GPR2 Library Quick Start Guide

Release 27.0w

AdaCore

Jun 10, 2026

CONTENTS:

Version 27.0w
Date: Jun 10, 2026

CHAPTER

ONE

QUICK START

The GPR2 library is an Ada library for loading, querying, and processing GPR project trees programmatically. It is the foundation of all GPR tools (GPRbuild, GPRclean, GPRinstall, and others) and is the recommended API for tools that need to read or interpret GPR project files.

This guide covers the most common operations — loading a project tree, iterating over views, reading attributes, listing sources, and navigating Ada compilation units — with complete code snippets for each. For a full API reference, see the *GPR2 Library Reference*.

Note

This guide assumes familiarity with the GPR project file language. If you are new to GPR, read the *GPR User Guide* for a task-oriented introduction or the *GPR Reference Manual* for a complete language specification before proceeding.

This chapter shows the most common operations step by step: loading a project tree, inspecting views, reading attributes, and working with sources and units. Each section builds on the previous one. A minimal project file `hello.gpr` is used throughout:

```
project Hello is
   for Source_Dirs use ("src");
   for Object_Dir  use "obj";
   for Main        use ("hello.adb");
end Hello;
```

1.1 Loading a project tree

The entry point is `GPR2.Project.Tree.Object`. Options are passed through a `GPR2.Options.Object` value; the `Add_Switch` procedure maps directly to the command-line switches that GPR tools accept.

```
with Ada.Text_IO;          use Ada.Text_IO;
with GPR2.Options;
with GPR2.Project.Tree;

procedure Load_Example is
   Options : GPR2.Options.Object;
   Tree    : GPR2.Project.Tree.Object;
begin
   Options.Add_Switch (GPR2.Options.P, "hello.gpr");
   --  Add scenario variables with Options.X:
```

(continues on next page)

(continued from previous page)

```
   --    Options.Add_Switch (GPR2.Options.X, "BUILD", "", "debug");
   --  Point to a non-standard runtime:
   --    Options.Add_Switch (GPR2.Options.RTS, "light-cortex-m4f");

   if not Tree.Load (Options) then
      Put_Line ("Loading failed - check messages above.");
   end if;
end Load_Example;
```

`Tree.Load` returns `False` when a fatal error prevents the tree from being usable. It always reports diagnostics through the configured reporter (the console by default). Any leftover messages can also be inspected programmatically after the call:

```
for Msg of Tree.Log_Messages.all loop
   Put_Line (Msg.Format);
end loop;
```

`GPR2.Message.Object` exposes `Level` (`Warning`, `Error`, `End_User`, `Hint`, `Lint`), `Message` (raw text), `Sloc` (file/line/column), and `Format` (a human-readable string combining all of the above).

Using `Options.Autoconf` instead of `Options.Config` lets the library generate a configuration project automatically, which is the typical mode when no pre-built `.cgpr` file is available:

```
Options.Add_Switch (GPR2.Options.Autoconf, "auto.cgpr");
```

For cross-compilation, add the target before loading:

```
Options.Add_Switch (GPR2.Options.Target, "arm-elf");
Options.Add_Switch (GPR2.Options.RTS,    "light-cortex-m4f");
```

1.2 Iterating over views

After a successful load, each GPR file in the closure is available as a `GPR2.Project.View.Object`. The tree is iterable directly:

```
for V of Tree loop
   Put_Line (String (V.Name) & " at " & V.Path_Name.Value);
end loop;
```

The root project is always accessible via `Tree.Root_Project`.

The iterator accepts `Kind`, `Filter`, and `Status` parameters. `Filter` is a record whose fields are Boolean flags for each project kind (`F_Standard`, `F_Library`, `F_Aggregate`, etc.). To visit only library projects:

```
for V of Tree.Iterate
  (Filter => (GPR2.Project.F_Library => True, others => False))
loop
   Put_Line ("Library: " & String (V.Name));
end loop;
```

Each view exposes its kind (`V.Kind`), its project directory (`V.Dir_Name`), object directory (`V.Object_Directory`), and the set of projects it imports (`V.Imports`).

A specific project can be retrieved by name from any view:

```
declare
   Dep : constant GPR2.Project.View.Object :=
            Tree.Root_Project.View_For ("some_library");
begin
   Put_Line (Dep.Path_Name.Value);
end;
```

1.3 Reading attributes

Predefined attribute identifiers live in `GPR2.Project.Registry.Attribute` (abbreviated `PRA` below). The `Attribute` function takes a `Q_Attribute_Id` and an optional index.

```
with GPR2.Project.Attribute;
with GPR2.Project.Attribute_Index;
with GPR2.Project.Registry.Attribute;

--  In the body, with a loaded Tree:

declare
   package PRA renames GPR2.Project.Registry.Attribute;
   View : constant GPR2.Project.View.Object := Tree.Root_Project;
   Attr : GPR2.Project.Attribute.Object;
begin
   --  Single-valued attribute (no index):
   Attr := View.Attribute (PRA.Object_Dir);
   if Attr.Is_Defined then
      Put_Line ("Object_Dir = " & Attr.Value.Text);
   end if;

   --  Indexed attribute (Compiler.Default_Switches for Ada):
   Attr := View.Attribute
             (Name  => PRA.Compiler.Default_Switches,
              Index => GPR2.Project.Attribute_Index.Create
                         (GPR2.Ada_Language));
   if Attr.Is_Defined then
      for V of Attr.Values loop
         Put (V.Text & " ");
      end loop;
      New_Line;
   end if;
end;
```

Use `Attribute.Kind` (`Single` or `List`) to decide whether to call `Value` or `Values`. `Attribute.Is_Default` is `True` when the value comes from the built-in default rather than the project file. `Attribute.Is_From_Config` is `True` when it originates from the configuration project.

