Debian policy for Ada

Seventh Edition for Debian 8.0 Jessie (2014-07-05) (preliminary)

Ludovic Brenta
Stephen Leake

Table of Contents

1 Introduction 1

2 Background information 2
- 2.1 History of GNAT 2
- 2.2 The variants of GNAT 2
- 2.3 Timeline of GNAT releases 3
  - 2.3.1 GNAT Pro releases 3
  - 2.3.2 Public releases from AdaCore 4
  - 2.3.3 GNAT Academic Programme and GNAT GPL Edition 4
  - 2.3.4 FSF releases 4
- 2.4 ASIS, GLADE and Florist 6
- 2.5 GDB, the GNU debugger 6
- 2.6 History of Ada standards 7

3 Ada and shared libraries 8
- 3.1 Package and shared object version numbers 8
  - 3.1.1 Scenario: multiple versions of a library 9
  - 3.1.2 Scenario: Upgrading a library 9
- 3.2 Ada Library Information files 10
  - 3.2.1 Scenario: Indirectly causing a package to FTBFS 10
  - 3.2.2 Avoiding the indirect FTBFS scenario 12
  - 3.2.3 Responsibilities of upstream authors and maintainers 15

4 The Debian Ada compiler 17

5 Policy for Ada libraries 18
- 5.1 Building a library for reuse 18
- 5.2 Library names and packaging structure 19
  - 5.2.1 Coexistence allowed 19
  - 5.2.2 No coexistence allowed 19
  - 5.2.3 Package names 19
  - 5.2.4 Inter-package dependencies 20
- 5.3 Files provided by the -dev package 20
  - 5.3.1 General rules 20
  - 5.3.2 Source files 21
  - 5.3.3 Ada object files 21
  - 5.3.4 Ada Library Information files 21
  - 5.3.5 Symbolic link to the shared library 22
  - 5.3.6 GNAT project file 22
  - 5.3.7 Documentation 23
- 5.4 Files provided by the run-time shared library package 23
- 5.5 Documentation 23
  - 5.5.1 Documents in the info format 23
  - 5.5.2 Examples 24
- 5.6 The separate debugging information package 24

6 Debian for Ada programmers 25
6.1 Installing an Ada development environment 25
6.2 In-place upgrades 25
6.3 Using shared libraries 26
6.4 Debugging programs that use shared libraries 27
6.5 Where to ask for help 27

7 Help wanted 28

Appendix A Bugs in GNAT 29

Appendix B A note about GNAT's shared libraries 30

Appendix C Ada transition in Debian 4.0 Etch 32
C.1 How the Ada compiler for Etch was chosen 32
C.2 Transition plan for Etch 32

Appendix D libgnat under GPL - for or against 34

1 Introduction

This document describes all there is to know about Ada in Debian. Most of it contains my personal thoughts and opinions, although I try to get the facts straight. I would appreciate it if you could send me corrections or additions at ludovic@ludovic-brenta.org.

After posting the first draft, I received numerous comments to improve this document. In particular, I would like to thank Dan Eilers of Irvine Compiler Corporation for providing me with details about GCC. Other contributors include Simon Wright, Ed Falis, Georg Bauhaus, Bob Rogers, and Stephe Leake.

Each chapter and appendix in this Policy is either informative or normative. Normative chapters contain binding rules that all packages compiled from Ada sources must follow; informative chapters only contain useful information.

2 Background information

(This chapter is informative)

2.1 History of GNAT

GNAT started as the GNU New-York University Ada Translator in 1994. Under a contract with the US Department of Defense, a team at NYU started writing GNAT as a reference implementation of the then upcoming Ada 95 ISO standard. The contract mandated the use of the GNU General Public Licence for the compiler.

After the Ada 95 standard was published, various corporations and DoD services started using GNAT in mission-critical projects, and asked for support. The authors of GNAT formed AdaCore to meet these needs and further develop GNAT. Today, AdaCore has offices in New York (`http://www.adacore.com`) and Paris (`http://www.adacore.fr`). GNAT is still free software under the GPL. AdaCore offers commercial support for it and continues development.

Over the years, several institutions contributed software to complement GNAT. For example, the Ecole Nationale Superieure de Telecommunications in Paris contributed GLADE, an implementation of Annex E (Distributed Systems) for GNAT. AdaCore commercially supports most of these contributions, and distributes them together with GNAT to supported customers. As a result, at one point, the GNAT acronym stood for GNU Ada Technology. Nowadays it is no longer considered an acronym.

2.2 The variants of GNAT

There are several places to get GNAT from, and different GNATs which, for lack of a better word, I call "variants". Each has a different binary interface from the others; it is therefore not possible to link object files produced by different variants into an executable; nor is it possible to link an executable produced by one variant with libraries produced by another. The variants of GNAT are:

- GNAT Pro, commercially supported and actively maintained by AdaCore (`http://www.adacore.com`). New releases are frequent but provided only to paying customers, under the GNAT-Modified GPL described below. AdaCore runs a test suite comprising more than 10,000 tests against this compiler, every night, on every supported platform. This test suite consists not only of the ACATS (the official ISO standard test suite for validated Ada compilers) but also of test cases for all bug reports ever submitted to AdaCore. This test suite thus ensures that no bug is ever reintroduced. GNAT Pro is being enhanced to implement the Ada 2005 standard; the new features are refined as the draft standard itself evolves. If you require GNAT Pro for professional work, please contact AdaCore at sales@adacore.com.
- The Public versions of GNAT ("p" releases). From time to time, AdaCore used to release a stable version of GNAT to the public at no cost and under the GNAT-Modified GPL. These releases benefited from bug fixes sent to paying customers, and underwent the same extensive test suite as GNAT Pro but were not supported by AdaCore. The last "p" release is GNAT 3.15p in October 2002. This is the default Ada compiler in Debian 3.1 "Sarge". It is still possible to download gnat 3.15p source and binaries for various platforms from mirrors such as Ada-Belgium and SourceForge.
- The FSF variant of GNAT. This is the Ada front-end which has been distributed as part of GCC since version 3.1. From GCC 3.1 to 4.3, the license terms of the run-time library are the GNAT-Modified GPL. For 4.4 and later, the licence terms are GPL version 3 with GCC Runtime Library Exception which is similar in spirit to the GMGPL.
- The GNAT Academic Programme (GAP) allows registered academics (i.e. university professors and their students) to obtain a copy of GNAT Pro, with support, at no cost. However,

in newer (2005 and later) releases, the license terms of the run-time library changed to pure GPL. Thus, binaries produced with this variant can only be distributed under GPL.

- The GNAT GPL Edition is similar to the GNAT Academic Programme but is available to the general public from AdaCore's Libre Software site. This Edition replaces the former "p" releases and receives no support from AdaCore.

The GNAT-Modified GPL is the GPL version 2 with the following text added to each source file:

> As a special exception, if other files instantiate generics from this unit or you link this unit with other files to produce an executable, this unit does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU Public License.

Starting with GCC 4.4, the library source files grant the GCC Run-time library exception by means of the following language:

> As a special exception under Section 7 of GPL version 3, you are granted additional permissions described in the GCC Runtime Library Exception, version 3.1, as published by the Free Software Foundation.

Thus, the GNAT-Modified GPL permits the distribution of proprietary software linked with the GNAT run-time library, libgnat. The pure GPL disallows this, as any binary linked with a GPL library must also be distributed under GPL or not at all. This does not apply to your source text: your source program is not linked with libgnat, therefore you can distribute it under whichever licensing terms you choose, even non-disclosure agreements.

2.3 Timeline of GNAT releases

2.3.1 GNAT Pro releases

Announced	Release	Based on
2003-02-18	3.16a	GCC 2.8.1
2003-07-01	3.16a1	GCC 2.8.1
2003-08-21	5.01a	GCC 3.2.3
2004-02-26	5.02a	GCC 3.2.3
2005-02-21	5.03a	GCC 3.4.1
2006-01-30	5.04	GCC 3.4.4
2006-05-09	5.05	GCC 3.4.6
2007-03-07	6.0.1	GCC 4.1.3
2008-03-05	6.1.1	GCC 4.1.3
2009-02-11	6.2.1	GCC 4.3.3
2010-02-09	6.3.1	GCC 4.4.2
2011-02-16	6.4.1	GCC 4.5.3
2012-02-28	7.0.1	GCC 4.5.4
2013-04-23	7.1	GCC 4.7.3
2014-02-25	7.2	GCC 4.7.4

In addition to these full releases, supported customers sometimes receive a "wavefront" release, with "w" in their version number, e.g. 5.02w. As the name suggests, the "wavefront" changes constantly as GNAT evolves.

Version 6.0.1 is the first to officially support the full Ada 2005 standard. It also uses an almost conventional three-component version numbering scheme in which the micro number is 0 for a beta release, 1 for a first production release, and 2 or greater for a bug-fix release.

Version 7.0.1 is the first to officially support the full (draft) Ada 2012 standard. Version 7.2 switches the default language to Ada 2012 and adds support for ARM Cortex processors running Android (as a cross-compiler).

2.3.2 Public releases from AdaCore

The dates in this timeline are the timestamps of the source files for each release, as stored in the tar.gz archive. These dates sometimes differ from the hardcoded "release date" which is visible with "gnatgcc -v".

Date	Release	Compiler back-end:
1995-11-29	3.01p	GCC 2.7.2
1996-05-03	3.04p	GCC 2.7.2
1996-06-07	3.05p	GCC 2.7.2
1996-10-23	3.07p	GCC 2.7.2
1997-01-21	3.09p	GCC 2.7.2
1997-08-14	3.10p	GCC 2.7.2
1998-12-18	3.11p	GCC 2.8.1
1999-09-24	3.12p	GCC 2.8.1
2000-07-19	3.13p	GCC 2.8.1
2001-10-07	3.14p	GCC 2.8.1
2002-10-24	3.15p	GCC 2.8.1
2005-09-15	2005	GCC 3.4.4
2006-05-22	2006	GCC 3.4.6
2007-05-11	2007	GCC 4.1.2
2008-06-12	2008	GCC 4.1.3
2009-05-30	2009	GCC 4.3.3
2010-06-09	2010	GCC 4.4.2
2011-06-16	2011	GCC 4.5.3
2012-06-25	2012	GCC 4.5.4
2013-05-29	2013	GCC 4.7.3
2014-03-22	2014	GCC 4.7.4

The GNAT GPL 2005 Edition changes the licensing terms of the run-time library to the pure GPL, thereby disallowing the distribution of non-GPL programs linked with this Edition's run-time library.

