This manual is for GNU Automake (version 1.12.2, 8 July 2012), a program that creates GNU standards-compliant Makefiles from template files.

Copyright © 1995-2012 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License.”
Table of Contents

1 Introduction ................................................. 1

2 An Introduction to the Autotools ..................... 1
   2.1 Introducing the GNU Build System ................. 1
   2.2 Use Cases for the GNU Build System ............. 2
      2.2.1 Basic Installation .......................... 2
      2.2.2 Standard Makefile Targets .................. 4
      2.2.3 Standard Directory Variables ............... 4
      2.2.4 Standard Configuration Variables .......... 5
      2.2.5 Overriding Default Configuration Setting with config.site .. 6
      2.2.6 Parallel Build Trees (a.k.a. VPATH Builds) ... 6
      2.2.7 Two-Part Installation ........................ 8
      2.2.8 Cross-Compilation ............................ 9
      2.2.9 Renaming Programs at Install Time .......... 10
      2.2.10 Building Binary Packages Using DESTDIR ...... 10
      2.2.11 Preparing Distributions ...................... 11
      2.2.12 Automatic Dependency Tracking ............... 11
      2.2.13 Nested Packages .............................. 12
   2.3 How Autotools Help .................................. 13
   2.4 A Small Hello World ................................. 13
      2.4.1 Creating amhello-1.0.tar.gz .................. 13
      2.4.2 amhello’s configure.ac Setup Explained ... 16
      2.4.3 amhello’s Makefile.am Setup Explained ...... 17

3 General ideas ............................................. 18
   3.1 General Operation .................................. 18
   3.2 Strictness .......................................... 19
   3.3 The Uniform Naming Scheme ......................... 20
   3.4 Staying below the command line length limit .... 22
   3.5 How derived variables are named .................. 22
   3.6 Variables reserved for the user .................... 23
   3.7 Programs automake might require .................. 23

4 Some example packages ................................. 24
   4.1 A simple example, start to finish ................. 25
   4.2 Building true and false ............................ 25

5 Creating a Makefile.in ................................. 26
8.7 Variables used when building a program ....................... 71
8.8 Yacc and Lex support ........................................... 72
8.9 C++ Support .................................................. 74
8.10 Objective C Support .......................................... 75
8.11 Objective C++ Support ......................................... 75
8.12 Unified Parallel C Support ................................... 75
8.13 Assembly Support ............................................ 76
8.14 Fortran 77 Support ............................................ 76
  8.14.1 Preprocessing Fortran 77 ................................. 77
  8.14.2 Compiling Fortran 77 Files ............................... 77
  8.14.3 Mixing Fortran 77 With C and C++ .................. 77
  8.14.3.1 How the Linker is Chosen ........................... 78
8.15 Fortran 9x Support ............................................ 79
  8.15.1 Compiling Fortran 9x Files .............................. 79
8.16 Compiling Java sources using gcj .......................... 79
8.17 Vala Support ................................................ 80
8.18 Support for Other Languages ................................ 80
8.19 Automatic dependency tracking ............................. 80
8.20 Support for executable extensions ......................... 81

9 Other Derived Objects ....................................... 82
  9.1 Executable Scripts ......................................... 82
  9.2 Header files ................................................. 83
  9.3 Architecture-independent data files ....................... 83
  9.4 Built Sources ............................................... 84
    9.4.1 Built Sources Example ................................. 84

10 Other GNU Tools ............................................. 87
  10.1 Emacs Lisp ................................................ 87
  10.2 Gettext .................................................... 88
  10.3 Libtool .................................................... 88
  10.4 Java bytecode compilation (deprecated) ................ 88
  10.5 Python ..................................................... 89

11 Building documentation ...................................... 91
  11.1 Texinfo .................................................... 91
  11.2 Man Pages ................................................. 93

12 What Gets Installed ......................................... 94
  12.1 Basics of Installation .................................... 94
  12.2 The Two Parts of Install .................................. 95
  12.3 Extending Installation ..................................... 95
  12.4 Staged Installs ........................................... 95
  12.5 Install Rules for the User ................................ 96