To iterate over all attributes of a view (or a package):

```
--  All top-level attributes (no defaults, no config-inherited):
for A of View.Attributes (With_Defaults => False, With_Config => False) loop
   Put (GPR2.Image (A.Name.Id.Attr));
   if A.Has_Index then
```

(continues on next page)

(continued from previous page)

```
      Put (" (""" & A.Index.Text & """)");
   end if;
   Put (" = ");
   if A.Kind = GPR2.Project.Registry.Attribute.Single then
      Put_Line (A.Value.Text);
   else
      for V of A.Values loop
         Put (V.Text & " ");
      end loop;
      New_Line;
   end if;
end loop;

--  Attributes inside a package, e.g. Compiler:
for A of View.Attributes
  (GPR2.Project.Registry.Pack.Compiler,
   With_Defaults => False,
   With_Config   => False)
loop
   Put_Line (GPR2.Image (A.Name.Id.Attr));
end loop;
```

1.4 Working with sources

Source information is not populated during `Tree.Load` by default. Call `Tree.Update_Sources` to request it, or pass `Artifacts_Info_Level => GPR2.Sources_Only` (or `Sources_Units`) to `Tree.Load` directly:

```
Tree.Update_Sources (Option => GPR2.Sources_Only);
--  Use Sources_Units to also resolve unit names from source content.
--  Use Sources_Units_Artifacts to additionally load ALI file data
--  (dependency information).
```

After that, sources are available through any view:

```
with GPR2.Build.Source;

--  Iterate every source owned by a view:
for Src of View.Sources loop
   Put_Line (Src.Path_Name.Value);
   Put_Line ("  language: " & GPR2.Image (Src.Language));
end loop;

--  Look up a specific source by simple file name:
declare
   Src : constant GPR2.Build.Source.Object :=
            View.Source ("hello.adb");
begin
   if Src.Is_Defined then
      Put_Line ("Found: " & Src.Path_Name.Value);
   end if;
end;
```

`View.Visible_Source` performs the same lookup but searches the full closure of the view, which is useful when a source is provided by an imported project.

Each `GPR2.Build.Source.Object` (a child of `GPR2.Build.Source_Base`) exposes:

- `Path_Name` - full filesystem path (`GPR2.Path_Name.Object`).
- `Language` - a `Language_Id` value; compare with `GPR2.Ada_Language`, or convert with `GPR2.Image`.
- `Kind` - one of `S_Spec`, `S_Body`, `S_Separate` (for Ada) or `S_Body` for other languages.
- `Has_Units` / `Units` - unit information, populated when `Sources_Units` or higher was requested.
- `Owning_View` - the view that declares this source.

1.5 Working with units

Units are Ada-specific and are retrieved through the root of a namespace (typically the root project or a non-aggregate project in a dependency). Call `Update_Sources` with at least `Sources_Units` first.

```
with GPR2.Build.Compilation_Unit;

Tree.Update_Sources (Option => GPR2.Sources_Units);

--  Look up a single unit by dotted name (case-insensitive):
declare
   Unit : constant GPR2.Build.Compilation_Unit.Object :=
            Tree.Root_Project.Unit ("hello");
begin
   if Unit.Is_Defined then
      Put_Line ("Unit: " & String (Unit.Name));

      if Unit.Has_Part (GPR2.Unit.S_Spec) then
         Put_Line ("  spec: "
                   & String (Unit.Spec.Source.Simple_Name));
      end if;

      if Unit.Has_Part (GPR2.Unit.S_Body) then
         Put_Line ("  body: "
                   & String (Unit.Main_Body.Source.Simple_Name));
      end if;

      for Sep of Unit.Separates loop
         Put_Line ("  separate: "
                   & String (Sep.Source.Simple_Name));
      end loop;
   end if;
end;

--  Iterate all units visible from the root project:
for Cursor in Tree.Root_Project.Units.Iterate loop
   declare
      U : constant GPR2.Build.Compilation_Unit.Object :=
            GPR2.Build.Compilation_Unit.Maps.Element (Cursor);
   begin
      Put_Line (String (U.Name));
```

(continues on next page)

(continued from previous page)

```
   end;
end loop;
```

`View.Own_Units` is the counterpart restricted to units whose sources belong to the view itself, and does not require `Is_Namespace_Root`.

1.6 Scenario variables

Scenario variables (`external(...)` references in project files) are managed through `GPR2.Context.Object`. Set the initial context before loading, and call `Set_Context` to switch scenarios on an already-loaded tree:

```
with GPR2.Context;

--  Before loading:
declare
   Ctx : GPR2.Context.Object;
begin
   Ctx.Include ("BUILD", "release");
   Options.Add_Switch (GPR2.Options.X, "BUILD", "", "release");
   --  Or load first with default values, then switch:
end;

--  Switch after loading:
declare
   Ctx : GPR2.Context.Object := Tree.Context;
begin
   Ctx.Include ("BUILD", "debug");
   if not Tree.Set_Context (Ctx) then
      Put_Line ("Context change produced errors.");
   end if;
end;
```

The optional `Changed` callback passed to `Set_Context` is invoked for every view that is affected by the context change, which is useful for invalidating cached data derived from project attributes.

CHAPTER

TWO

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

INDEX