2.3.3 GNAT Academic Programme and GNAT GPL Edition

The GNAT Academic Programme started in 2004. Under this Programme, AdaCore made GNAT Pro available at no cost to registered academics. On top of this, professors received support from AdaCore at no cost. GAP versions 1.0 and 1.1 came with the full complement of tools and libraries then current for paying customers, all under GMGPL. Subsequent releases are identical to the GNAT GPL releases starting with 2005.

2.3.4 FSF releases

In October 2001, AdaCore contributed their most recent GNAT Pro source code for the Ada 95 front-end and runtime library to the GCC source repository. The first release of GCC that included GNAT was 3.1 in May 2002. After that, little activity took place on the Ada parts of GCC; thus, GCC 3.1, 3.2 and 3.3 contain essentially the same Ada front-end and library.

In October 2003, after the second merge of GNAT Pro into GCC, AdaCore closed their own public CVS server, and started doing routine commits to the FSF's repository.

AdaCore only work on the main line of development; this means that no, or very few changes take place in minor versions of GCC, and I don't show them in the table below. This also means that AdaCore only merge from their internal repository during Stage 1 of the development cycle

of each major release of GCC. As a consequence, GNAT GPL, GNAT Pro and GCC all have different release cycles. From the dates where Stage 1 ends, we can infer a rough correspondence between GNAT GPL and GCC releases but no GCC release is strictly equivalent to any GNAT GPL or GNAT Pro release.

Release date	**End of stage 1**	**Release**	**Roughly equivalent to**
	2001-10-01	First merge into GCC	-
2002-05-22	2001-10-15	3.1	-
2002-08-14	(see 3.1)	3.2	-
2003-05-13	2002-06-22	3.3	-
2004-04-18	3003-07-04	3.4	-
	2003-10-21	Second merge into GCC	-
2005-04-20	2004-07-01	4.0	-
2006-02-28	2005-04-25	4.1	GNAT GPL 2005
2007-05-13	2006-01-18	4.2	GNAT GPL 2006
2008-03-05	2007-01-20	4.3	GNAT GPL 2007
2009-04-21	2008-09-01	4.4	GNAT GPL 2008
2010-04-14	2009-10-01	4.5	GNAT GPL 2009
2011-03-25	2010-11-03	4.6	GNAT GPL 2010
2012-03-22	2011-11-08	4.7	GNAT GPL 2011
2013-03-22	2012-11-06	4.8	GNAT GPL 2012
2014-04-22	2013-11-21	4.9	GNAT GPL 2014

GCC 3.4 has several new packages in the GNAT library, most notably a first version of the Ada 2005 container library, and also the "limited with" feature of Ada 2005. It also benefits from the new targets introduced in recent versions of GCC, such as AMD64 (also known as x86-64). There are approximately 70 bug fixes relative to GCC 3.3.

However, it also has a serious regression: it does not support tasking on `powerpc-*-linux` (see ada/13897).

A subset of ACATS 2.5 (basically the executable tests but not the tests with intentional errors) is included in GCC 3.4 and later. GCC 3.4 passes all of these tests.

GCC 4.0's Ada front-end restores tasking on powerpc, fixes about 100 bugs, and contains a newer version of the container library. One very major change inside GCC 4.0 warranted the new major version number. This change, called "tree-SSA", is a new infrastructure for optimisers, and a new interface between front-ends and the back-end of the compiler. This required updating all the language front-ends. Unfortunately, these changes introduced some bugs too, including a few ACATS failures.

GCC 4.1's Ada front-end contains a further 60 bug fixes, brings the Ada.Containers library closer to the definitive Ada 2005 standard, and has a more mature interface to Tree-SSA. The entire ACATS passes in this release. This is the default Ada compiler in Debian 4.0 "Etch".

In GCC 4.2, the Ada run-time library provides many more packages from the new language standard, support AltiVec instructions on machines that support them, and a software emulation of AltiVec on machines that don't. The compiler front-end supports more Ada 2005 features, in particular related to interfaces, tagged protected and tagged task types.

GCC 4.3 brings 110 bug fixes and several additional Ada 2005 packages (e.g. Ada.Calendar.Formatting). This is the default Ada compiler in Debian 5.0 "Lenny".

GCC 4.4 has 40 more bug fixes and introduces support for multilib. Multilib is the ability of the compiler to support more than one ABI on the same machine and to select from several corresponding versions of the run-time library. Examples of machines supporting multilib include amd64 (with i386 as secondary), sparc64 (sparc32 as secondary), mips and mipsel (with n64, o32 and n32 ABIs). Unfortunately, the Debian maintainers have not yet found the time and

energy to add support for multilib in the Debian packages. This is the default Ada compiler in Debian 6.0 “Squeeze”.

GCC 4.5 improves stack overflow detection (enabled with -fstack-check), introduces a new version of the Partition Communications Subsystem of the Distributed Systems Annex and brings several more enhancements and bug fixes.

GCC 4.6 introduces preliminary support for Ada 2012. This is the default compiler in Debian 7.0 “Wheezy”.

GCC 4.8 adds several Ada 2012 packages like `Ada.Containers.Synchronized_Queue_Interfaces` and its various implementations.

GCC 4.9 completes the support for Ada 2012 and makes Ada 2012 the default mode of compilation (i.e. one needs to specify `-gnat05` explicitly to compile in Ada 2005 mode). This will be the default compiler in Debian 8.0 “Jessie”.

2.4 ASIS, GLADE and Florist

ASIS is the Ada Semantic Interface Specification. GNAT conforms to this Specification, and exposes the parse tree of programs, so that other programs can query and manipulate the parse tree. The ASIS distribution comes with several such programs, for example “gnatpp” a pretty-printer or “gnatelim” which eliminates unused subprograms. Third-party programs that take advantage of ASIS include the “gch” and “adastyle” style checkers, “adabrowse” the document generator, “adasubst” which does mass substitutions of identifier names, and “adadep” which manages dependencies between compilation units.

AdaCore make a source-only release of ASIS-for-GNAT as part of all GNAT Pro and GNAT GPL Editions but these releases are not coordinated with those of GCC. Because of its nature, ASIS-for-GNAT is tightly integrated with the compiler, to the point that its source distribution contains parts of the compiler’s internal run-time library. Thus, building ASIS-for-GNAT with a release of GCC requires replacing this partial copy of GNAT GPL’s run-time library with the corresponding parts of GCC. To achieve this, Debian introduces packages ‘`libgnatvsn-dev`’, ‘`libgnatvsnX.Y`’ and ‘`libgnatvsn-dbg`’, licensed under GNAT-Modified GPL, which contain those parts of GCC required for ASIS. These packages are built from the ‘`gnat-X.Y`’ source package.

GLADE is the Ecole Nationale de Telecommunications’ implementation of the Distributed Systems annex (Annex E) of the Ada standard. With GLADE, it is possible to write a set of programs that run on different machines and communicate by means of remote procedure calls and shared objects. In Ada parlance, these programs are called “partitions” and the entire system is the “program”. In 2008, GLADE was superseded by PolyORB, a general middleware technology supporting not only Annex E but also CORBA, SOAP and several other specifications for distributed systems, all at the same time.

Florist is Florida State University’s implementation of POSIX.5 standard, which allows Ada programs to use POSIX system services such as shared memory, pipes, and message queues.

All GNAT Pro, GAP, GPL and “p” releases of GNAT come with ASIS, GLADE or PolyORB, and Florist. In contrast, FSF releases come with none of them. Both GLADE and PolyORB rely on parts of GNAT’s internal implementation, so porting them from a GPL Edition to GCC requires intimate knowledge of GNAT.

2.5 GDB, the GNU debugger

Because debugging information is in a standard format in object files, you can use GDB to debug programs written in any language. However, special Ada support allows GDB to demangle names and to understand Ada syntax when displaying variables.

GNAT 3.15p comes with a patched GDB 4.17 which understands Ada.

In 2003, AdaCore released a similarly patched GDB 5.3 which understands Ada; this is the `gnat-gdb` shipped as part of Debian 3.1 "Sarge".

In GCC 3.4, the default format for debugging information changed from STABS to DWARF2. This change breaks compatibility with GDB 5.3; it is possible to force GCC to use the old STABS format by compiling with the '`-gstabs+`' option, like this:

'`gnatmake -gstabs+ my_program`'

As part of the GNAT GPL 2005 Edition (2005-09-15), AdaCore released an updated patch to GDB 5.3 and also a new patch to GDB 6.3, which understands DWARF2 natively and is the `gnat-gdb` in Debian 4.0 "Etch".

AdaCore's patches were eventually merged into GDB. Thus, GDB 6.7 can debug Ada programs out of the box. As a consequence, starting with Debian 5.0 "Lenny", there is no longer a need for a separate `gnat-gdb` package.

2.6 History of Ada standards

All Ada standards are freely available from http://www.adaic.org/ada-resources/standards/.

Ada, as an ISO standard, undergoes at least one review every 5 years. The timeline of past standards is as follows:

1980	MIL-STD-1815
1983	ANSI/MIL-STD-1815A
1987	Ada 83: ISO/IEC 8652:1987
1995	Ada 95: ISO/IEC 8652:1995
2001	Ada 95 Technical Corrigendum 1
2007	Ada 95 Amendment 1 aka Ada 2005
2012	Ada 2012: ISO/IEC 8652:2012

MIL-STD-1815 was numbered after the birth year of Lady Ada Lovelace, namesake of the language.

Ada 2012 became official on 2012-12-15, five days after the 197th birthday of the Lady.

3 Ada and shared libraries

(This chapter is informative.)

This chapter discusses the general concept of shared libraries and shared object names ("sonames") and binary compatibility. Then it introduces the rules peculiar to the Ada programming language about source consistency and how these rules impact how to package Ada programs and libraries in Debian.

3.1 Package and shared object version numbers

This section explains the general concept of sonames, which apply to all programming languages; if you are already familiar with it, you can skip to the next section which details the implications on Ada.

There are two version numbers associated with a library package; the "source" version, and the "shared object" version. They can be the same or they can be different.