13 What Gets Cleaned ............................................ 96
### Chapter 14: What Goes in a Distribution

- **14.1 Basics of Distribution**
- **14.2 Fine-grained Distribution Control**
- **14.3 The dist Hook**
- **14.4 Checking the Distribution**
- **14.5 The Types of Distributions**

### Chapter 15: Support for test suites

- **15.1 Generalities about Testing**
- **15.2 Simple Tests**
  - **15.2.1 Scripts-based Testsuites**
  - **15.2.2 Serial Test Harness**
  - **15.2.3 Parallel Test Harness**
- **15.3 Custom Test Drivers**
  - **15.3.1 Overview of Custom Test Drivers Support**
  - **15.3.2 Declaring Custom Test Drivers**
  - **15.3.3 API for Custom Test Drivers**
    - **15.3.3.1 Command-line arguments for test drivers**
    - **15.3.3.2 Log files generation and test results recording**
    - **15.3.3.3 Testsuite progress output**
- **15.4 Using the TAP test protocol**
  - **15.4.1 Introduction to TAP**
  - **15.4.2 Use TAP with the Automake test harness**
  - **15.4.3 Incompatibilities with other TAP parsers and drivers**
  - **15.4.4 Links and external resources on TAP**
- **15.5 DejaGnu Tests**
- **15.6 Install Tests**

### Chapter 16: Rebuilding Makefiles

### Chapter 17: Changing Automake’s Behavior

- **17.1 Options generalities**
- **17.2 List of Automake options**

### Chapter 18: Miscellaneous Rules

- **18.1 Interfacing to etags**
- **18.2 Handling new file extensions**

### Chapter 19: Include

### Chapter 20: Conditionals

- **20.1 Usage of Conditionals**
- **20.2 Limits of Conditionals**
21 Silencing make ........................................... 126
  21.1 Make is verbose by default ............................ 126
  21.2 Standard and generic ways to silence make .......... 127
  21.3 How Automake can help in silencing make .......... 127

22 The effect of --gnu and --gnits ....................... 130

23 The effect of --cygnus ................................ 131

24 When Automake Isn’t Enough ............................ 132
  24.1 Extending Automake Rules ............................ 132
  24.2 Third-Party Makefiles ............................... 133

25 Distributing Makefile.ins .............................. 136

26 Automake API Versioning ............................... 136

27 Upgrading a Package to a Newer Automake Version .... 137

28 Frequently Asked Questions about Automake ........... 138
  28.1 CVS and generated files ............................. 138
  28.2 missing and AM_MAINTAINER_MODE ................ 140
  28.3 Why doesn’t Automake support wildcards? .......... 141
  28.4 Limitations on File Names ........................... 142
  28.5 Errors with distclean ............................... 143
  28.6 Flag Variables Ordering ............................. 145
  28.7 Why are object files sometimes renamed? .......... 147
  28.8 Per-Object Flags Emulation .......................... 148
  28.9 Handling Tools that Produce Many Outputs .......... 149
  28.10 Installing to Hard-Coded Locations ................. 154
  28.11 Debugging Make Rules .............................. 156
  28.12 Reporting Bugs .................................... 156

Appendix A Copying This Manual ......................... 158
  A.1 GNU Free Documentation License ...................... 158

Appendix B Indices ....................................... 166
  B.1 Macro Index .......................................... 166
  B.2 Variable Index ....................................... 166
  B.3 General Index ........................................ 170
1 Introduction

Automake is a tool for automatically generating Makefile.in files from files called Makefile.am. Each Makefile.am is basically a series of make variable definitions\(^1\), with rules being thrown in occasionally. The generated Makefile.in are compliant with the GNU Makefile standards.

The GNU Makefile Standards Document (see Section “Makefile Conventions” in The GNU Coding Standards) is long, complicated, and subject to change. The goal of Automake is to remove the burden of Makefile maintenance from the back of the individual GNU maintainer (and put it on the back of the Automake maintainers).

The typical Automake input file is simply a series of variable definitions. Each such file is processed to create a Makefile.in. There should generally be one Makefile.am per directory of a project.

Automake does constrain a project in certain ways; for instance, it assumes that the project uses Autoconf (see Section “Introduction” in The Autoconf Manual), and enforces certain restrictions on the configure.ac contents.

