19. GNAT and Libraries

This chapter describes how to build and use libraries with GNAT, and also shows how to recompile the GNAT run-time library. You should be familiar with the Project Manager facility (see section 11. GNAT Project Manager) before reading this chapter.

19.1 Introduction to Libraries in GNAT

A library is, conceptually, a collection of objects which does not have its own main thread of execution, but rather provides certain services to the applications that use it. A library can be either statically linked with the application, in which case its code is directly included in the application, or, on platforms that support it, be dynamically linked, in which case its code is shared by all applications making use of this library.

GNAT supports both types of libraries. In the static case, the compiled code can be provided in different ways. The simplest approach is to provide directly the set of objects resulting from compilation of the library source files. Alternatively, you can group the objects into an archive using whatever commands are provided by the operating system. For the latter case, the objects are grouped into a shared library.

In the GNAT environment, a library has three types of components:

- Source files.
- `` `ALI' `` files. See section 2.8 The Ada Library Information Files.
- Object files, an archive or a shared library.

A GNAT library may expose all its source files, which is useful for documentation purposes. Alternatively, it may expose only the units needed by an external user to make use of the library. That is to say, the specs reflecting the library services along with all the units needed to compile those specs, which can include generic bodies or any body implementing an inlined routine. In the case of *stand-alone libraries* those exposed units are called *interface units* (see section 19.3 Stand-alone Ada Libraries).

All compilation units comprising an application, including those in a library, need to be elaborated in an order partially defined by Ada's semantics. GNAT computes the elaboration order from the `` `ALI' `` files and this is why they constitute a mandatory part of GNAT libraries. Except in the case of *stand-alone libraries*, where a specific library elaboration routine is produced independently of the application(s) using the library.

19.2 General Ada Libraries

19.2.1 Building a library

The easiest way to build a library is to use the Project Manager, which supports a special type of project called a *Library Project* (see section 11.12 Library Projects).

A project is considered a library project, when two project-level attributes are defined in it: `Library_Name` and `Library_Dir`. In order to control different aspects of library configuration, additional optional project-level attributes can be specified:

`Library_Kind`
: This attribute controls whether the library is to be static or dynamic

`Library_Version`
: This attribute specifies the library version; this value is used during dynamic linking of shared libraries to determine if the currently installed versions of the binaries are compatible.

`Library_Options`
`Library_GCC`
: These attributes specify additional low-level options to be used during library generation, and redefine the actual application used to generate library.

The GNAT Project Manager takes full care of the library maintenance task, including recompilation of the source files for which objects do not exist or are not up to date, assembly of the library archive, and installation of the library (i.e., copying associated source, object and `` `ALI' `` files to the specified location).

Here is a simple library project file:

```
project My_Lib is
   for Source_Dirs use ("src1", "src2");
   for Object_Dir use "obj";
   for Library_Name use "mylib";
   for Library_Dir use "lib";
   for Library_Kind use "dynamic";
end My_lib;
```

and the compilation command to build and install the library:

```
  $ gnatmake -Pmy_lib
```

It is not entirely trivial to perform manually all the steps required to produce a library. We recommend that you use the GNAT Project Manager for this task. In special cases where this is not desired, the necessary steps are discussed below.

There are various possibilities for compiling the units that make up the library: for example with a Makefile (see section 20. Using the GNU `make` Utility) or with a conventional script. For simple libraries, it is also possible to create a dummy main program which depends upon all the packages that comprise the interface of the library. This dummy main program can then be given to `gnatmake`, which will ensure that all necessary objects are built.

After this task is accomplished, you should follow the standard procedure of the underlying operating system to produce the static or shared library.

Here is an example of such a dummy program:

```
with My_Lib.Service1;
with My_Lib.Service2;
with My_Lib.Service3;
procedure My_Lib_Dummy is
begin
   null;
end;
```

Here are the generic commands that will build an archive or a shared library.

```
# compiling the library
$ gnatmake -c my_lib_dummy.adb

# we don't need the dummy object itself
$ rm my_lib_dummy.o my_lib_dummy.ali

# create an archive with the remaining objects
$ ar rc libmy_lib.a *.o
# some systems may require "ranlib" to be run as well

# or create a shared library
$ gcc -shared -o libmy_lib.so *.o
# some systems may require the code to have been compiled with -fPIC

# remove the object files that are now in the library
$ rm *.o

# Make the ALI files read-only so that gnatmake will not try to
# regenerate the objects that are in the library
$ chmod -w *.ali
```

Please note that the library must have a name of the form `` `libxxx.a' `` or `` `libxxx.so' `` (or `` `libxxx.dll' `` on Windows) in order to be accessed by the directive `` `-lxxx' `` at link time.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

19.2.2 Installing a library

If you use project files, library installation is part of the library build process. Thus no further action is needed in order to make use of the libraries that are built as part of the general application build. A usable version of the library is installed in the directory specified by the `Library_Dir` attribute of the library project file.

