuild -P
gprbuild: project file name missing after -P

$ gprbuild -P c_main.gpr -WW
gprbuild: illegal option "-WW"
```

GPRbuild looks for the configuration project file first in the current working directory, then in the default configuration project directory. If the GPRbuild executable is located in a subdirectory '`<prefix>/bin`', then the default configuration project directory is '`<prefix>/share/gpr`', otherwise there is no default configuration project directory.

When it has found its configuration project path, GPRbuild needs to obtain its configuration. By default, the file name of the main configuration project is '`default.cgpr`'. This default may be modified using the switch '`--config=...`'

Example:

```
$ gprbuild --config=my_standard.cgpr -P my_project.gpr
```

If GPRbuild cannot find the main configuration project on the configuration project path, then it will look for all the languages specified in the user project tree and invoke GPRconfig to create a configuration project file named '`auto.cgpr`' that is located in the object directory of the main project file.

Once it has found the configuration project, GPRbuild will process its configuration: if a single string attribute is specified in the configuration project and

is not specified in a user project, then the attribute is added to the user project. If a string list attribute is specified in the configuration project then its value is prepended to the corresponding attribute in the user project.

After GPRbuild has processed its configuration, it will process the user project file or files. If these user project files are incorrect then GPRbuild will fail with the appropriate error messages:

```
$ gprbuild -P my_project.gpr
ada_main.gpr:3:26: "src" is not a valid directory
gprbuild: "my_project.gpr" processing failed
```

Once the user project files have been dealt with successfully, GPRbuild will start its processing.

## 3.4 Compilation of one or several sources

If GPRbuild is invoked with -c and there are one or several source file names specified on the command line, GPRbuild will compile or recompile these sources, if they are not up to date or if -f is also specified. Then GPRbuild will stop its execution, even if -b or -l are specified.

The options/switches used to compile these sources are described in section Section 3.5 [Compilation Phase], page 23.

## 3.5 Compilation Phase

When switch '`-c`' is used or when switches '`-b`' or '`-l`' are not used, GPRbuild will first compile or recompile all the sources that are not up to date in all the projects in the project tree.

GPRbuild will first consider each source and decide if it needs to be (re)compiled.

A source needs to be compiled in the following cases:

- Switch '`-f`' (force recompilations) is used
- The object file does not exist
- The source is more recent than the object file
- The dependency file does not exist
- The source is more recent than the dependency file
- The switch file does not exist
- The source is more recent than the switch file
- The dependency file cannot be read
- The dependency file is empty
- The dependency file has a wrong format
- A source listed in the dependency file does not exist

- A source listed in the dependency file has an incompatible time stamp
- A source listed in the dependency file has been replaced
- Switch '`-s`' is used and the source has been compiled with different switches or with the same switches in a different order

When a source is successfully compiled, the following files are normally created in the object directory of the project of the source:

- An object file
- A dependency file, except when the dependency kind for the language is `none`
- A switch file

The compiler for the language of the source is invoked with the following switches/options:

- The required compilation switches for the language
- The compilation switches coming from package `Compiler` of the project of the source
- The compilation switches specified on the command line for all compilers, after '`-cargs`'
- The compilation switches for the language of the source, specified after '`-cargs:<language>`'
- Various other options including a switch to create the dependency file while compiling, a switch to specify a configuration file, a switch to specify a mapping file, and switches to indicate where to look for other source or header files that are needed to compile the source.

If compilation is needed, then all the options/switches, except those described as "Various other options" are written to the switch file. The switch file is a text file. Its file name is obtained by replacing the suffix of the source with '`.cswi`'. For example, the switch file for source '`main.adb`' is '`main.cswi`' and for '`toto.c`' it is '`toto.cswi`'.

If the compilation is successful, then if the creation of the dependency file is not done during compilation but after (see configuration attribute `Compute_Dependency`), then the process to create the dependency file is invoked.

If GPRbuild is invoked with a switch '`-j`' specifying more than one compilation process, then several compilation processes for several sources of possibly different languages are spawned concurrently.

For each project file, attribute Interfaces may be declared. Its value is a list of sources or header files of the project file. For a project file extending another one, directly or indirectly, inherited sources may be in the list. When Interfaces is not declared, all sources or header files are part of the interface of the project. When Interfaces is declared, only those sources or header files are

part of the interface of the project file. After a successful compilation, gprbuild checks that all imported or included sources or header files that are from an imported project are part of the interface of the imported project. If this check fails, the compilation is invalidated and the compilation artifacts (dependency, object and switches files) are deleted.

Example:

```
project Prj is
   for Languages use ("Ada", "C");
   for Interfaces use ("pkg.ads", "toto.h");
end Prj;
```

If a source from a project importing project Prj imports sources from Prj other than package Pkg or includes header files from Prj other than "toto.h", then its compilation will be invalidated.

## 3.6 Post-Compilation Phase

*To be provided in a subsequent version of the document.*

## 3.7 Linking Phase

*To be provided in a subsequent version of the document.*

# 4 Cleaning up with GPRclean

The GPRclean tool removes the files created by GPRbuild. At a minimum, to invoke GPRclean you must specify a main project file in a command such as `gprclean proj.gpr` or `gprclean -P proj.gpr`.

Examples of invocation of GPRclean:

```
gprclean -r prj1.gpr
gprclean -c -P prj2.gpr
```

## 4.1 Switches for GPRclean

The switches for GPRclean are:

- '`--config=<main config project file name>`' : Specify the configuration project file name
- '`--autoconf=<config project file name>`'

  This specifies a configuration project file name that already exists or will be created automatically. Option '`--autoconf=`' cannot be specified more than once. If the configuration project file specified with '`--autoconf=`' exists, then it is used. Otherwise, GPRconfig is invoked to create it automatically.
- '`-c`' : Only delete compiler-generated files. Do not delete executables and libraries.
- '`-f`' : Force deletions of unwritable files
- '`-F`' : Display full project path name in brief error messages
- '`-h`' : Display this message
- '`-n`' : Do not delete files, only list files to delete
- '`-P<proj>`' : Use Project File <*proj*>.
- '`-q`' : Be quiet/terse. There is no output, except to report problems.
- '`-r`' : (recursive) Clean all projects referenced by the main project directly or indirectly. Without this switch, GPRclean only cleans the main project.
- '`-v`' : Verbose mode
- '`-vPx`' : Specify verbosity when parsing Project Files. x = 0 (default), 1 or 2.
- '`-Xnm=val`' : Specify an external reference for Project Files.

# 5 Configuring with GPRconfig

## 5.1 Using GPRconfig

### 5.1.1 Description

The GPRconfig tool helps you generate the configuration files for GPRbuild. It automatically detects the available compilers on your system and, after you have selected the one needed for your application, it generates the proper configuration file.


In general, you will not launch GPRconfig explicitly. Instead, it is used implicitly by GPRbuild through the use of `--config` and `--autoconf` switches

### 5.1.2 Command line arguments

GPRconfig supports the following command line switches:

‘`--target=platform`’
: This switch indicates the target computer on which your application will be run. It is mostly useful for cross configurations. Examples include ‘`ppc-elf`’, ‘`ppc-vx6-windows`’. It can also be used in native configurations and is useful when the same machine can run different kind of compilers such as mingw32 and cygwin on Windows or x86-32 and x86-64 on GNU Linux. Since different compilers will often return a different name for those targets, GPRconfig has an extensive knowledge of which targets are compatible, and will for example accept ‘`x86-linux`’ as an alias for ‘`i686-pc-linux-gnu`’. The default target is the machine on which GPRconfig is run.

  If you enter the special target ‘`all`’, then all compilers found on the `PATH` will be displayed.

‘`--show-targets`’
: As mentioned above, GPRconfig knows which targets are compatible. You can use this switch to find the list of targets that are compatible with `--target`.

‘`--config=language[,version[,runtime[,path[,name]]]]`’
: The intent of this switch is to preselect one or more compilers directly from the command line. This switch takes several optional arguments, which you can omit simply by passing the empty string. When omitted, the arguments will be computed automatically by GPRconfig.

In general, only '`language`' needs to be specified, and the first compiler on the `PATH` that can compile this language will be selected. As an example, for a multi-language application programmed in C and Ada, the command line would be:

```
--config=Ada --config=C
```

'`path`' is the directory that contains the compiler executable, for instance '`/usr/bin`' (and not the installation prefix '`/usr`').

'`name`' should be one of the compiler names defined in the GPRconfig knowledge base. The list of supported names can be found in the output of '`-h`', and includes '`GNAT`', '`GCC`',.... This name is generally not needed, but can be used to distinguish among several compilers that could match the other arguments of '`--config`'.