The source version is what most users think of as "the version"; it identifies the features of the package; it is the same as the upstream (non-Debian) version, plus a Debian part (the "upload number").

The shared object version is part of the "shared object name", or soname, of the library. The soname mechanism first appeared in Unix System V Release 4 in 1990 and is designed to allow multiple incompatible versions of a shared library to be installed at the same time, for use by different binary programs. By convention, the soname is of the form `library_name`.so.`soversion`; the `soversion` part is the actual shared object version.

Note that applications (as opposed to libraries) don't need shared object versions; they only have a source version.

The mechanism works in five steps:

1. The linker creates the shared library and embeds the soname, as a string, in the header of the shared library file. The soname is not necessarily the same as the name of the file; for example:

```
$ gcc -o libfoo.so.1.2.3 $(object_files) -Wl,-soname,libfoo.so.1
```

In this example, the name of the file contains the full shared object version number while the soname only contains the major part of it. There is no requirement that this be the case.

2. The package containing the shared library installs a symbolic link from the soname to the actual file name:

```
$ ln -s libfoo.so.1.2.3 /usr/lib/libfoo.so.1
```

3. The `-dev` package installs a second symlink to the shared library file; this second symlink does not contain any version information:

```
$ ln -s libfoo.so /usr/lib/libfoo.so.1
```

4. When creating a binary application program, the linker follows the versionless symlink from step 3 to find the shared library, reads the soname, and encodes the soname in the binary program:

```
$ ld -o program -lfoo
$ ldd program
libfoo.so.1 => /usr/lib/libfoo.so.1.2.3
```

5. When the application program starts, the dynamic loader (`/lib/ld.so`) reads from the binary program the sonames of all libraries required and then looks for the shared libraries in its library path.

As explained above, the goal of this mechanism is to allow multiple versions of a library to coexist on a system. To understand how this works, consider the following scenario.

3.1.1 Scenario: multiple versions of a library

Abe, a developer, installs the following packages on his machine:

```
libfoo-dev  (= 1.2.3-1)
libfoo1     (= 1.2.3-1)
libfoo1-dbg (= 1.2.3-1)
```

The shared object version of libfoo is 1, while the source version is 1.2.3-1. Note that the development package name does not contain the shared object version.

Abe uses these packages to write an application program called `books`, which he packages as `books (= 1.0-1)`.

Beth, a customer, installs the `books` package on her system, and the package manager also installs the required library `libfoo1`:

```
libfoo1     (= 1.2.3-1)
books       (= 1.0-1)
```

When she runs the `books` program, the dynamic loader reads the file `/usr/bin/books` which says it depends on the shared library with the soname `libfoo.so.1`; then it finds the symbolic link `/usr/lib/libfoo.so.1`, named after the soname, that points to the actual file containing the shared library, `/usr/lib/libfoo.so.1.2.3`.

Much later, `libfoo` has been upgraded to shared object version 2, source version 2.1-5, and Charlie, another developer, installs these packages on his system:

```
libfoo-dev  (= 2.1-5)
libfoo2     (= 2.1-5)
libfoo2-dbg (= 2.1-5)
```

Charlie then writes the program `chaliesmusic` which he packages.

Beth now installs the new package `chaliesmusic` which depends on `libfoo2`; she now has the following packages installed (the Debian installer keeps all versions of binary libraries that are used by installed packages):

```
libfoo1      (= 1.2.3-1)
libfoo2      (= 2.1-5)
books        (= 1.0-1)
chaliesmusic (= 1.1-3)
```

All is well because the new version of the library coexists peacefully with the old one. When `books` runs, it dynamically links with `/usr/lib/libfoo.so.1`; when `charliesmusic` runs, it dynamically links with `/usr/lib/libfoo.so.2`.

For this scenario to work, the two versions of the library must have different sonames, different package names and different file names; these are all requirements of the Debian Policy for shared library packages; see chapter 8 for full details.

3.1.2 Scenario: Upgrading a library

Suppose Beth has the following packages installed:

```
libfoo1      (= 1.2.3-1)
libfoo2      (= 2.1-5)
books        (= 1.0-1)
chaliesmusic (= 1.1-3)
```

Now the upstream developers of libfoo provide source version 2.2 and Abe, the developer of `books`, provides Beth with an updated version which requires `libfoo2 (= 2.2-1)`. The soname of this new version of libfoo is `libfoo.so.2`; it has not changed since version 2.1 (more on this below).

Beth upgrades her system. Because the package `libfoo1` is no longer used, the package manager deletes it, so she ends up with:

```
libfoo2      (= 2.2-1)
books        (= 1.1-1)
chaliesmusic (= 1.1-3)
```

We know that `books` works well with the new version of libfoo2 because Abe has tested it extensively; however, will `charliesmusic` still work? For this the following conditions must hold:

- The application programming interface (API) of libfoo must be stable between 2.1-5 and 2.2-1. That is, existing subprograms in the library may not change their semantics or parameter profile. The new version may however add new subprograms without breaking its API. If this is true, we say the new version is "binary backward compatible" with the old version.
- The compiler that produced `/usr/lib/libfoo.so.2.2` must assume the same ABI as the one that produced `/usr/lib/libfoo.so.2.1` earlier. The ABI includes the subprogram calling convention (how parameters and return results are passed), name mangling scheme for overloaded subprogams, minimum alignment of objects (1 byte, 2 bytes, 4 bytes), exception propagation mechanism, object file format, etc.
- The program `charliesmusic` may not rely on bugs in `/usr/lib/libfoo.so.2.1` which are corrected in `/usr/lib/libfoo.so.2.2` (this is a particular case of a semantic change in the library).

3.2 Ada Library Information files

The Ada language definition (RM 10.1.4(5)) requires that the compiler ensure consistency between all compilation units in a program in the sense that no out-of-date compilation unit can be part of the final binary executable. A compilation unit becomes out of date when its source text changes or when it depends on a compilation unit that becomes out-of-date. In Debian, the set of all compilation units in a program (the "closure") consists of all the compilation units in the binary executable plus all those in the shared libraries used by the program. Therefore, if a shared library changes, all shared libraries and binary executables depending on it become out of date in the Ada sense.

GNAT implements the requirement for consistency by means of Ada Library Information (`*.ali`) files containing text in a format and syntax particular to GNAT. The format occasionally changes from one major version of GNAT to the next. As part of the normal compilation process, GNAT creates one `*.ali` for each compilation unit. This file contains 32-bit checksums of the source files of the unit and of the entire closure of the unit, calculated over the entire source text of all units, ignoring white space and comments so that a comment-only source file change does not invalidate the corresponding compilation unit or units depending on it.

Note that upgrading the source for a shared library while preserving its soname and API breaks Ada's consistency rule; the old application, which is consistent with the old source, is run with the new library, which is consistent with the new source. However, this breakage is not detected by the binary program, since the dynamic linker is not aware of the Ada requirement.

We have seen above that this is not a problem for customers that use the binary files. However, it is a problem for developers who compile, since they use the `*.ali` files. Consider the following scenario.

3.2.1 Scenario: Indirectly causing a package to FTBFS

Abe, a developer, installs the following packages on his machine:

```
libconfig-dev (= 1.0-1)
libconfig1    (= 1.0-1)
libfoo-dev    (= 1.2.3-1)
```

```
libfoo1       (= 1.2.3-1)
```

`libfoo1` depends on `libconfig1`. Abe uses these packages to write an application program called `books`, which he packages as `books (= 1.0-1)`.

Beth, a customer, installs the following packages on her system:

```
libconfig1  (= 1.0-1)
libfoo1     (= 1.2.3-1)
books       (= 1.0-1)
```

Later, the package maintainer for `libconfig-dev` upgrades to source version 1.1-1; the `*.ali` files change. The library API is the same, so the soname does not change. However, the Debian maintainer of `libfoo-dev` has not yet upgraded it to use the latest libconfig sources.

Beth upgrades her system to:

```
libconfig1  (= 1.1-1)
libfoo1     (= 1.2.3-1)
books       (= 1.0-1)
```

From Beth's point of view, all is well because `books` continues to run, even though the Ada rules say `libconfig1` is not consistent with `books`.

Abe also upgrades his system and ends up with:

```
libconfig-dev (= 1.1-1)
libconfig1    (= 1.1-1)
libfoo-dev    (= 1.2.3-1)
libfoo1       (= 1.2.3-1)
```

Now Abe has a problem when he rebuilds `books` and finds this error message:

```
gnatbind -shared -E -I- -x /tmp/books-1.0/obj/books.ali
> error: "foo.adb" must be compiled
> error: ("/usr/lib/ada/adalib/config/config.ali" is obsolete and read-only)
```

This is because `foo.adb`, which is part of `libfoo-dev`, is now out of date at the source level. This, in turn, is because the Ada package '`Foo`' depends on '`Config`' which changed when `libconfig-dev` changed source version.

Abe could decide that, because the source breakage does not affect Beth, the problem is minor. However, Debian Policy disagrees with him. It says that any package that fails to build from source has a serious (release-critical) bug. Consequently, Daniel, who runs a daemon constantly rebuilding all packages in Debian, soon files such a bug against package `books`. The problem, of course, is not in `books`; it is that `libfoo` has not yet been rebuilt to incorporate the changes in `libconfig`.

The immediate fix for Abe's problem is to recompile `libfoo`, regenerating the `libfoo` `*.ali` files. Note that this then requires recompiling any other libraries that depend on `libfoo`; ultimately, all packages that depend directly or indirectly on `libconfig` must be recompiled, and their source version changed. Because neither the API nor the ABI has changed, any libraries thus rebuilt can retain their sonames.

For the Debian repository, the updated packages should be uploaded in reverse dependency order, possibly by requesting binary-only non-maintainer uploads (also known as "binNMU" in the Debian jargon; see http://wiki.debian.org/binNMU).

The person who should request these binNMUs is the maintainer of `libconfig`, not Abe or Beth. However, it is not generally possible for the maintainer of a library to discover all packages that build-depend on the library because some of these packages might not be in the Debian archive at all; they may be in third-party archives or even hidden inside organizations that use their packages internally and do not publish them at all. Consequently, changing the contents of a library's `*.ali` files has far-reaching consequences that are not all immediately obvious.