You may want to install a library in a context different from where the library is built. This situation arises with third party suppliers, who may want to distribute a library in binary form where the user is not expected to be able to recompile the library. The simplest option in this case is to provide a project file slightly different from the one used to build the library, by using the `externally_built` attribute. For instance, the project file used to build the library in the previous section can be changed into the following one when the library is installed:

```
project My_Lib is
   for Source_Dirs use ("src1", "src2");
   for Library_Name use "mylib";
   for Library_Dir use "lib";
   for Library_Kind use "dynamic";
   for Externally_Built use "true";
end My_lib;
```

This project file assumes that the directories `` `src1' ``, `` `src2' ``, and `` `lib' `` exist in the directory containing the project file. The `externally_built` attribute makes it clear to the GNAT builder that it should not attempt to recompile any of the units from this library. It allows the library provider to restrict the source set to the minimum necessary for clients to make use of the library as described in the first section of this chapter. It is the responsibility of the library provider to install the necessary sources, ALI files and libraries in the directories mentioned in the project file. For convenience, the user's library project file should be installed in a location that will be searched automatically by the GNAT builder. These are the directories referenced in the `ADA_LIBRARY_PATH` environment variable (see section 11.5 Importing Projects), and also the default GNAT library location that can be queried with `gnatls -v` and is usually of the form $gnat_install_root/lib/gnat.

When project files are not an option, it is also possible, but not recommended, to install the library so that the sources needed to use the library are on the Ada source path and the ALI files & libraries be on the Ada Object path (see 3.3 Search Paths and the Run-Time Library (RTL). Alternatively, the system administrator can place general-purpose libraries in the default compiler paths, by specifying the libraries' location in the configuration files `` `ada_source_path' `` and `` `ada_object_path' ``. These configuration files must be located in the GNAT installation tree at the same place as the gcc spec file. The location of the gcc spec file can be determined as follows:

```
$ gcc -v
```

The configuration files mentioned above have a simple format: each line must contain one unique directory name. Those names are added to the corresponding path in their order of appearance in the file. The names can be either absolute or relative; in the latter case, they are relative to where theses files are located.

The files `` `ada_source_path' `` and `` `ada_object_path' `` might not be present in a GNAT installation, in which case, GNAT will look for its run-time library in the directories `` `adainclude' `` (for the sources) and `` `adalib' `` (for the objects and `` `ALI' `` files). When the files exist, the compiler does not look in `` `adainclude' `` and `` `adalib' ``, and thus the `` `ada_source_path' `` file must contain the location for the GNAT run-time sources (which can simply be `` `adainclude' ``). In the same way, the `` `ada_object_path' `` file must contain the location for the GNAT run-time objects (which can simply be `` `adalib' ``).

You can also specify a new default path to the run-time library at compilation time with the switch `` `--RTS=rts-path' ``. You can thus choose / change the run-time library you want your program to be compiled with. This switch is recognized by `gcc`, `gnatmake`, `gnatbind`, `gnatls`, `gnatfind` and `gnatxref`.

It is possible to install a library before or after the standard GNAT library, by reordering the lines in the configuration files. In general, a library must be installed before the GNAT library if it redefines any part of it.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

19.2.3 Using a library

Once again, the project facility greatly simplifies the use of libraries. In this context, using a library is just a matter of adding a `with` clause in the user project. For instance, to make use of the library `My_Lib` shown in examples in earlier sections, you can write:

```
with "my_lib";
project My_Proj is
  ...
end My_Proj;
```

Even if you have a third-party, non-Ada library, you can still use GNAT's Project Manager facility to provide a wrapper for it. For example, the following project, when `with`ed by your main project, will link with the third-party library `` `liba.a' ``:

```
project Liba is
   for Externally_Built use "true";
   for Library_Dir use "lib";
   for Library_Name use "a";
   for Library_Kind use "static";
end Liba;
```

This is an alternative to the use of `pragma Linker_Options`. It is especially interesting in the context of systems with several interdependant static libraries where finding a proper linker order is not easy and best be left to the tools having visibility over project dependancy information.