Automake requires perl in order to generate the Makefile.in. However, the distributions created by Automake are fully GNU standards-compliant, and do not require perl in order to be built.

For more information on bug reports, See Section 28.12 [Reporting Bugs], page 156.

2 An Introduction to the Autotools

If you are new to Automake, maybe you know that it is part of a set of tools called The Autotools. Maybe you’ve already delved into a package full of files named configure, configure.ac, Makefile.in, Makefile.am, aclocal.m4, …, some of them claiming to be generated by Autoconf or Automake. But the exact purpose of these files and their relations is probably fuzzy. The goal of this chapter is to introduce you to this machinery, to show you how it works and how powerful it is. If you’ve never installed or seen such a package, do not worry: this chapter will walk you through it.

If you need some teaching material, more illustrations, or a less automake-centered continuation, some slides for this introduction are available in Alexandre Duret-Lutz’s Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html). This chapter is the written version of the first part of his tutorial.

2.1 Introducing the GNU Build System

It is a truth universally acknowledged, that as a developer in possession of a new package, you must be in want of a build system.

In the Unix world, such a build system is traditionally achieved using the command make (see Section “Overview” in The GNU Make Manual). You express the recipe to build your package in a Makefile. This file is a set of rules to build the files in the package.

\(^1\) These variables are also called make macros in Make terminology, however in this manual we reserve the term macro for Autoconf’s macros.
For instance the program `prog` may be built by running the linker on the files `main.o`, `foo.o`, and `bar.o`; the file `main.o` may be built by running the compiler on `main.c`; etc. Each time `make` is run, it reads `Makefile`, checks the existence and modification time of the files mentioned, decides what files need to be built (or rebuilt), and runs the associated commands.

When a package needs to be built on a different platform than the one it was developed on, its `Makefile` usually needs to be adjusted. For instance the compiler may have another name or require more options. In 1991, David J. MacKenzie got tired of customizing `Makefile` for the 20 platforms he had to deal with. Instead, he handcrafted a little shell script called `configure` to automatically adjust the `Makefile` (see Section “Genesis” in The Autoconf Manual). Compiling his package was now as simple as running `./configure && make`.

Today this process has been standardized in the GNU project. The GNU Coding Standards (see Section “Managing Releases” in The GNU Coding Standards) explains how each package of the GNU project should have a `configure` script, and the minimal interface it should have. The `Makefile` too should follow some established conventions. The result? A unified build system that makes all packages almost indistinguishable by the installer. In its simplest scenario, all the installer has to do is to unpack the package, run `./configure && make && make install`, and repeat with the next package to install.

We call this build system the GNU Build System, since it was grown out of the GNU project. However it is used by a vast number of other packages: following any existing convention has its advantages.

The Autotools are tools that will create a GNU Build System for your package. Autoconf mostly focuses on `configure` and Automake on `Makefile`. It is entirely possible to create a GNU Build System without the help of these tools. However it is rather burdensome and error-prone. We will discuss this again after some illustration of the GNU Build System in action.

### 2.2 Use Cases for the GNU Build System

In this section we explore several use cases for the GNU Build System. You can replay all of these examples on the `amhello-1.0.tar.gz` package distributed with Automake. If Automake is installed on your system, you should find a copy of this file in `prefix/share/doc/automake/amhello-1.0.tar.gz`, where `prefix` is the installation prefix specified during configuration (`prefix` defaults to `/usr/local`, however if Automake was installed by some GNU/Linux distribution it most likely has been set to `/usr`). If you do not have a copy of Automake installed, you can find a copy of this file inside the `doc/` directory of the Automake package.

Some of the following use cases present features that are in fact extensions to the GNU Build System. Read: they are not specified by the GNU Coding Standards, but they are nonetheless part of the build system created by the Autotools. To keep things simple, we do not point out the difference. Our objective is to show you many of the features that the build system created by the Autotools will offer to you.