3.2.2 Avoiding the indirect FTBFS scenario

We will now discuss possible ways to avoid or mitigate the problem described in the previous subsection.

First we introduce the *aliversion*. This is similar to the *soversion*; it is a version number that changes when the `*.ali` files change.

The *aliversion* of a package must change when the `*.ali` files of the package change, possibly as a result of rebuilding against a changed library. It must *not* change otherwise. When the *aliversion* changes, all packages that depend on the changing package must be recompiled and re-uploaded; we don't want to do that unless strictly necessary.

As with the *soversion*, there are choices about how the *aliversion* relates to the source version. For the purposes of this discussion, there are five reasons a Debian package source version needs to change:

1. The upstream Ada source changes, changing the ali files.
2. The upstream non-Ada source changes, not changing the ali files.
3. The Debian maintainer patches Ada source files, changing the ali files.
4. A library the package depends on changes its `*.ali` files; the Debian maintainer recompiles the package, changing the `*.ali` files of the package. This includes the case of a GNAT compiler version change.
5. The Debian maintainer patches non-Ada source files (this includes Debian packaging files, such as `control` and `rules`), not changing the `*.ali` files.

To handle all of these changes orthogonally, we could use a source version number that contains all of these parts, in some order:

- *upstream-ada*
- *upstream-non-ada*
- *library-ali*
- *debian-ada-patch*
- *debian-non-ada-patch*

However, a source version that actually contains all of these parts is unwieldy, so we look for typical situations that let us simplify it.

If the package is overwhelmingly Ada (lots of Ada source files, very few non-Ada source files), then it will be very unlikely that there will be a release that has only non-Ada changes in it. So we can simply drop the non-Ada parts of the source version.

If the upstream distribution does not include any `*.ali` files (it is a source only distribution), we can combine the *library-ali* and *debian-ada-patch* parts.

In this simplest case, the source version is *upstream-ada-library-ali-and-debian-ada-patch*, where *library-ali-and-debian-ada-patch* is the Debian upload number. In this case, the *aliversion* can be either the full source version (since that only changes when the Ada files do, which is when the `*.ali` files change), or just *library-ali-and-debian-ada-patch*. However, we will see below that a policy of never doing a release without any non-Ada changes is not workable in the long run.

If the package has lots of non-Ada sources, so that a release with non-Ada only changes is likely, and the upstream distribution contains no `*.ali` files, the upstream maintainers are not likely to want to split the source version into Ada and non-Ada parts. In this case the *aliversion* can't be the source version (since the source version changes when `*.ali` files don't); so it can be a simple integer.

If the upstream distribution does contain `*.ali` files, it is likely that the upstream maintainers simply don't want to deal with the consistency issue; without a known packaging system, it's

not easy to deal with. So this reduces to the other two cases. The Debian package maintainer must regenerate the `*.ali` files, not use the upstream versions.

In both cases, the distinction between *upstream-ada* and *upstream-non-ada* is gone, as is the distinction between *debian-ada-patch* and *debian-non-ada-patch*. Also, *library-ali* does not need to be distinct from the other Debian changes. So we are left with this structure for the source version:

upstream-version-debian-upload-number

where *debian-upload-number* changes whenever the `*.ali` files do, and also for any other Debian patches.

In principle, there are two choices for how to incorporate the *aliversion* into Debian packages:

aliversion in package version

This choice is appropriate when every upload (even binNMU) is guaranteed to change the `*.ali` files.

The maintainer of `libconfig` produces the binary package `libconfig-dev (=aliversion)`. The *aliversion* is the source version *upstream-version-debian-upload-number*.

The maintainer of `libfoo` specifies '`Build-Depends: libconfig-dev (=aliversion)`' in `debian/control`. This very tight build-dependency causes `libfoo` to FTBFS as soon as a later version of `libconfig-dev` reaches unstable.

For the purposes of the '`Build-Depends`' clause, it is generally impossible to construct a meaningful range of values wider than this, because a change in any part of the source version means that the `*.ali` files have changed.

This method has the disadvantage that developers using the `-dev` package can forget to include a version restriction in their '`Build-Depends`'. It is not possible to include a rule to enforce this in `lintian`; there is no way to distinguish Ada packages (that require the restriction) from non-Ada packages (that don't).

In addition, the policy of "no releases without Ada changes" is difficult to live with. Most packages get into situations where there needs to be a packaging fix, or a documentation fix; these are non-Ada changes.

aliversion in development package name

This choice is appropriate when there will be releases with only non-Ada changes.

The maintainer of `libconfig` produces the binary package `libconfig<aliversion>-dev`. The *aliversion* cannot be the source version (except in very special circumstances), because they change for different reasons; it can be a simple integer.

The maintainer of `libfoo` specifies '`Build-Depends: libconfig<aliversion>-dev`' in `debian/control`; the version number, or range of version numbers, are optional (they may be required for reasons not related to `*.ali` files).

One advantage of this method is that the maintainer of `libfoo` can no longer forget to specify the *aliversion* of `libconfig` in the '`Build-Depends`', since it is now part of the `libconfig` package name.

Another advantage is that the maintainer of `libconfig` can now upload new versions of the package that do not change the *aliversion*, without breaking build-dependencies of other packages. For example, it is possible to upgrade `libconfig1-dev (=1.0-1)` to `libconfig1-dev (=1.0-2)` without breaking `libfoo`.

Similarly, simple binNMUs for packaging fixes do not change the *aliversion*.

The drawbacks mainly fall on the maintainer of `libconfig` and the Debian release managers, relieving the maintainers of `libfoo` and `books`.

First, it is necessary to detect any changes to the `*.ali` files; such a change requires a change to the *aliversion*. False positives should be avoided; the cost of changing *aliversion* is the recompilation and re-release of all packages using `libconfig`, and we want to minimize that.

Second, every upload that changes the contents of the library's `*.ali` files needs to go through the NEW package queue; the change in package names requires some work by the Debian release managers. This is similar to the burden due to changing the *soversion*.

In summary, it is not likely that a package can maintain a policy of no non-Ada releases, and it is more likely that the *aliversion in development package name* policy will be implemented properly, so we forbid the *aliversion in package version* policy. As a consequence, every change to the *aliversion* requires a new `-dev` package name. This is because, unlike other languages, Ada has consistency rules that apply to the entire closure of a program at the source level (OCaml has similar rules).

To fully avoid the FTBFS problem, multiple versions of a `-dev` package must coexist on a system. However, that is quite complicated to achieve, so we allow both choices:

Coexistence Allowed

If multiple `libconfig<aliversion>-dev` are to coexist on a system, they must avoid file conflicts; we must be able to install and remove each package without affecting any other package's files. This implies that they must install their files in different directories, the names of which should probably include the *aliversion*.

The corresponding run-time library packages must also coexist without conflict. This means that the *soname* must be different for each package, even if the library API has not changed. The *soname* must change whenever the *aliversion* changes, and any change in API will cause a change in *aliversion*. So the *soversion* can be the *aliversion*.

Quite likely, the multiple `libconfig<aliversion>-dev` packages will be built from multiple source packages, so the source package should also have *aliversion* in its name, e.g. `libconfig<aliversion>`.

The maintainer of `libconfig` agrees to maintain the multiple versions of the source, development, and run-time packages in parallel.

Packages that depend directly or indirectly on `libconfig1.0` still build when `libconfig1.1` is released, avoiding the FTBFS problem. They may all be upgraded gradually to the new version, as the maintainers have time for it. They should all be upgraded eventually, and `libconfig1.0` removed, to avoid a proliferation of package versions.

No coexistence allowed

If the maintainer of `libconfig` wants to disallow coexistence, it is necessary that the package `libconfig<aliversion>-dev` have '`Breaks`' and '`Replaces`' clauses in its `debian/control` that list all previous versions of the `-dev` package. The '`Breaks`'/'`Replaces`' relationship can be indirect and implicit through `gnat-X.Y`. If `libconfig1-dev` depends on `gnat-4.1` and `libconfig2-dev` depends on `gnat-4.3`, an explicit '`Breaks:`' is unnecessary because `gnat-4.1` and `gnat-4.3` already conflict with each other, preventing coexistence of `libconfig1-dev` and `libconfig2-dev` as intended.

The *soname* of the shared library can change independently of the *aliversion*.

The maintainer of `libconfig` maintains only one version of the package at a time.

> Installing a new `libconfig<aliversion>-dev` immediately removes any previous version from the system and causes packages build-depending on the old version to FTBFS. There is no grace period for these packages; they must be recompiled immediately. However, Daniel's build daemon will report the correct problem. This policy avoids the *indirect* FTBFS scenario, replacing it with a *direct* FTBFS scenario.

It is up to each library's maintainer to decide which method to choose. The purpose of this policy is to have build failures report the correct problem and to force recompilation of all packages that directly or indirectly build-depend (not merely depend) on an upgraded package, or to avoid the build failures entirely.

The Ada run-time library follows the *Coexistence allowed* policy. The development files (`*.ads`, `*.adb`, `*.ali`, `*.a` and `*.so`) are in a version- and target-specific directory, as specified by the GCC directory structure: `/usr/lib/gcc/VERSION/TARGET`. Since `gnat-4.3`, in fact, *two* versions of the libgnat development files are provided: one using the default zero-cost exception handling mechanism, the other using setjump/longjump. However, the Ada compiler driver, `/usr/bin/gnatmake` and its friends `/usr/bin/gnatbind`, `/usr/bin/gnatlink`, etc. are in a location that causes all `gnat-x.y` packages to conflict with each other.

The run-time packages, `libgnat-x.y`, can coexist on a system.

For the GNAT package the *aliversion* consists of the first two numbers in the source version, e.g. '`4.3`', '`4.4`'. This works for GNAT because there are no Ada libraries that GNAT depends on; this means the *aliversion* changes only when GNAT sources change.

3.2.3 Responsibilities of upstream authors and maintainers

Ada libraries have three version numbers; source, shared object, and ali. It should be clear by now that:

- The upstream portion of the source version is a responsibility of the upstream authors.
- The Debian upload number, which supplements the upstream source version, is a responsibility of the Debian package maintainer.
- The *soname* (which includes the *soversion*) is a responsibility of the Debian package maintainer. This follows from the fact that upstream provides sources, not binaries, while the Debian package maintainer provides the binaries. If the upstream distribution does provide binaries, the Debian package does not use them, since they are almost certainly not built with other Debian packages.
- The *aliversion* is also a responsibility of the Debian package maintainer.