In order to use an Ada library manually, you need to make sure that this library is on both your source and object path (see 3.3 Search Paths and the Run-Time Library (RTL) and 4.4 Search Paths for `gnatbind`). Furthermore, when the objects are grouped in an archive or a shared library, you need to specify the desired library at link time.

For example, you can use the library `` `mylib' `` installed in `` `/dir/my_lib_src' `` and `` `/dir/my_lib_obj' `` with the following commands:

```
$ gnatmake -aI/dir/my_lib_src -aO/dir/my_lib_obj my_appl \
  -largs -lmy_lib
```

This can be expressed more simply:

```
$ gnatmake my_appl
```

when the following conditions are met:

- `` `/dir/my_lib_src' `` has been added by the user to the environment variable `ADA_INCLUDE_PATH`, or by the administrator to the file `` `ada_source_path' ``
- `` `/dir/my_lib_obj' `` has been added by the user to the environment variable `ADA_OBJECTS_PATH`, or by the administrator to the file `` `ada_object_path' ``
- a pragma `Linker_Options` has been added to one of the sources. For example:

  ```
  pragma Linker_Options ("-lmy_lib");
  ```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

19.3 Stand-alone Ada Libraries

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

19.3.1 Introduction to Stand-alone Libraries

A Stand-alone Library (abbreviated "SAL") is a library that contains the necessary code to elaborate the Ada units that are included in the library. In contrast with an ordinary library, which consists of all sources, objects and `` `ALI' `` files of the library, a SAL may specify a restricted subset of compilation units to serve as a library interface. In this case, the fully self-sufficient set of files will normally consist of an objects archive, the sources of interface units' specs, and the `` `ALI' `` files of interface units. If an interface spec contains a generic unit or an inlined subprogram, the body's source must also be provided; if the units that must be provided in the source form depend on other units, the source and `ALI' files of those must also be provided.

The main purpose of a SAL is to minimize the recompilation overhead of client applications when a new version of the library is installed. Specifically, if the interface sources have not changed, client applications do not need to be recompiled. If, furthermore, a SAL is provided in the shared form and its version, controlled by `Library_Version` attribute, is not changed, then the clients do not need to be relinked.

SALs also allow the library providers to minimize the amount of library source text exposed to the clients. Such "information hiding" might be useful or necessary for various reasons.

Stand-alone libraries are also well suited to be used in an executable whose main routine is not written in Ada.

[<] [>]   [<<] [Up] [>>]   [Top] [Contents] [Index] [?]

19.3.2 Building a Stand-alone Library

GNAT's Project facility provides a simple way of building and installing stand-alone libraries; see 11.13 Stand-alone Library Projects. To be a Stand-alone Library Project, in addition to the two attributes that make a project a Library Project (`Library_Name` and `Library_Dir`; see 11.12 Library Projects), the attribute `Library_Interface` must be defined. For example:

```
     for Library_Dir use "lib_dir";
     for Library_Name use "dummy";
     for Library_Interface use ("int1", "int1.child");
```

Attribute `Library_Interface` has a non-empty string list value, each string in the list designating a unit contained in an immediate source of the project file.

When a Stand-alone Library is built, first the binder is invoked to build a package whose name depends on the library name (`b~dummy.ads/b' in the example above). This binder-generated package includes initialization and finalization procedures whose names depend on the library name (`dummyinit` and `dummyfinal` in the example above). The object corresponding to this package is included in the library.

You must ensure timely (e.g., prior to any use of interfaces in the SAL) calling of these procedures if a static SAL is built, or if a shared SAL is built with the project-level attribute `Library_Auto_Init` set to `"false"`.

For a Stand-Alone Library, only the `ALI' files of the Interface Units (those that are listed in attribute `Library_Interface`) are copied to the Library Directory. As a consequence, only the Interface Units may be imported from Ada units outside of the library. If other units are imported, the binding phase will fail.

The attribute `Library_Src_Dir` may be specified for a Stand-Alone Library. `Library_Src_Dir` is a simple attribute that has a single string value. Its value must be the path (absolute or relative to the project directory) of an existing directory. This directory cannot be the object directory or one of the source directories, but it can be the same as the library directory. The sources of the Interface Units of the library that are needed by an Ada client of the library will be copied to the designated directory, called the Interface Copy directory. These sources include the specs of the Interface Units, but they may also include bodies and subunits, when pragmas `Inline` or `Inline_Always` are used, or when there is a generic unit in the spec. Before the sources are copied to the Interface Copy directory, an attempt is made to delete all files in the Interface Copy directory.