```
gprconfig --config Ada,,,/usr/bin # automatic parameters
gprconfig --config C,,,/usr/bin,GCC # automatic version
```

This switch is also the only possibility to include in your project some languages that are not associated with a compiler. This is sometimes useful especially when you are using environments like GPS that support project files. For instance, if you select "Project file" as a language, the files matching the '`.gpr`' extension will be shown in the editor, although they of course play no role for gprbuild itself.

'`--batch`' If this switch is specified, GPRconfig automatically selects the first compiler matching each of the `--config` switches, and generates the configuration file immediately. It will not display an interactive menu.

'`-o file`' This specifies the name of the configuration file that will be generated. If this switch is not specified, a default file is generated in the installation directory of GPRbuild (assuming you have write access to that directory), so that it is automatically picked up by GPRbuild later on. If you select a different output file, you will need to specify it to GPRbuild.

'`--db directory`'
'`--db-`' Indicates another directory that should be parsed for GPRconfig's knowledge base. Most of the time this is only useful if you are creating your own XML description files locally. The second version of the switch prevents GPRconfig from reading its default knowledge base.

'`-h`' Generates a brief help message listing all GPRconfig switches and the default value for their arguments. This includes the location of the knowledge base, the default target,...

### 5.1.3 Interactive use

When you launch GPRconfig, it first searches for all compilers it can find on your `PATH`, that match the target specified by '`--target`'. It is recommended, although not required, that you place the compilers that you expect to use for your application in your `PATH` before you launch `gprconfig`, since that simplifies the setup.

GPRconfig then displays the list of all the compilers it has found, along with the language they can compile, the run-time they use (when applicable),.... It then waits for you to select one of the compilers. This list is sorted by language, then by order in the `PATH` environment variable (so that compilers that you are more likely to use appear first), then by run-time names and finally by version of the compiler. Thus the first compiler for any language is most likely the one you want to use.

You make a selection by entering the letter that appears on the line for each compiler (be aware that this letter is case sensitive). If the compiler was already selected, it is deselected.

A filtered list of compilers is then displayed: only compilers that target the same platform as the selected compiler are now shown. GPRconfig then checks whether it is possible to link sources compiled with the selected compiler and each of the remaining compilers; when linking is not possible, the compiler is not displayed. Likewise, all compilers for the same language are hidden, so that you can only select one compiler per language.

As an example, if you need to compile your application with several C compilers, you should create another language, for instance called C2, for that purpose. That will give you the flexibility to indicate in the project files which compiler should be used for which sources.

The goal of this filtering is to make it more obvious whether you have a good chance of being able to link. There is however no guarantee that GPRconfig will know for certain how to link any combination of the remaining compilers.

You can select as many compilers as are needed by your application. Once you have finished selecting the compilers, select (s), and GPRconfig will generate the configuration file.

## 5.2 The GPRconfig knowledge base

GPRconfig itself has no hard-coded knowledge of compilers. Thus there is no need to recompile a new version of GPRconfig when a new compiler is distributed.

The role and format of the knowledge base are irrelevant for most users of GPRconfig, and are only needed when you need to add support

for new compilers. You can skip this section if you only want to learn how to use GPRconfig.

All knowledge of compilers is embedded in a set of XML files called the *knowledge base*. Users can easily contribute to this general knowledge base, and have GPRconfig immediately take advantage of any new data.

The knowledge base contains various kinds of information:

- Compiler description

  When it is run interactively, GPRconfig searches the user's `PATH` for known compilers, and tries to deduce their configuration (version, supported languages, supported targets, run-times, ...). From the knowledge base GPRconfig knows how to extract the relevant information about a compiler.

  This step is optional, since a user can also enter all the information manually. However, it is recommended that the knowledge base explicitly list its known compilers, to make configuration easier for end users.

- Specific compilation switches

  When a compiler is used, depending on its version, target, run-time,..., some specific command line switches might have to be supplied. The knowledge base is a good place to store such information.

  For instance, with the GNAT compiler, using the soft-float runtime should force *gprbuild* to use the '`-msoft-float`' compilation switch.

- Linker options

  Linking a multi-language application often has some subtleties, and typically requires specific linker switches. These switches depend on the list of languages, the list of compilers,....

- Unsupported compiler mix

  It is sometimes not possible to link together code compiled with two particular compilers. The knowledge base should store this information, so that end users are informed immediately when attempting to use such a compiler combination.

The end of this section will describe in more detail the format of this knowledge base, so that you can add your own information and have GPRconfig advantage of it.

### 5.2.1 General file format

The knowledge base is implemented as a set of XML files. None of these files has a special name, nor a special role. Instead, the user can freely create new files, and put them in the knowledge base directory, to contribute new knowledge.

The location of the knowledge base is '`$prefix/share/gprconfig`', where '`$prefix`' is the directory in which GPRconfig was installed. Any file with

extension '.xml' in this directory will be parsed automatically by GPRconfig at startup.

All files must have the following format:

```
<?xml version="1.0">
<gprconfig>
...
</gprconfig>
```

The root tag must be <gprconfig>.

The remaining sections in this chapter will list the valid XML tags that can be used to replace the ". . ." code above. These tags can either all be placed in a single XML files, or split across several files.

### 5.2.2 Compiler description

One of the XML tags that can be specified as a child of <gprconfig> is <compiler_description>. This node and its children describe one of the compilers known to GPRconfig. The tool uses them when it initially looks for all compilers known on the user's PATH environment variable.

This is optional information, but simplifies the use of GPRconfig, since the user is then able to omit some parameters from the '--config' command line argument, and have them automatically computed.

The <compiler_description> node doesn't accept any XML attribute. However, it accepts a number of child tags that explain how to query the various attributes of the compiler. The child tags are evaluated (if necessary) in the same order as they are documented below.

<name> This tag contains a simple string, which is the name of the compiler. This name must be unique across all the configuration files, and is used to identify that compiler_description node.

```
 <compiler_description>
<name>GNAT</name>
</compiler_description>
```

<executable>
This tag contains a string, which is the name of an executable to search for on the PATH. Examples are 'gnatls', 'gcc',. . .

In some cases, the tools have a common suffix, but a prefix that might depend on the target. For instance, GNAT uses 'gnatmake' for native platforms, but 'powerpc-wrs-vxworks-gnatmake' for cross-compilers to VxWorks. Most of the compiler description is the same, however. For such cases, the value of the executable node is considered as beginning a regular expression. The tag also accepts an attribute prefix, which is an integer indicating the parenthesis

group that contains the prefix. In the following example, you obtain the version of the GNAT compiler by running either `gnatls` or `powerpc-wrs-vxworks-gnatls`, depending on the name of the executable that was found.

The regular expression needs to match the whole name of the file, i.e. it contains an implicit "^" at the start, and an implicit "$" at the end. Therefore if you specify '`.*gnatmake`' as the regexp, it will not match '`gnatmake-debug`'.

A special case is when this node is empty (but it must be specified!). In such a case, you must also specify the language (see <language> below) as a simple string. It is then assumed that the specified language does not require a compiler. In the configurations file (see Section 5.2.3 [Configurations], page 39), you can test whether that language was specified on the command line by using a filter such as