The upgrade scenario illustrates precisely that the *soname* of a library must change whenever the new version of the library changes its API. It also illustrates the importance of tight control on the *soname*. The *soname* provided by the library package maintainer has an impact on both developers linking their applications against the library and on end users running the programs.

It is not always practical to determine with certainty whether a new version of a library breaks its API or not.

It is always safe to increment a library shared object version, if there is any doubt about backward compatibility. However, there is a cost to this; the Debian build system must accomodate the new package name, which requires manual labor by the Debian release managers. In addition, users of the package must examine the changes, to see if their code needs to change.

The Debian package maintainer must, therefore, change the *soname* of libraries as often as necessary and as seldom as possible.

Similarly, the *aliversion* must change whenever the Ada sources of the package change or when a package it depends on changes its *aliversion*, and it should not change otherwise.

The Debian maintainer can use one of several policies for choosing the *aliversion* and *soversion*.

The simplest policy that always works is to make *aliversion* an integer that increases only when `*.ali` files change. This integer is entirely artificial as it is not derived from the upstream source version number.

In the *No coexistence allowed* policy, *soversion* can similarly be an integer that changes only when the API changes.

In the *Coexistence allowed* policy, however, it must additionally change whenever the *aliversion* changes, so the simplest convention is to make it identical to *aliversion*.

The Debian maintainer is free to use another policy that meets all the requirements. For example, it is sometimes (not always) possible to derive the *aliversion* and *soversion* from the source version number.

4 The Debian Ada compiler

(This chapter is normative.)

Rule: Package `gnat` is the default Ada compiler for Debian. All packages containing Ada programs or libraries SHALL use this compiler and Build-Depend on it. As a special exception, packages uploaded to experimental may omit this dependency.

Rationale: This forces all programs and libraries to use the same ABI, so we can link them together.

Rationale: the special exception allows package maintainers to upload packages to experimental ahead of a transition of the `gnat` package to a newer `gnat-X.Y`. Such packages must be re-uploaded to unstable, with a Build-Dependency on `gnat`, to complete the transition.

Remark: `gnat-x.y` also provides the `gnatgcc` command, which shall be used for C code instead of `gcc` when they differ.

Rule: Package `gnat-X.Y` provides a specific version of the Ada compiler. Packages containing Ada libraries SHALL in addition Build-Depend on the specific `gnat-X.Y` used at build time.

Rationale: Specifying both the actual version and the intended global compatibility ensures an explicit build failure when both wishes are incompatible.

Additional information: The default Ada compiler for Debian is as follows. Some versions of Debian provide alternative versions of GNAT. These alternative versions are for experimental purposes only; they receive no support and Debian packages may not build-depend on any of them.

Debian release	**Default Ada compiler**	**Alternatives**	**Supported Platforms**
2.0 Ham	GNAT 3.07		i386
2.1 Slink	GNAT 3.10p		i386
2.2 Potato	GNAT 3.12p		i386
3.0 Woody	GNAT 3.14p		i386, powerpc, sparc
3.1 Sarge	GNAT 3.15p	gnat-3.3, gnat-3.4	i386, powerpc, sparc
4.0 Etch	GCC 4.1		+amd64, hppa, hppa64, ia64, kfreebsd-i386, ppc64, sparc64
5.0 Lenny	GCC 4.3		+alpha, mips, mipsel
6.0 Squeeze	GCC 4.4		+arm, armel, kfreebsd-amd64
7.0 Wheezy	GCC 4.6		+m68k, hurd-i386, multiarch
8.0 Jessie	GCC 4.9		-s390, +s390x, -sparc

5 Policy for Ada libraries

The Policy for Ada libraries is mostly interesting if you want to package libraries.

If you only want to use libraries, see Section 6.3 [Using shared libraries], page 26.

The goal of this policy is to make Debian a robust and complete development platform for Ada programmers. This platform should appeal both to seasoned programmers and beginners. The basic tenet is that it Just Works. At the same time, the Policy makes Debian an equally attractive deployment platform for users of Ada programs. Thanks to Debian's outstanding package management system, users simply '`apt-get install`' watever package they want to use and `apt-get` installs all the required run-time libraries. The Policy ensures there are no conflicts between libraries.

This policy is based on the GNU Ada Environment Specification as far as is reasonable. This specification is an attempt by Florian Weimer to make all libraries install files in consistent places.

The GNAE is distribution-agnostic; it mandates a mechanism (similar to GNOME's pkg-config) to retrieve compiler switches for each library in a uniform way. However, thanks to GNAT project files, this is unnecessary; and Debian's packaging system handles dependencies. So, what follows is a stripped-down version of the GNAE suitable for Debian.

5.1 Building a library for reuse

(This section is informative.)

For some libraries, the upstream authors provide a `Makefile` that does not compile all of the source files of the library; in particular, some upstream `Makefile`s do not compile all generic bodies. These upstream authors assume one of the following:

- The library user has permission to write into the library's directory structure and uses GNAT project files or `gnatmake -i`.
- The library user compiles the library's source code into their application, using `gnatmake -I`.
- The library user copies the sources of the library into their program's directory structure.

However, in Debian, none of the above applies. If the Debian maintainer were to use such `Makefile`s and installed the `*.ali` files in `/usr/lib/ada/adalib/LIBRARY`, then some `*.ali` files would be missing. This would be a problem for non-root users, because they are not allowed to write `*.ali` files in `/usr/lib/ada/adalib/LIBRARY`. To prevent this, it is sometimes necessary for the Debian package maintainer to bypass upstream's `Makefile` with a scheme that ensures that all files are compiled. Here is a sample from `debian/build_library.gpr` that demonstrates how to do this:

```
project LIBRARY is
   for Library_Name use "LIBRARY";
   for Library_Kind use External ("LIBRARY_KIND");
   for Library_Version use External ("SONAME");
   for Source_Dirs use (".");
   for Object_Dir use External ("OBJ_DIR");
   package Compiler is
      for Default_Switches ("Ada") use
        ("-g", "-O2", "-gnatafnoy", "-gnatVa", "-gnatwa", "-fstack-check");
   end Compiler;
   package Binder is
      for Default_Switches ("Ada") use ("-E");
   end Binder;
```

```
end LIBRARY;
```

A Makefile would call this project file like so:

```
# change the soversion manually
soversion:=1
soname:=libLIBRARY.so.$(soversion)

obj-shared/libLIBRARY.so:
        gnatmake -p -Pbuild_LIBRARY.gpr \
                 -XLIBRARY_KIND=dynamic -XOBJ_DIR=$(dir $@@) -XSONAME=$(soname)

obj-static/libLIBRARY.a:
        gnatmake -p -Pbuild_LIBRARY.gpr \
                 -XLIBRARY_KIND=static -XOBJ_DIR=$(dir $@@)
```

This scheme effectively replaces upstream's `Makefile` altogether, modulo any configuration or preprocessing steps that may be necessary. I would personally advise upstream library authors to consider using such a scheme in their `Makefile`s.

5.2 Library names and packaging structure

(This section is normative.)

Rule: The package maintainer SHALL choose either of the *Coexistence allowed* or *No coexistence allowed* policies; no other choices are allowed.

Rule: The package maintainer SHALL change the *aliversion* if and only if the contents of the `*.ali` files change.

Additional information: `*.ali` files can change not only as a result of changes in the Ada source files but also as a result of recompiling the library against a new version of another library. This includes recompiling with a new compiler against a new version of the Ada run-time library.

5.2.1 Coexistence allowed

Rule: If the package maintainer chooses the *Coexistence allowed* policy, the *soname* of the shared library SHALL change if and only if the contents of the `*.ali` files change.

Additional information: the `*.ali` files will change when the API does, so this includes the normal reason for changing the *soname*. The *aliversion* should be the *soversion*, e.g. soname `libfoo.so.1` and development package `libfoo1-dev`.

Additional information: if each version of the library is built by a different source package, the source package name should contain the *aliversion* as well.

5.2.2 No coexistence allowed

Rule: If the package maintainer chooses the *No coexistence allowed* policy, the *soname* of the shared library SHALL change every time the API of the library changes in a backward-incompatible way, and not otherwise.

Additional information: the *aliversion* should not be the *soversion*.

Additional information: in this case, it is a good idea that the source package name not contain any version number.

5.2.3 Package names

Rule: Each Ada library SHALL consist of the following packages:

'`libLIBRARY[-]V-dev`'
: the development package containing the static libraries and development files, in section `libdevel`.

‘`libLIBRARY[-]N`’
: the run-time package containing the shared library, in section `libs`.

‘`libLIBRARY[[-]V]-doc`’
: the documentation (optional), in section `doc`.

‘`libLIBRARY[[-]N]-dbg`’
: the separate debugging information for the shared library, in section `debug`.

where:

V
: is the *aliversion*.

  It is present in the name of the documentation package if the *Coexistence allowed* policy is chosen.

N
: is the *soversion*.

  It is present in the name of the debug package if the *Coexistence allowed* policy is chosen.

-
: is used when the library name ends in a number.

5.2.4 Inter-package dependencies

Rule: The `-dev` package SHALL ‘`Depend:`’ on the packages `gnat-X.Y` and, if uploaded to unstable, `gnat`.

Rationale: depending on both packages ensures that only the default Ada compiler can be used to build programs depending on the library, pursuant to Chapter 4 [The Debian Ada compiler], page 17. For example, during `gnat` transitions, all `-dev` packages will blocked in unstable until they are recompiled with the new compiler. Additionnaly, this rule makes the `-dev` package visible in the list of packages that depend on `gnat`; people looking for an Ada compiler in Debian will therefore find all `-dev` packages easily.

Rationale: during transitions, it may be desirable to upload packages to experimental before a new `gnat` package is available in unstable.

Rule: If the package maintainer chooses the *No coexistence allowed* policy, the `-dev` package's `debian/control` file SHALL contain ‘`Breaks:`’ and ‘`Replaces:`’ lines listing all previous versions of the package, if any, that ever depended on the same `gnat-X.Y` package.

Rule: the `-dbg` package SHALL Depend: on the exact version of the corresponding shared library package.

Rationale: the `-dbg` package is useless without the shared library package.