Building stand-alone libraries by hand is somewhat tedious, but for those occasions when it is necessary here are the steps that you need to perform:

- Compile all library sources.

- Invoke the binder with the switch `` `-n' `` (No Ada main program), with all the `` `ALI' `` files of the interfaces, and with the switch `` `-L' `` to give specific names to the `init` and `final` procedures. For example:
  ```
  gnatbind -n int1.ali int2.ali -Lsal1
  ```
- Compile the binder generated file:
  ```
  gcc -c b~int2.adb
  ```
- Link the dynamic library with all the necessary object files, indicating to the linker the names of the `init` (and possibly `final`) procedures for automatic initialization (and finalization). The built library should be placed in a directory different from the object directory.
- Copy the `ALI` files of the interface to the library directory, add in this copy an indication that it is an interface to a SAL (i.e. add a word `` `SL' `` on the line in the `` `ALI' `` file that starts with letter "P") and make the modified copy of the `` `ALI' `` file read-only.

Using SALs is not different from using other libraries (see 19.2.3 Using a library).

19.3.3 Creating a Stand-alone Library to be used in a non-Ada context

It is easy to adapt the SAL build procedure discussed above for use of a SAL in a non-Ada context.

The only extra step required is to ensure that library interface subprograms are compatible with the main program, by means of `pragma Export` or `pragma Convention`.

Here is an example of simple library interface for use with C main program:

```
package Interface is

   procedure Do_Something;
   pragma Export (C, Do_Something, "do_something");

   procedure Do_Something_Else;
   pragma Export (C, Do_Something_Else, "do_something_else");

end Interface;
```

On the foreign language side, you must provide a "foreign" view of the library interface; remember that it should contain elaboration routines in addition to interface subprograms.

The example below shows the content of `mylib_interface.h` (note that there is no rule for the naming of this file, any name can be used)

```
/* the library elaboration procedure */
extern void mylibinit (void);

/* the library finalization procedure */
extern void mylibfinal (void);

/* the interface exported by the library */
extern void do_something (void);
extern void do_something_else (void);
```

Libraries built as explained above can be used from any program, provided that the elaboration procedures (named `mylibinit` in the previous example) are called before the library services are used. Any number of libraries can be used simultaneously, as long as the elaboration procedure of each library is called.

Below is an example of a C program that uses the `mylib` library.

```
#include "mylib_interface.h"

int
main (void)
{
   /* First, elaborate the library before using it */
   mylibinit ();

   /* Main program, using the library exported entities */
   do_something ();
   do_something_else ();

   /* Library finalization at the end of the program */
   mylibfinal ();
   return 0;
}
```

Note that invoking any library finalization procedure generated by `gnatbind` shuts down the Ada run-time environment. Consequently, the finalization of all Ada libraries must be performed at the end of the program. No call to these libraries or to the Ada run-time library should be made after the finalization phase.

19.3.4 Restrictions in Stand-alone Libraries

The pragmas listed below should be used with caution inside libraries, as they can create incompatibilities with other Ada libraries:

- pragma `Locking_Policy`
- pragma `Queuing_Policy`
- pragma `Task_Dispatching_Policy`
- pragma `Unreserve_All_Interrupts`

When using a library that contains such pragmas, the user must make sure that all libraries use the same pragmas with the same values. Otherwise, `Program_Error` will be raised during the elaboration of the conflicting libraries. The usage of these pragmas and its consequences for the user should therefore be well documented.

Similarly, the traceback in the exception occurrence mechanism should be enabled or disabled in a consistent manner across all libraries. Otherwise, Program_Error will be raised during the elaboration of the conflicting libraries.

If the `Version` or `Body_Version` attributes are used inside a library, then you need to perform a `gnatbind` step that specifies all `` `ALI' `` files in all libraries, so that version identifiers can be properly computed. In practice these attributes are rarely used, so this is unlikely to be a consideration.

19.4 Rebuilding the GNAT Run-Time Library

It may be useful to recompile the GNAT library in various contexts, the most important one being the use of partition-wide configuration pragmas such as `Normalize_Scalars`. A special Makefile called `Makefile.adalib` is provided to that effect and can be found in the directory containing the GNAT library. The location of this directory depends on the way the GNAT environment has been installed and can be determined by means of the command:

```
 $ gnatls -v
```

The last entry in the object search path usually contains the gnat library. This Makefile contains its own documentation and in particular the set of instructions needed to rebuild a new library and to use it.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by *Mail Server* on *June, 15 2005* using *texi2html*