```
 <compilers>
  <compiler language="name"/>
 </compilers>
<executable prefix="1">(powerpc-wrs-vxworks-)?gnatmake</executable>
<version><external>${PREFIX}gnatls -v</external></version>
```

GPRconfig searches in all directories listed on the PATH for such an executable. When one is found, the rest of the `<compiler_description>` children are checked to know whether the compiler is valid. The directory in which the executable was found becomes the "current directory" for the remaining XML children.

`<target>`

This node indicates how to query the target architecture for the compiler. See Section 5.2.2.1 [GPRconfig external values], page 35 for valid children.

`<version>`

This tag contains any of the nodes defined in Section 5.2.2.1 [GPRconfig external values], page 35 below. It shows how to query the version number of the compiler. If the version cannot be found, the executable will not be listed in the list of compilers.

`<variable name="varname">`

This node will define a user variable which may be later referenced. The variables are evaluated just after the version but before the languages and the runtimes nodes. See Section 5.2.2.1 [GPRconfig external values], page 35 below for valid children of this node. If the evaluation of this variable is empty then the compiler is considered as invalid.

`<languages>`
: This node indicates how to query the list of languages. See Section 5.2.2.1 [GPRconfig external values], page 35 below for valid children of this node.

  The value returned by the system will be split into words. As a result, if the returned value is "ada,c,c++", there are three languages supported by the compiler (and three entries are added to the menu when using GPRconfig interactively).

  If the value is a simple string, the words must be comma-separated, so that you can specify languages whose name include spaces. However, if the actual value is computed from the result of a command, the words can also be space-separated, to be compatible with more tools.

`<runtimes>`
: This node indicates how to query the list of supported runtimes for the compiler. See Section 5.2.2.1 [GPRconfig external values], page 35 below for valid children. The returned value is split into words as for `<languages>`.

### 5.2.2.1 External values

A number of the XML nodes described above can contain one or more children, and specify how to query a value from an executable. Here is the list of valid contents for these nodes. The `<directory>` and `<external>` children can be repeated multiple times, and the `<filter>` and `<must_match>` nodes will be applied to each of these. The final value of the external value is the concatenation of the computation for each of the `<directory>` and `<external>` nodes.

- A simple string

  A simple string given in the node indicates a constant. For instance, the list of supported languages might be defined as:

  ```
  <compiler_description>
  <name>GNAT</name>
  <executable>gnatmake</executable>
  <languages>Ada</languages>
  </compiler_description>
  ```

  for the GNAT compiler, since this is an Ada-only compiler.

  Variables can be referenced in simple strings.

- `<getenv name="variable" />`

  If the contents of the node is an `<getenv>` child, the value of the environment variable `variable` is returned. If the variable is not defined, this is an error and the compiler is ignored.

```
<compiler_description>
<name>GCC-WRS</name>
<executable prefix="1">cc(arm|pentium)</executable>
<version>
<getenv name="WIND_BASE" />
</version>
</compile_description>
```

- `<external>command</external>`

  If the contents of the node is an `<external>` child, this indicates that a command should be run on the system. When the command is run, the current directory (i.e., the one that contains the executable found through the `<executable>` node, is placed first on the `PATH`. The output of the command is returned and may be later filtered. The command is not executed through a shell; therefore you cannot use output redirection, pipes, or other advanced features.

  For instance, extracting the target processor from `gcc` can be done with:

  ```
  <version>
  <external>gcc -dumpmachine</external>
  </version>
  ```

  Since the `PATH` has been modified, we know that the `gcc` command that is executed is the one from the same directory as the `<external>` node.

  Variables are substituted in *command*.

- `<grep regexp="regexp" group="0" />`

  This node must come after the previously described ones. It is used to further filter the output. The previous output is matched against the regular expression *regexp* and the parenthesis group specified by *group* is returned. By default, group is 0, which indicates the whole output of the command.

  For instance, extracting the version number from `gcc` can be done with:

  ```
  <version>
  <external>gcc -v</external>
  <grep regexp="^gcc version (\S+)" group="1" />
  </version>
  ```

- `<directory group="0">regexp</directory>`

  If the contents of the node is a `<directory>` child, this indicates that GPRconfig should find all the files matching the regular expression. Regexp is a path relative to the directory that contains the `<executable>` file, and should use unix directory separators (ie '/'), since the actual directory will be converted into this format before the match, for system independence of the knowledge base.

The group attribute indicates which parenthesis group should be returned. It defaults to 0 which indicates the whole matched path. If this attribute is a string rather than an integer, then it is the value returned.

*regexp* can be any valid regular expression. This will only match a directory name, not a subdirectory. Remember to quote special characters, including ".", if you do not mean to use a regexp.

For instance, finding the list of supported runtimes for the GNAT compiler is done with:

```
<runtimes>
<directory group="1">
\.\./lib/gcc/${TARGET}/.*/rts-(.*)/adainclude
</directory>
<directory group="default">
\.\./lib/gcc/${TARGET}/.*/adainclude
</directory>
</runtimes>
```

Note the second node, which matches the default run-time, and displays it as such.

- `<filter>value1,value2,...</filter>`

  This node must come after one the previously described ones. It is used to further filter the output. The previous output is split into words (it is considered as a comma-separated or space-separated list of words), and only those words in '`value1`', '`value2`',... are kept.

  For instance, the `gcc` compiler will return a variety of supported languages, including "ada". If we do not want to use it as an Ada compiler we can specify:

```
<languages>
<external regexp="languages=(\S+)" group="1">gcc -v</external>
<filter>c,c++,fortran</filter>
</languages>
```

- `<must_match>regexp</must_match>`

  If this node is present, then the filtered output is compared with the specified regular expression. If no match is found, then the executable is not stored in the list of known compilers.

  For instance, if you want to have a `<compiler_description>` tag specific to an older version of GCC, you could write:

```
<version>
<external regexp="gcc version (\S+)"
group="1">gcc -v </external>
<must_match>2.8.1</must_match>
</version>
```

  Other versions of gcc will not match this `<compiler_description>` node.

#### 5.2.2.2 Variable Substitution

The various compiler attributes defined above are made available as variables in the rest of the XML files. Each of these variable can be used in the value of the various nodes (for instance in `<directory>`), and in the configurations (see Section 1.1 [Configuration], page 3).

A variable is referenced by `${name}` where *name* is either a user variable or a predefined variable. An alternate reference is `$name` where *name* is a sequence of alpha numeric characters or underscores. Finally `$$` is replaced by a simple `$`.

User variables are defined by `<variable>` nodes and may override predefined variables. To avoid a possible override use lower case names.

Predefined variables are always in upper case. Here is the list of predefined variables

*${EXEC}* is the name of the executable that was found through `<executable>`. It only contains the basename, not the directory information.

*${HOST}* is replaced by the architecture of the host on which GPRconfig is running. This name is hard-coded in GPRconfig itself, and is generated by `configure` when GPRconfig was built.

*${TARGET}*
is replaced by the target architecture of the compiler, as returned by the `<target>` node. This is of course not available when computing the target itself.

*${VERSION}*
is replaced by the version of the compiler. This is not available when computing the target or, of course, the version itself.

*${PREFIX}*
is replaced by the prefix to the executable name, as defined by the `<executable>` node.

*${PATH}* is the current directory, i.e. the one containing the executable found through `<executable>`. It always ends with a directory separator.

*${GPRCONFIG_PREFIX}*
is the directory in which GPRconfig was installed (e.g '`"/usr/local/"`' if the executable is '`"/usr/local/bin/gprconfig"`'. This directory always ends with a directory separator.

*${LANGUAGE}*
is the language supported by the compiler, always folded to lowercase

*${RUNTIME}*
*${RUNTIME_DIR}*
: This string will always be substituted by the empty string when the value of the external value is computed. These are special strings used when substituting text in configuration chunks.

  *RUNTIME_DIR* always end with a directory separator.

If a variable is not defined, an error message is issued and the variable is substituted by an empty string.

### 5.2.3 Configurations

The second type of information stored in the knowledge base are the chunks of *gprbuild* configuration files.

Each of these chunks is also placed in an XML node that provides optional filters. If all the filters match, then the chunk will be merged with other similar chunks and placed in the final configuration file that is generated by GPRconfig.

For instance, it is possible to indicate that a chunk should only be included if the GNAT compiler with the soft-float runtime is used. Such a chunk can for instance be used to ensure that Ada sources are always compiled with the `-msoft-float` command line switch.

GPRconfig does not perform sophisticated merging of chunks. It simply groups packages together. For example, if the two chunks are:

```
chunk1:
package Language_Processing is
for Attr1 use ("foo");
end Language_Processing;
chunk2:
package Language_Processing is
for Attr1 use ("bar");
end Language_Processing;
```

Then the final configuration file will look like:

```
package Language_Processing is
for Attr1 use ("foo");
for Attr1 use ("bar");
end Language_Processing;
```

As a result, to avoid conflicts, it is recommended that the chunks be written so that they easily collaborate together. For instance, to obtain something equivalent to

```
package Language_Processing is
for Attr1 use ("foo", "bar");
end Language_Processing;
```

the two chunks above should be written as:

```
chunk1:
package Language_Processing is
for Attr1 use Language_Processing'Attr1 & ("foo");
end Language_Processing;
chunk2:
package Language_Processing is
for Attr1 use Language_Processing'Attr1 & ("bar");
end Language_Processing;
```

The chunks are described in a <configuration> XML node. The most important child of such a node is <config>, which contains the chunk itself. For instance, you would write:

```
<configuration>
... list of filters, see below
<config>
package Language_Processing is
for Attr1 use Language_Processing'Attr1 & ("foo");
end Language_Processing;
</config>
</configuration>
```

If <config> is an empty node (i.e., '<config/>' or '<config></config>') was used, then the combination of selected compilers will be reported as invalid, in the sense that code compiled with these compilers cannot be linked together. As a result, GPRconfig will not create the configuration file.

The special variables (see Section 5.2.2.2 [GPRconfig variable substitution], page 38) are also substituted in the chunk. That allows you to compute some attributes of the compiler (its path, the runtime,. . . ), and use them when generating the chunks.

The filters themselves are of course defined through XML tags, and can be any of:

<compilers negate="false">
: This filter contains a list of <compiler> children. The <compilers> filter matches if any of its children match. However, you can have several <compilers> filters, in which case they must all match. This can be used to include linker switches chunks. For instance, the following code would be used to describe the linker switches to use when GNAT 5.05 or 5.04 is used in addition to g++ 3.4.1:

```
<configuration>
<compilers>
<compiler name="GNAT" version="5.04" />
<compiler name="GNAT" version="5.05" />
</compilers>
<compilers>
<compiler name="G++" version="3.4.1" />
</compilers>
...
</configuration>
```

If the attribute *negate* is 'true', then the meaning of this filter is inverted, and it will match if none of its children matches.

The format of the <compiler> is the following:

```
<compiler name="name" version="..."
runtime="..." language="..." />
```

The name and language attributes, when specified, match the corresponding attributes used in the <compiler_description> children. All other attributes are regular expressions, which are matched against the corresponding selected compilers. When an attribute is not specified, it will always match. Matching is done in a case-insensitive manner.

For instance, to check a GNAT compiler in the 5.x family, use:

```
<compiler name="GNAT" version="5.\d+" />
```

<hosts negate="false">

This filter contains a list of <host> children. It matches when any of its children matches. You can specify only one <hosts> node. The format of <host> is a node with a single mandatory attribute *name*, which is a regexp matched against the architecture on which GPRconfig is running. The name of the architecture was computed by `configure` when GPRconfig was built.

If the *negate* attribute is 'true', then the meaning of this filter is inverted, and it will match when none of its children matches.

For instance, to active a chunk only if the compiler is running on an intel linux machine, use:

```
<hosts>
<host name="i.86-.*-linux(-gnu)?" />
</hosts>
```

<targets negate="false">

This filter contains a list of <target> children. It behaves exactly like <hosts>, but matches against the architecture targeted by the selected compilers. For instance, to activate a chunk only when the code is targeted for linux, use:

If the *negate* attribute is 'true', then the meaning of this filter is inverted, and it will match when none of its children matches.