Rule: the `-doc`, `-dbg` and any other packages produced from the same source package, SHALL Suggest: the package `gnat`.

Rationale: this makes all the packages visible in aptitude and other package managers; the user can simply browse the “Package which depend on” `gnat`, look under `Suggests` and get a list of all Ada packages available in Debian.

5.3 Files provided by the -dev package

(This section is normative.)

5.3.1 General rules

Rule: The `-dev` package SHALL NOT provide any other files than the ones described in this section.

Rule: The `-dev` package SHALL be architecture-dependent (i.e. `Arch: any` or a specific list of architectures).

Rule: The `-dev` package SHALL depend on the `gnat-x.y` and on any other `-dev` packages with which it was built.

Rule: If the package maintainer chooses the *Coexistence allowed* policy, for purposes of all the following rules in this section, the term *LIBRARY* SHALL include the same version number as the name of the `-dev` package.

Additional information: for example, if the development package name is `libfoo1-dev` then the term *LIBRARY* shall stand for '`foo1`'.

Rule: If the package maintainer chooses the *No coexistence allowed* policy, the term *LIBRARY* shall not contain any version number.

Remark: Paths given from now on in this manual refer to the `DEB_HOST_MULTIARCH` value. If the maintainer is interested in preparing the multiarch migration, `DEB_HOST_MULTIARCH` must be replaced with the value returned by a call to `dpkg-buildflags -qDEB_HOST_MULTIARCH` or set by including the `/usr/share/dpkg/default.mk` makefile snippet, both from the `dpkg-dev` package. Else, `DEB_HOST_MULTIARCH` may be ignored.

5.3.2 Source files

Rule: The `-dev` package SHALL provide all source files (specs and bodies) necessary for compilation of code that uses the library.

Rule: Source files SHALL reside in directory `/usr/include/ada/adainclude/LIBRARY`.

Recommendation: There should not be any subdirectories under `/usr/share/ada/adainclude/LIBRARY`. If the upstream authors split the sources into several subdirectories, merge all source files into one directory or else, provide several separate library packages. If you insist on providing subdirectories, make sure your project file (described below) reflects this.

Rationale: Splitting the sources in several directories makes navigation in the library source code more difficult in some editors or debuggers.

Rule: If the upstream sources require preprocessing the Ada source files before compilation, then the Ada files in `/usr/share/ada/adainclude/LIBRARY` SHALL be the preprocessed files corresponding to the shared library in `/usr/lib/DEB_HOST_MULTIARCH`.

Rule: The directory with all Ada source files SHALL not contain any `Makefiles` or any other files that might be necessary to recompile the library.

Rationale: The intention is not that the user can recompile the library from `/usr/share/ada/adainclude/LIBRARY`; only that they can use it in their Ada programs. They can always recompile the library from the source package.

Rule: The `-dev` package SHALL, however, provide any source files in languages other than Ada (e.g. C) that are compiled into the library.

Rationale: This allows programmers using GDB to trace the execution of their programs into the library. See below Section 5.6 [The separate debugging information package], page 24.

5.3.3 Ada object files

Rule: The `-dev` package SHALL NOT provide any `*.o` files.

Rule: The `-dev` package SHALL provide a static library in `/usr/lib/DEB_HOST_MULTIARCH/libLIBRARY.a`.

5.3.4 Ada Library Information files

Rule: The `-dev` package SHALL provide Ada Library Information (`*.ali`) files that `gnatgcc` creates when compiling the shared (not static) library.

Rule: The `*.ali` files SHALL have read-only permissions for all users (i.e. `r--r--r--` or 0444).

Rationale: During compilation of a program against the library as described later in this document, object files are missing because they are lumped together into the shared and static libraries. Gnatmake understands the `Externally_Built` attribute in library project files but with `ADA_INCLUDE_PATH` and `-aI`, it insists on recompiling every object whose ALI file has permissions different from `r--r--r--`.

Rule: The `*.ali` files SHALL reside in `/usr/lib/DEB_HOST_MULTIARCH/ada/adalib/LIBRARY`.

Additional information: Starting with Etch, `lintian`, `Linda` and `dh_fixperms` are aware of these rules and enforce them.

5.3.5 Symbolic link to the shared library

Rule: The `-dev` package SHALL provide a symbolic link to the shared library, as follows:

```
/usr/lib/DEB_HOST_MULTIARCH/libLIBRARY.so -> library_file_name
```

where *library_file_name* is the full name of the shared library file. For example:

```
/usr/lib/DEB_HOST_MULTIARCH/libtexttools.so -> libtexttools.so.2.0.3
```

Additional information: the *library_file_name* is, in theory, independent from the *soname*. However, traditionally it is either equal to the *soname* or is the *soname* followed by one or more minor version numbers. If simplicity is a goal, the package maintainer can always use a *library_file_name* equal to the *soname*.

Rule: The `-dev` package SHALL depend on the run-time package.

5.3.6 GNAT project file

Rule: The `-dev` package for each library SHALL provide a GNAT project file named `/usr/share/ada/adainclude/LIBRARY.gpr`.

Rule: The project file SHALL have

- a `Source_Dirs` attribute containing at least `/usr/share/ada/adainclude/LIBRARY`.
- a `Library_ALI_Dir` equal to `/usr/lib/DEB_HOST_MULTIARCH/ada/adalib/LIBRARY`.
- a `Library_Name` attribute equal to the library name of the shared library.
- a `Library_Kind` attribute equal to '`dynamic`'.
- a `Library_Dir` attribute equal to `/usr/lib/DEB_HOST_MULTIARCH`.
- an `Externally_Built` attribute equal to '`true`'.

Remark: the presence of `Library_Name` and `Library_Dir` makes the project file a "library" project file which gnatmake treats specially. See the GNAT documentation for details.

Remark: Since the `-dev` and library packages provide precompiled libraries, it is not necessary to provide '`package Compiler`' or '`package Builder`' in the project file. However, if the library depends on another dynamic library, every program linked with the former also indirectly depends on the latter. A '`package Linker`' must then list the implied options.

Example:

```
project LIBRARY is
   for Library_Name use "LIBRARY";
   for Library_Dir use "/usr/lib/DEB_HOST_MULTIARCH";
   for Library_Kind use "dynamic";
   for Source_Dirs use ("/usr/share/ada/adainclude/LIBRARY");
   for Library_ALI_Dir use "/usr/lib/DEB_HOST_MULTIARCH/ada/adalib/LIBRARY";
   for Externally_Built use "true";
   package Linker is
      for Linker_Options use ("-lindirectdependency");
   end Linker;
end LIBRARY;
```

5.3.7 Documentation

Rule: The `-dev` package SHALL have a `README.Debian` that explains how to use the library in new programs.

Additional information: This file is not mandatory per Debian Policy, but this Debian Ada Policy does require it. See Section 6.3 [Using shared libraries], page 26 for details.

5.4 Files provided by the run-time shared library package

Recommendation: If `DEB_HOST_MULTIARCH` is used, the library package should be declared `Multiarch: same` and `Pre-Depend: ${misc:Pre-Depends}` in `debian/control`.

Rule: If the *soname* of the shared library is identical to its *library_file_name*, the run-time shared library package (*libLIBRARYN*) SHALL provide this file:

```
/usr/lib/DEB_HOST_MULTIARCH/soname
```

Rule: If *soname* and *library_file_name* are different, the run-time shared library package (*libLIBRARYN*) SHALL provide one file and one symbolic link:

```
/usr/lib/DEB_HOST_MULTIARCH/library_file_name
/usr/lib/DEB_HOST_MULTIARCH/soname -> library_file_name
```

Additional information: for example,

```
/usr/lib/DEB_HOST_MULTIARCH/libtexttools.so.2.0.3
/usr/lib/DEB_HOST_MULTIARCH/libtexttools.so.2 -> libtexttools.so.2.0.3

$ objdump -p /usr/lib/DEB_HOST_MULTIARCH/libtexttools.so.2.0.3 | grep SONAME
  SONAME      libtexttools.so.2
```

Additional information: it is acceptable to provide several shared libraries in the same package if and only if these shared libraries depend on each other and are built from the same source package. The package maintainer can do this if most, if not all, users of one shared library are likely to also need the other libraries. This, of course, implies that changing the *soname* of one library requires changing the *soname* of all libraries.

5.5 Documentation

(this section is normative.)

Rule: Documentation other than the `README.Debian` file SHALL be included either in the `-doc` package or in the `-dev` package.

Rationale: If the amount of documentation is small or if the documentation consists mostly of comments in the Ada source files, it is preferable to place it in the `-dev` package. The `README.Debian` file must always be in the `-dev` package, as explained above.

Remark: You may consider using `adabrowse` to generate HTML documentation from the source files.

5.5.1 Documents in the info format

Rule: If the library provides Info files, then the Info files SHALL appear under “GNU Ada tools” in the top-level Info directory.

Additional information: This usually requires adding a few lines near the top of the first Info file, before the first node. Here is an example from package `libaws-doc`:

```
INFO-DIR-SECTION GNU Ada tools
START-INFO-DIR-ENTRY
* AWS: (aws).    The Ada Web Server.
END-INFO-DIR-ENTRY
```

If you generate **.info* files from **.texi* files using makeinfo, the following lines achieve the desired effect:

```
@dircategory GNU Ada tools
@direntry
* AWS: (aws).    The Ada Web Server.
@end direntry
```

Based on this information, `dh_installinfo` takes care of updating the top-level Info directory.

5.5.2 Examples

Rule: If the library provides example source files, they SHALL reside under `/usr/share/doc/LIBRARY-doc/examples` or, if there is no `-doc` package, in `/usr/share/doc/LIBRARY-dev/examples`.

Rule: Neither the source files, nor any `Makefile` present in the examples, SHALL be compressed.

Additional information: Be careful if you use `dh_installexamples`, as it compresses any files larger than 4 kiB by default — use `-X.ads -X.adb -XMakefile` as appropriate.

5.6 The separate debugging information package

(this section is normative.)

Rule: the package maintainer SHALL provide a package containing the debugging symbols for the shared library.

Rationale: Debian has a mechanism for producing and using `-dbg` packages containing debugging information. For this to work, you must compile the shared library with '`-g`' and the `-dbg` package must appear in `debian/control`. Then, you call '`dh_strip -plibLIBRARYN --dbg-package=libLIBRARY-dbg`' from within `debian/rules`. Since that's all there is to it, package maintainers have little excuse not to provide a `-dbg` package.