#### 2.2.1 Basic Installation

The most common installation procedure looks as follows.
Chapter 2: An Introduction to the Autotools

```
~ % tar zxf amhello-1.0.tar.gz
~ % cd amhello-1.0
~/amhello-1.0 % ./configure
...
config.status: creating Makefile
config.status: creating src/Makefile
...
~/amhello-1.0 % make
...
~/amhello-1.0 % make check
...
~/amhello-1.0 % su
Password:
/home/adl/amhello-1.0 # make install
...
/home/adl/amhello-1.0 # exit
~/amhello-1.0 % make installcheck
...
```

The user first unpacks the package. Here, and in the following examples, we will use the non-portable `tar zxf` command for simplicity. On a system without GNU `tar` installed, this command should read `gunzip -c amhello-1.0.tar.gz | tar xf -`.

The user then enters the newly created directory to run the `configure` script. This script probes the system for various features, and finally creates the `Makefile`s. In this toy example there are only two `Makefile`s, but in real-world projects, there may be many more, usually one `Makefile` per directory.

It is now possible to run `make`. This will construct all the programs, libraries, and scripts that need to be constructed for the package. In our example, this compiles the `hello` program. All files are constructed in place, in the source tree; we will see later how this can be changed.

`make check` causes the package's tests to be run. This step is not mandatory, but it is often good to make sure the programs that have been built behave as they should, before you decide to install them. Our example does not contain any tests, so running `make check` is a no-op.

After everything has been built, and maybe tested, it is time to install it on the system. That means copying the programs, libraries, header files, scripts, and other data files from the source directory to their final destination on the system. The command `make install` will do that. However, by default everything will be installed in subdirectories of `/usr/local`: binaries will go into `/usr/local/bin`, libraries will end up in `/usr/local/lib`, etc. This destination is usually not writable by any user, so we assume that we have to become root before we can run `make install`. In our example, running `make install` will copy the program `hello` into `/usr/local/bin` and `README` into `/usr/local/share/doc/amhello`.

A last and optional step is to run `make installcheck`. This command may run tests on the installed files. `make check` tests the files in the source tree, while `make installcheck` tests their installed copies. The tests run by the latter can be different from those run by the
former. For instance, there are tests that cannot be run in the source tree. Conversely, some packages are set up so that `make installcheck` will run the very same tests as `make check`, only on different files (non-installed vs. installed). It can make a difference, for instance when the source tree’s layout is different from that of the installation. Furthermore it may help to diagnose an incomplete installation.

Presently most packages do not have any `installcheck` tests because the existence of `installcheck` is little known, and its usefulness is neglected. Our little toy package is no better: `make installcheck` does nothing.

### 2.2.2 Standard Makefile Targets

So far we have come across four ways to run `make` in the GNU Build System: `make`, `make check`, `make install`, and `make installcheck`. The words `check`, `install`, and `installcheck`, passed as arguments to `make`, are called targets. `make` is a shorthand for `make all`, `all` being the default target in the GNU Build System.

Here is a list of the most useful targets that the GNU Coding Standards specify.

- **make all**: Build programs, libraries, documentation, etc. (same as `make`).
- **make install**: Install what needs to be installed, copying the files from the package’s tree to system-wide directories.
- **make install-strip**: Same as `make install`, then strip debugging symbols. Some users like to trade space for useful bug reports...
- **make uninstall**: The opposite of `make install`: erase the installed files. (This needs to be run from the same build tree that was installed.)
- **make clean**: Erase from the build tree the files built by `make all`.
- **make distclean**: Additionally erase anything `./configure` created.
- **make check**: Run the test suite, if any.
- **make installcheck**: Check the installed programs or libraries, if supported.
- **make dist**: Recreate `package-version.tar.gz` from all the source files.

### 2.2.3 Standard Directory Variables

The GNU Coding Standards also specify a hierarchy of variables to denote installation directories. Some of these are:

<table>
<thead>
<tr>
<th>Directory variable</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>/usr/local</td>
</tr>
<tr>
<td>exec_prefix</td>
<td>${prefix}</td>
</tr>
</tbody>
</table>
bindir      ${exec_prefix}/bin
libdir      ${exec_prefix}/lib
...

includedir  ${prefix}/include
datarootdir ${prefix}/share
datadir     ${datarootdir}
mandir      ${datarootdir}/man
infodir     ${datarootdir}/info
docdir      ${datarootdir}/doc/${PACKAGE}
...