```
<targets>
<target name="i.86-.*-linux(-gnu)?" />
</targets>
```

# 6 Configuration File Reference

GPRbuild needs to have a configuration file to know the different characteristics of the toolchains that can be used to compile sources and build libraries and executables.

A configuration file is a special kind of project file: it uses the same syntax as a standard project file. Attributes in the configuration file define the configuration. Some of these attributes have a special meaning in the configuration.

The default name of the configuration file, when not specified to GPRbuild by switches –config= or –autoconf= is 'default.cgpr'. Although the name of the configuration file can be any valid file name, it is recommended that its suffix be '.cgpr' (for Configuration GNAT Project), so that it cannot be confused with a standard project file which has the suffix '.gpr'.

When 'default.cgpr' cannot be found in the configuration project path, GPRbuild invokes GPRconfig to create a configuration file.

In the following description of the attributes, when an attribute is an associative array indexed by the language name, for example Spec_Suffix (<language>), then the name of the language is case insensitive. For example, both C and c are allowed.

Any attribute may appear in a configuration project file. All attributes in a configuration project file are inherited by each user project file in the project tree. However, usually only the attributes listed below make sense in the configuration project file.

## 6.1 Project Level Attributes

### 6.1.1 General Attributes

- Default_Language

  Specifies the name of the language of the immediate sources of a project when attribute Languages is not declared in the project. If attribute Default_Language is not declared in the configuration file, then each user project file in the project tree must have an attribute Languages declared, unless it extends another project. Example:

  ```
  for Default_Language use "ada";
  ```

- Run_Path_Option

  Specifies a "run path option"; i.e., an option to use when linking an executable or a shared library to indicate the path where to look for other libraries. The value of this attribute is a string list. When linking an executable or a shared library, the search path is concatenated with the last string in the list, which may be an empty string. Example:

```
for Run_Path_Option  use ("-Wl,-rpath,");
```

- Toolchain_Version (<language>)

  Specifies a version for a toolchain, as a single string. This toolchain version is passed to the library builder. Example:

  ```
  for Toolchain_Version ("Ada") use "GNAT 6.1";
  ```

  This attribute is used by GPRbind to decide on the names of the shared GNAT runtime libraries.

- Toolchain_Description (<language>)

  Specifies as a single string a description of a toolchain. This attribute is not directly used by GPRbuild or its auxiliary tools (GPRbind and GPRlib) but may be used by other tools, for example GPS. Example:

  ```
  for Toolchain_Description ("C") use "gcc version 4.1.3 20070425";
  ```

### 6.1.2 General Library Related Attributes

- Library_Support

  Specifies the level of support for library project. If this attribute is not specified, then library projects are not supported. The only potential values for this attribute are `none`, `static_only` and `full`. Example:

  ```
  for Library_Support use "full";
  ```

- Library_Builder

  Specifies the name of the executable for the library builder. Example:

  ```
  for Library_Builder use "/.../gprlib";
  ```

### 6.1.3 Archive Related Attributes

- Archive_Builder

  Specifies the name of the executable of the archive builder with the minimum options, if any. Example:

  ```
  for Archive_Builder use ("ar", "cr");
  ```

- Archive_Indexer

  Specifies the name of the executable of the archive indexer with the minimum options, if any. If this attribute is not specified, then there is no archive indexer. Example:

  ```
  for Archive_Indexer use ("ranlib");
  ```

- Archive_Suffix

  Specifies the suffix of the archives. If this attribute is not specified, then the suffix of the archives is defaulted to '.a'. Example:

  ```
  for Archive_Suffix use ".olb"; --  for VMS
  ```

- Library_Partial_Linker

  Specifies the name of the executable of the partial linker with the options to be used, if any. If this attribute is not specified, then there is no partial linking. Example:

```
for Library_Partial_Linker use ("gcc", "-nostdlib", "-Wl,-r", "-o");
```

### 6.1.4 Shared Library Related Attributes

- Shared_Library_Prefix

  Specifies the prefix of the file names of shared libraries. When this attribute is not specified, the prefix is `lib`. Example:

```
for Shared_Library_Prefix use ""; --  for Windows, if needed
```

- Shared_Library_Suffix

  Specifies the suffix of the file names of shared libraries. When this attribute is not specified, the suffix is '`.so`'. Example:

```
for Shared_Library_Suffix use ".dll"; --  for Windows
```

- Symbolic_Link_Supported

  Specifies if symbolic links are supported by the platforms. The possible values of this attribute are `"false"` (the default) and `"true"`. When this attribute is not specified, symbolic links are not supported.

```
for Symbolic_Link_Supported use "true";
```

- Library_Major_Minor_ID_Supported

  Specifies if major and minor IDs are supported for shared libraries. The possible values of this attribute are `"false"` (the default) and `"true"`. When this attribute is not specified, major and minor IDs are not supported.