Recommendation: If `DEB_HOST_MULTIARCH` has been used, the `-dbg` package should be declared `Multiarch: same`.

6 Debian for Ada programmers

(This chapter is informative)

This Debian Policy for Ada makes Debian attractive to beginners, professionals and teachers alike. This appendix is for people using Debian to develop software in Ada, as opposed to package maintainers.

6.1 Installing an Ada development environment

Packages that comply with this Policy leverage the renowned Debian package management tools to make it easy for a user to discover and install all Ada packages available. Here is a simple recipe for people who wish to install a complete Ada development environment, consisting of many packages, on their Debian machine.

- On a command line, launch `aptitude` which is the officially blessed tool to manage Debian packages.
- Type `/^gnat$` to find the package `gnat`.
- Press ENTER to display the details of the package.
- Scroll down to `Packages which depend on gnat`. Press `[` on that line to expand all levels under it.
- Review the list of packages that `Depend` on `gnat`: these are the `-dev` packages for each library available. The packages that `Recommend` or `Suggest gnat` are additional packages that complement the development environment.
- As you scroll on that list, the status bar at the bottom of the screen displays a short description of each package.
- Type `+` on each package that you wish to install.
- When you are done, press `g`. You will get a list of all actions pending. Press `g` to launch these actions, i.e. to download and install the packages you selected.

The `aptitude` GUI is based on ncurses. If you are more comfortable with a Gtk interface, `synaptic` provides the same capabilities. To search for packages that depend on `gnat`, click the "search" button, enter "gnat", and choose "dependencies" from the "look in" list. Click on the check box next to a package to select it for installation; click on the "apply" button to perform the selected installations.

6.2 In-place upgrades

This Debian Policy for Ada respects the Ada language rule that any executable program can only contain object files that are consistent with their sources and with one another. The section Section 5.2 [Library names and packaging structure], page 19 mandates that, whenever any of the `.ali` files in a `-dev` package change, the name of the package must also change. An unfortunate consequence of this rule is that upgrades are a little bit more problematic than for lesser languages that do not enforce consistency. Here is a recipe for in-place system upgrades. It assumes use of the `aptitude` ncurses GUI. It is *not* a good idea to attempt this with `synaptic`; however, command line `aptitude` is a viable alternative.

- Edit your `/etc/apt/sources.list` to refer to a newer version of Debian.
- On a command line, launch `aptitude` which is the officially blessed tool to manage Debian packages.
- In the full-screen interface of `aptitude`, type `u` to read the new package lists; this may take some time to download the new package lists, which can be quite large.
- After `aptitude` shows the new list of packages, scroll down to `Obsolete or locally created packages` and press `[` to expand this list.

- You will now see all the `-dev` packages that have disappeared from Debian. Press `_` to purge each of them.
- Repeat the steps you performed when installing your Ada development environment, i.e. browse the list of packages that `Depend` on, `Recommend` or `Suggest` the package `gnat`.
- From one stable release of Debian to the next, chances are very high that the list of Ada packages available will grow. New tools and libraries will appear. Conversely, it is possible that some libraries disappear, being superseded by others. For example, `libcharles0-dev` which was present in Debian 3.1 "Sarge" was superseded by the standard Ada.Containers library provided by `gnat-4.1` in Debian 4.0 "Etch".

6.3 Using shared libraries

As a consequence of the Debian policy for Ada libraries, users of the `-dev` packages have three consistent ways of using libraries in their programs:

```
ADA_INCLUDE_PATH += :/usr/share/ada/adainclude/LIBRARY
ADA_OBJECTS_PATH += :/usr/lib/TARGET/ada/adalib/LIBRARY
gnatmake PROGRAM -largs -lLIBRARY
```

or

```
gnatmake -aI/usr/share/ada/adainclude/LIBRARY \
  -aO/usr/lib/TARGET/ada/adalib/LIBRARY \
  PROGRAM \
  -largs -lLIBRARY
```

or

write their own project file and:

```
with "LIBRARY";
project PROGRAM is
   for Source_Dirs use (".");
   for Object_Dir use "obj";
   for Exec_Dir use ".";
   for Main use ("PROGRAM");
end;
```

Depending on the state of the multiarch migration, `TARGET` may be ignored, or replaced with the multiarch triplet describing the binary architecture the user is compiling for. In most cases, the multiarch triplet may be found with:

```
gnatgcc -v 2>&1 | awk '/Target:/ { print $2; }'
```

Starting with Debian 4.0 Etch, GNAT looks for project files in `/usr/share/ada/adainclude` by default. In 3.1 Sarge, it was necessary to specify the full path to the project file in the '`with`' clause or set '`ADA_PROJECT_PATH`' to `/usr/share/ada/adainclude` in the environment.

Also, the '`.gpr`' filename extension is optional in the '`with`' clause.

Some upstream library authors provide scripts (e.g. '`/usr/bin/gtkada-config`' for GtkAda) but they tend not to work well when using several libraries simultaneously because the order of the switches is important. The recommended way to use libraries is the project files, especially when using several libraries at the same time:

```
with "LIBRARY_ONE";
with "LIBRARY_TWO";
project PROGRAM is
   for Source_Dirs use (".");
   for Object_Dir use "obj";
   for Exec_Dir use ".";
```

```
   for Main use ("PROGRAM");
end;
```

The user may also wish to use the static library. Doing this is straightforward:

```
with "LIBRARY";
project PROGRAM is
   Target := "TARGET";
   for Source_Dirs use (".");
   for Object_Dir use "obj";
   for Exec_Dir use ".";
   for Main use ("PROGRAM");
   package Linker is
      for Default_Switches ("Ada") use ("/usr/lib/" & Target & "/libLIBRARY.a");
   end Linker;
end;
```

The user may also wish to recompile parts of the library into their program, possibly with different compile-time options. This is also possible with GNAT project files but without the library-supplied project file:

```
project PROGRAM is
   for Source_Dirs use (".", "/usr/share/ada/adainclude/LIBRARY");
   for Object_Dir use "obj";
   for Main use ("PROGRAM");
   package Compiler is
      for Default_Switches ("Ada") use
         ("-gnatafoy", "-O3", "-fno-unroll-loops", "-gnatVa", "-gnatwa");
   end Compiler;
end PROGRAM;
```

The above examples can be mixed to suit the user's requirements. For example, one library can be linked statically, another can be compiled-in and a third can be linked dynamically.

6.4 Debugging programs that use shared libraries

If the package maintainer provides a `-dbg` package, you simply '`apt-get install`' it and launch `gdb` on your program (preferably inside an integrated development environment). `gdb` automagically looks for the debugging symbols for the library in the proper place and will see them with no further help. However, it may not see the source files automatically. If `gdb` fails to find the source files, you need to say:

```
(gdb) dir /usr/share/ada/adainclude/LIBRARY
```

6.5 Where to ask for help

The Ada community and its Debian sub-community are both very active and friendly. They welcome newcomers and often provide expert advice to those who can formulate their questions precisely.

If your question is about the Ada language in general, the best place is the Usenet newsgroup `comp.lang.ada` and its French-speaking counterpart, `fr.comp.lang.ada`. Experienced users and even compiler writers lurk there, so there is no better place for help short of a support contract with a vendor.

If your question is more specific to Ada in Debian or a particular Debian package, the best place is the mailing list `debian-ada@lists.debian.org`, where all Debian Ada package maintainers lurk as well as many of your fellow Debian Ada programmers. You can browse the archives of this list at http://lists.debian.org/debian-ada.

7 Help wanted

(This chapter is informative)

Help is wanted to further Ada, Debian and Ada in Debian. There are several areas where you can help: as an upstream author or as a packager.

If you would like to help or are simply interested in discussions, the first thing you should do is browse the archives of the debian-ada mailing list and announce your intentions in a mail to debian-ada@lists.debian.org.

As an upstream author, you can contribute to any of the packages already in Debian. You can also write new Ada libraries and programs.

As a packager, you can select existing libraries or software programs, and package them for Debian. You should follow the Debian policy for Ada if you do this.

If you are looking for a project to which you can contribute or for something to package for Debian, here are a few web sites where you can look:

- Browse the main Ada portals: the Ada Information Clearinghouse, Ada Programming Wikibook
- Look for Ada projects on SourceForge, Berlios and Tigris

Appendix A Bugs in GNAT

(This appendix is informative)

Since I started maitaining GNAT, I have received over a hundred bug reports against the compiler. In most of these bugs, GNAT is too lax; it does not complain when presented with an illegal program. There are some quite subtle cases involved; processing these bugs has been a very good learning experience for me. You may be interested in perusing these bugs for your own benefit.

My processing of these bug reports has been, I believe, quite thorough. For each bug:

- I wrote a small test case that demonstrates the bug (most bug reports came with a test case already; this was not much work).
- I ran the test case against GNAT 3.15p and 3.4.
- When GNAT 3.15p had the bug, I reported it to the Debian bug tracking system.
- When GCC 3.4 had the bug, I reported it to the GCC Bugzilla database.
- When both versions had the bug, I linked the Debian bug to the GCC bug.

As it turns out, GCC 3.4 has approximately 80% of the bugs of GNAT 3.15p. It fixes some bugs but also adds a few new ones.

When possible, I try to track the changes that fix bugs in GCC and backport them to GNAT 3.15p. Thus, GNAT 3.15p is under active maintenance in Debian. If you would like to contribute, feel free to send me patches, I will accept them gratefully.

AdaCore also fix bugs that I report in each major release of GCC.

The test suite is available in the GNAT source package; to get the test suite, just `apt-get source gnat` and look in the `testsuite` directory.

You can search the Debian bug tracking system on the web here: `http://bugs.debian.org`

You can search the GCC bug database here: `http://gcc.gnu.org`.

Appendix B A note about GNAT's shared libraries

(This appendix is informative)

I've done some extensive research on several GNU/Linux distributions, and even FreeBSD, to see how they handled libgnat's soname. Most distros never included the AdaCore public version of GNAT and started shipping Ada only after GCC 3.2 went out.

Variant	Version	Distribution	Shared libraries	soname
AdaCore	3.07	binary tgz	/usr/lib/libgnat.so.3.07	libgnat.so
AdaCore	3.09	binary tgz	/usr/lib/libgnat.so.3.09	libgnat.so
AdaCore	3.12p	Red Hat 7.0	/usr/lib/libgnat-3.12p.so.1.12	libgnat-3.12p.so.1
AdaCore	3.13p	Ada for Linux	/usr/lib/libgnat-3.13p.so.1.8	libgnat-3.13p.so.1
AdaCore	3.13p	Red Hat 7.1	/usr/lib/libgnat-3.13p.so.1.13	libgnat-3.13p.so.1
AdaCore	3.13p	SuSE 8.0	/usr/lib/libgnat-3.13p.so.1.7	libgnat-3.13p.so.1
AdaCore	3.14p	Debian Woody	/usr/lib/libgnat-3.14p.so.1.1	libgnat-3.14p.so.1
AdaCore	3.14p	FreeBSD 4.7	/usr/local/lib/libgnat-3.14.so.1	libgnat-3.14.so.1
AdaCore	3.15p	binary tgz	<prefix>/lib/gcc-lib/.../libgnat-3.15.so	libgnat-3.15.so
AdaCore	3.15p	from source	<prefix>/lib/gcc-lib/.../libgnat-3.15.so	libgnat-3.15.so
AdaCore	3.15p	Debian Sarge	/usr/lib/libgnat-3.15p.so.1.8	libgnat-3.15p.so.1
AdaCore	3.15p	FreeBSD 4.8	/usr/local/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.2	Mandrake 9.0	/usr/lib/libgnat-3.15a.so.1	libgnat-3.15a.so.1
FSF	3.2	Red Hat 8	/usr/lib/libgnat-3.15a.so.1	libgnat-3.15a.so.1
FSF	3.2	SuSE 8.1	/usr/lib/libgnat-3.15a.so	libgnat-3.15a.so
FSF	3.2.2	ASPLinux	/usr/lib/libgnat-3.15a.so.1	libgnat-3.15a.so.1
FSF	3.2.2	Mandrake 9.1	/usr/lib/libgnat-3.15a.so.1	libgnat-3.15a.so.1
FSF	3.2.2	Red Hat 9	/usr/lib/libgnat-3.15a.so.1	libgnat-3.15a.so.1
FSF	3.2.2	Slackware 9.0	none	none
FSF	3.2.2	Yellow Dog	/usr/lib/libgnat-.so.1	libgnat-.so.1
FSF	3.2.3	Slackware 9.1	none	none
FSF	3.3	Polish Linux	/usr/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.3	SuSE 8.2	/usr/lib/libgnat-3.15.so	libgnat-3.15.so
FSF	3.3	from source	<prefix>/lib/gcc-lib/.../libgnat-3.15.so	libgnat-3.15.so
FSF	3.3.1	Mandrake 9.2	/usr/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.3.1	SuSE 9.0	/usr/lib/libgnat-3.15.so	libgnat-3.15.so
FSF	3.3.2	Fedora Core 1	/usr/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.3.2	Mandrake 10.0	/usr/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.3.3	Fedora Core 2	/usr/lib/libgnat-3.15.so.1	libgnat-3.15.so.1
FSF	3.3.3	SuSE 9.1	/usr/lib/libgnat-3.15.so	libgnat-3.15.so
FSF	3.3.4	SuSE 9.2	/usr/lib/libgnat-3.15.so	libgnat-3.15.so
FSF	3.3.5	SuSE 9.3	/usr/lib/libgnat-3.15.so	libgnat-3.15.so
FSF	3.3.6	Debian Sarge	none	none
FSF	3.4	source CVS	<prefix>/lib/gcc-lib/.../libgnat-3.4.so	libgnat-3.4.so
FSF	3.4.1	Mandrake 10.1	/usr/lib/libgnat-3.4.so.1	libgnat-3.4.so.1
FSF	3.4.2	Fedora Core 3	/usr/lib/libgnat-3.4.so.1	libgnat-3.4.so.1
FSF	3.4.4	Debian Sarge	/usr/lib/libgnat-3.4.so.1	libgnat-3.4.so.1
FSF	4.0.0	from source	<prefix>/lib/gcc/.../libgnat-4.0.so	libgnat-4.0.so
FSF	4.0.0	Fedora Core 4	/usr/lib/libgnat-4.0.so.1	libgnat-4.0.so.1
FSF	4.1.0	from source	<prefix>/lib/gcc/.../libgnat-4.1.so	libgnat-4.1.so
FSF	4.1.0	Debian Etch	/usr/lib/libgnat-4.1.so.1	libgnat-4.1.so.1
FSF	4.3.0	Debian Lenny	/usr/lib/libgnat-4.3.so.1	libgnat-4.3.so.1

The ellipsis (...) stands for <target>/<version>/adalib, for example `i486-linux-gnu/4.1.0/adalib`.

All GNU/Linux distributions have historically ignored the AdaCore default soname and used one consistent pattern, libgnat-x.xxp.so.1. This includes Debian Woody. For Sarge and Etch, I have chosen to continue this tradition. This allows, by the way, installing both Debian's and AdaCore's binary distributions on the same system.

Appendix C Ada transition in Debian 4.0 Etch

(This appendix is informative)

The compiler and all Ada packages transitioned from gnat 3.15p in Sarge to gnat-4.1 in Etch. This appendix is here for historical purposes.

C.1 How the Ada compiler for Etch was chosen

In 2004, even before Sarge was released I had laid out a plan for Etch and published it in the First Edition of this Policy. The plan was to choose a new Ada compiler for Etch and transition all packages to it. The new Ada compiler would be either the next "p" release from AdaCore, which was the preferred solution or the latest release of GCC available roughly 3 months before the freeze of Etch. These 3 months would be used to actually perform the transition.

When AdaCore released GNAT GPL 2005 Edition, my nice plans for Etch fell apart, because the license change for libgnat meant that some users would be unhappy about this compiler. While I have no objection to this license change, I felt that my duty as a distribution maintainer was to ask what users wanted and so I did. I asked for all interested people to vote on one of GNAT GPL 2005 Edition, gnat-3.4, gnat-4.0 or gnat-3.4 with patches backported from GNAT GPL, 4.0 and 4.1. Each voter could give one negative vote to express utter rejection, two positive votes for a strong preference or one vote for a normal preference. Here are the results of the vote:

Variant	**Votes**
GNAT GPL 2005 Edition	-11
gnat-3.4	7
gnat-4.0	16
gnat-3.4+patches	0

C.2 Transition plan for Etch

June 2005 - Sarge released. gcc-4.0 enters testing. Several major transitions start simultaneously: g++ from 3.3 to 4.0 (with ABI change), XFree86 4.3 to X.org 6.8, GNOME from 2.8 to 2.10. Other transitions are planned, as well.

September 2005 - Java, Treelang, libffi and several other binary packages are no longer built from gcc-3.3 or gcc-3.4 but only from the gcc-4.0 source package.

November 2005 - Release branch for GCC 4.1 created. gcc-4.1 reaches experimental.

December 2005 - gnat-3.3 removed from Etch.

February 2006 - GCC 4.1.0 released.

March 2006 - gnat-3.4 removed from Etch. Ground work starts on gnat-4.1 in experimental:

- Link the GNAT tools (gnatmake, gnatkr, gnatprep, etc.) dynamically against libgnat-4.1.so, instead of statically. (Done: 2006-03-17)
- Port the support for symbolic tracebacks from gnat 3.15p
- Port libgnatvsn and libgnatprj from gnat 3.15p
- Build ASIS with gnat-4.1. This will probably be the latest GPL version of ASIS, because all the packages that depend on ASIS are GPL, and ASIS allows manipulation of programs and so is not just an execution library.
- Build GLADE with gnat-4.1. This will be the GLADE from AdaCore's CVS repository, under GNAT-Modified GPL.

July 2006 - gnat-4.1 reaches unstable. ghdl transitions to gnat-4.1.

August 2006 - gnat-4.1 reaches testing and becomes the default Ada compiler. All Ada libraries must change their sonames and all packages require recompilation.

September 2006 - toolchain freeze

October 2006 - general freeze

December 2006 - Etch released 18 months after Sarge.

Appendix D libgnat under GPL - for or against

Here is a summary of the arguments that were expressed by various people on comp.lang.ada while debating on the Ada compiler for Etch. The arguments are in no particular order and the summaries are mine.

The anarchist argument: it is immoral for AdaCore to dictate how libgnat should be used; everyone should be free to use libgnat however they see fit and the GPL uses violent but lawful means to make this impossible. Therefore, everyone should reject the GNAT GPL 2005 Edition.

The Free Software argument: it is immoral to write non-free software; it is even more immoral to use a free software library such as libgnat in non-free software. It is appropriate to make this illegal by releasing libgnat under the GPL. Therefore, everyone should embrace GNAT GPL 2005 Edition.

The Other Free Software argument: we make Free Software under a license which is incompatible in some ways with the GPL (e.g. the BSD license). The GPL prohibits linking libgnat with our software and distributing our binaries. Therefore, we reject the GNAT GPL 2005 Edition.

The selfish argument: we make commercial proprietary software with GNAT but we cannot or will not pay for GNAT Pro. If libgnat is under GPL, we can no longer distribute our proprietary software. Therefore, we reject GNAT GPL 2005 Edition.

The interoperability argument: our software has to link with non-free software which is outside our control and the license of which prohibits use of GPLed libraries (variant: we link with other Free libraries which are under licenses not compatible with the GPL); therefore we are forced to reject GNAT GPL 2005 Edition.

The betrayed author's argument: I made contributions to the software that is now supported commercially by AdaCore, with the understanding that the license was the GMGPL. AdaCore revoked the special permission without consulting me. While this is specifically allowed by the GPL, I feel betrayed. Therefore, I reject the GNAT GPL 2005 Edition.

The technical quality argument: GNAT GPL 2005 Edition is the best available Ada compiler. GCC 3.4 is not as up-to-date with respect to Ada 2006 and GCC 4.0 is less stable. Therefore, we should embrace GNAT GPL 2005 Edition.

The marketing argument: licensing libgnat under the GPL hinders promotion of Ada, especially to small businesses (per the selfish argument). The move tries to promote Free Software at the expense of Ada. Free Software does not need much promotion while Ada does. Therefore, we should reject GNAT GPL 2005 Edition.