```
for Library_Major_Minor_ID_Supported use "True";
```

- Library_Auto_Init_Supported

  Specifies if library auto initialization is supported. The possible values of this attribute are `"false"` (the default) and `"true"`. When this attribute is not specified, library auto initialization is not supported.

```
for Library_Auto_Init_Supported use "true";
```

- Shared_Library_Minimum_Switches

  Specifies the minimum options to be used when building shared library. These options are put in the appropriate section in the library exchange file when the library builder is invoked. Example:

```
for Shared_Library_Minimum_Switches use  ("-shared");
```

- Library_Version_Switches

  Specifies the option or options to be used when a library version is used. These options are put in the appropriate section in the library exchange file when the library builder is invoked. Example:

```
for Library_Version_Switches use ("-Wl,-soname,");
```

- Runtime_Library_Dir (<language>)
  Specifies the directory for the runtime libraries for the language. Example:
  ```
  for Runtime_Library_Dir ("Ada") use "/path/to/adalib";
  ```
  This attribute is used by GPRlib to link shared libraries with Ada code.

## 6.2 Package Naming

Attributes in package `Naming` of a configuration file specify defaults. These attributes may be used in user project files to replace these defaults.

The following attributes usually appear in package `Naming` of a configuration file:

- Spec_Suffix (<language>)
  Specifies the default suffix for a "spec" or header file. Examples:
  ```
  for Spec_Suffix ("Ada") use ".ads";
  for Spec_Suffix ("C")   use ".h";
  for Spec_Suffix ("C++") use ".hh";
  ```
- Body_Suffix (<language>)
  Specifies the default suffix for a "body" or a source file. Examples:
  ```
  for Body_Suffix ("Ada") use ".adb";
  for Body_Suffix ("C")   use ".c";
  for Body_Suffix ("C++") use ".cpp";
  ```
- Separate_Suffix
  Specifies the suffix for a subunit source file (separate) in Ada. If attribute `Separate_Suffix` is not specified, then the default suffix of subunit source files is the same as the default suffix for body source files. Example:
  ```
  for Separate_Suffix use ".sep";
  ```
- Casing
  Specifies the casing of spec and body files in a unit based language (such as Ada) to know how to map a unit name to its file name. The values for this attribute may only be `"lowercase"`, `"UPPERCASE"` and `"Mixedcase"`. The default, when attribute `Casing` is not specified is lower case. This attribute rarely needs to be specified, since on platforms where file names are not case sensitive (such as Windows or VMS) the default (lower case) will suffice.
- Dot_Replacement
  Specifies the string to replace a dot (".") in unit names of a unit based language (such as Ada) to obtain its file name. If there is any unit based language in the configuration, attribute `Dot_Replacement` must be declared. Example:
  ```
  for Dot_Replacement use "-";
  ```

## 6.3 Package Builder

- Executable_Suffix

  Specifies the default executable suffix. If no attribute `Executable_Suffix` is declared, then the default executable suffix for the host platform is used. Example:

```
for Executable_Suffix use ".exe";
```

## 6.4 Package Compiler

### 6.4.1 General Compilation Attributes

- Driver (<language>)

  Specifies the name of the executable for the compiler of a language. The single string value of this attribute may be an absolute path or a relative path. If relative, then the execution path is searched. Specifying the empty string for this attribute indicates that there is no compiler for the language.

  Examples:

```
for Driver ("C++") use "g++";
for Driver ("Ada") use "/.../bin/gcc";
for Driver ("Project file") use "";
```

- Required_Switches (<language>)

  Specifies the minimum options that must be used when invoking the compiler of a language. Examples:

```
for Required_Switches ("C")   use ("-c", "-x", "c");
for Required_Switches ("Ada") use ("-c", "-x", "ada", "-gnatA");
```

- PIC_Option (<language>)

  Specifies the option or options that must be used when compiling a source of a language to be put in a shared library. Example:

```
for PIC_Option ("C") use ("-fPIC");
```

### 6.4.2 Mapping File Related Attributes

- Mapping_File_Switches (<language>)

  Specifies the switch or switches to be used to specify a mapping file to the compiler. When attribute `Mapping_File_Switches` is not declared, then no mapping file is specified to the compiler. The value of this attribute is a string list. The path name of the mapping file is concatenated with the last string in the string list, which may be empty. Example:

```
for Mapping_File_Switches ("Ada") use ("-gnatem=");
```

- Mapping_Spec_Suffix (<language>)

  Specifies, for unit based languages that support mapping files, the suffix in the mapping file that needs to be added to the unit name for specs. Example:

  ```
  for Mapping_Spec_Suffix ("Ada") use "%s";
  ```

- Mapping_Body_Suffix (<language>)

  Specifies, for unit based languages that support mapping files, the suffix in the mapping file that needs to be added to the unit name for bodies. Example:

  ```
  for Mapping_Spec_Suffix ("Ada") use "%b";
  ```

### 6.4.3 Config File Related Attributes

In the value of config file attributes defined below, there are some placeholders that GPRbuild will replace. These placeholders are:

- %u : the unit name
- %f : the file name of the source
- %s : the spec suffix
- %b : the body suffix
- %c : the casing
- %d : the dot replacement string

Attributes:

- Config_File_Switches (<language>)

  Specifies the switch or switches to be used to specify a configuration file to the compiler. When attribute `Config_File_Switches` is not declared, then no config file is specified to the compiler. The value of this attribute is a string list. The path name of the config file is concatenated with the last string in the string list, which may be empty. Example:

  ```
  for Config_File_Switches ("Ada") use ("-gnatec=");
  ```

- Config_Body_File_Name (<language>)

  Specifies the line to be put in a config file to indicate the file name of a body. Example:

  ```
  for Config_Body_File_Name ("Ada") use
      "pragma Source_File_Name_Project (%u, Body_File_Name => ""%f"");";
  ```

- Config_Spec_File_Name (<language>)

  Specifies the line to be put in a config file to indicate the file name of a spec. Example:

  ```
  for Config_Spec_File_Name ("Ada") use
      "pragma Source_File_Name_Project (%u, Spec_File_Name => ""%f"");";
  ```

- Config_Body_File_Name_Pattern (<language>)

  Specifies the line to be put in a config file to indicate a body file name pattern. Example:

```
for Config_Body_File_Name_Pattern ("Ada") use
    "pragma Source_File_Name_Project " &
    "  (Body_File_Name  => ""*%b"","  &
    "   Casing          => %c,"  &
    "   Dot_Replacement => ""%d"");";
```

- Config_Spec_File_Name_Pattern (<language>)

  Specifies the line to be put in a config file to indicate a spec file name pattern. Example:

```
for Config_Spec_File_Name_Pattern ("Ada") use
    "pragma Source_File_Name_Project " &
    "  (Spec_File_Name  => ""*%s"","  &
    "   Casing          => %c,"  &
    "   Dot_Replacement => ""%d"");";
```

- Config_File_Unique (<language>)

  Specifies, for languages that support config files, if several config files may be indicated to the compiler, or not. This attribute may have only two values: `"true"` or `"false"` (case insensitive). The default, when this attribute is not specified, is `"false"`. When the value `"true"` is specified for this attribute, GPRbuild will concatenate the config files, if there are more than one. Example:

```
for Config_File_Unique ("Ada") use "True";
```

### 6.4.4 Dependency Related Attributes

There are two dependency-related attributes: `Dependency_Switches` and `Dependency_Driver`. If neither of these two attributes are specified for a language other than Ada, then the source needs to be (re)compiled if the object file does not exist or the source file is more recent than the object file or the switch file.

- Dependency_Switches (<language>)

  For languages other than Ada, attribute `Dependency_Switches` specifies the option or options to add to the compiler invocation so that it creates the dependency file at the same time. The value of attribute `Dependency_Option` is a string list. The name of the dependency file is added to the last string in the list, which may be empty. Example:

```
for Dependency_Switches ("C") use ("-Wp,-MD,");
```

  With these `Dependency_Switches`, when compiling '`file.c`' the compiler will be invoked with the option '`-Wp,-MD,file.d`'.

- Dependency_Driver (<language>)

  Specifies the command and options to create a dependency file for a source. The full path name of the source is appended to the last string of the string list value. Example:

  ```
  for Dependency_Driver ("C") use ("gcc", "-E", "-Wp,-M", "");
  ```

  Usually, attributes `Dependency_Switches` and `Dependency_Driver` are not both specified.

### 6.4.5 Search Path Related Attributes

- Include_Switches (<language>)

  Specifies the option or options to use when invoking the compiler to indicate that a directory is part of the source search path. The value of this attribute is a string list. The full path name of the directory is concatenated with the last string in the string list, which may be empty. Example:

  ```
  for Include_Switches ("C") use ("-I");
  ```

  Attribute `Include_Switches` is ignored if either one of the attributes `Include_Path` or `Include_Path_File` are specified.

- Include_Path (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the source search path. The value of the environment variable is the source search path to be used by the compiler. Example:

  ```
  for Include_Path ("C")   use "CPATH";
  for Include_Path ("Ada") use "ADA_INCLUDE_PATH";
  ```

  Attribute `Include_Path` is ignored if attribute `Include_Path_File` is declared for the language.

- Include_Path_File (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the source search path. The value of the environment variable is the path name of a text file that contain the path names of the directories of the source search path. Example:

  ```
  for Include_Path_File ("Ada") use "ADA_PRJ_INCLUDE_FILE";
  ```

## 6.5 Package Binder

- Driver (<language>)

  Specifies the name of the executable of the binder driver. When this attribute is not specified, there is no binder for the language. Example:

  ```
  for Driver ("Ada") use "/.../gprbind";
  ```

- Required_Switches (<language>)

  Specifies the minimum options to be used when invoking the binder driver. These options are put in the appropriate section in the binder exchange file, one option per line. Example:

  ```
  for Required_Switches ("Ada") use ("--prefix=<prefix>");
  ```

- Prefix (<language>)

  Specifies the prefix to be used in the name of the binder exchange file. Example:

  ```
  for Prefix ("C++") use ("c__");
  ```

- Objects_Path (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the object search path. The value of the environment variable is the object search path to be used by the compiler. Example:

  ```
  for Objects_Path ("Ada") use "ADA_OBJECTS_PATH";
  ```

- Objects_Path_File (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the object search path. The value of the environment variable is the path name of a text file that contain the path names of the directories of the object search path. Example:

  ```
  for Objects_Path_File ("Ada") use "ADA_PRJ_OBJECTS_FILE";
  ```

## 6.6 Package Linker

- Driver

  Specifies the name of the executable of the default linker. Example:

  ```
  for Driver use "g++";
  ```

- Required_Switches

  Specifies the minimum options to be used when invoking the default linker.

# Table of Contents