Each of these directories has a role which is often obvious from its name. In a package, any installable file will be installed in one of these directories. For instance in amhello-1.0, the program hello is to be installed in bindir, the directory for binaries. The default value for this directory is /usr/local/bin, but the user can supply a different value when calling configure. Also the file README will be installed into docdir, which defaults to /usr/local/share/doc/amhello.

As a user, if you wish to install a package on your own account, you could proceed as follows:

```
~/amhello-1.0 % ./configure --prefix ~
```

```
~/amhello-1.0 % make
```

```
~/amhello-1.0 % make install
```

This would install ~/usr/bin/hello and ~/usr/share/doc/amhello/README.

The list of all such directory options is shown by ./configure --help.

### 2.2.4 Standard Configuration Variables

The GNU Coding Standards also define a set of standard configuration variables used during the build. Here are some:

- **CC**  
  C compiler command

- **CFLAGS**  
  C compiler flags

- **CXX**  
  C++ compiler command

- **CXXFLAGS**  
  C++ compiler flags

- **LDFLAGS**  
  linker flags

- **CPPFLAGS**  
  C/C++ preprocessor flags

configure usually does a good job at setting appropriate values for these variables, but there are cases where you may want to override them. For instance you may have several versions of a compiler installed and would like to use another one, you may have header files installed outside the default search path of the compiler, or even libraries out of the way of the linker.
Here is how one would call `configure` to force it to use `gcc-3` as C compiler, use header files from `~/usr/include` when compiling, and libraries from `~/usr/lib` when linking.

```
~/amhello-1.0 % ./configure --prefix ~/usr CC=gcc-3 \
CPPFLAGS=-I$HOME/usr/include LDFLAGS=-L$HOME/usr/lib
```

Again, a full list of these variables appears in the output of `./configure --help`.

### 2.2.5 Overriding Default Configuration Setting with `config.site`

When installing several packages using the same setup, it can be convenient to create a file to capture common settings. If a file named `prefix/share/config.site` exists, `configure` will source it at the beginning of its execution.

Recall the command from the previous section:

```
~/amhello-1.0 % ./configure --prefix ~/usr CC=gcc-3 \
CPPFLAGS=-I$HOME/usr/include LDFLAGS=-L$HOME/usr/lib
```

Assuming we are installing many packages in `~/usr`, and will always want to use these definitions of `CC`, `CPPFLAGS`, and `LDFLAGS`, we can automate this by creating the following `~/usr/share/config.site` file:

```
test -z "$CC" && CC=gcc-3

test -z "$CPPFLAGS" && CPPFLAGS=-I$HOME/usr/include

test -z "$LDFLAGS" && LDFLAGS=-L$HOME/usr/lib
```

Now, any time a `configure` script is using the `~/usr` prefix, it will execute the above `config.site` and define these three variables.

```
~/amhello-1.0 % ./configure --prefix ~/usr
configure: loading site script /home/adl/usr/share/config.site
```

See Section “Setting Site Defaults” in *The Autoconf Manual*, for more information about this feature.

### 2.2.6 Parallel Build Trees (a.k.a. VPATH Builds)

The GNU Build System distinguishes two trees: the source tree, and the build tree.

The source tree is rooted in the directory containing `configure`. It contains all the sources files (those that are distributed), and may be arranged using several subdirectories.

The build tree is rooted in the directory in which `configure` was run, and is populated with all object files, programs, libraries, and other derived files built from the sources (and hence not distributed). The build tree usually has the same subdirectory layout as the source tree; its subdirectories are created automatically by the build system.

If `configure` is executed in its own directory, the source and build trees are combined: derived files are constructed in the same directories as their sources. This was the case in our first installation example (see Section 2.2.1 [Basic Installation], page 2).

A common request from users is that they want to confine all derived files to a single directory, to keep their source directories uncluttered. Here is how we could run `configure` to build everything in a subdirectory called `build/`.

```
~ % tar zxf ~/amhello-1.0.tar.gz
~ % cd amhello-1.0
```
These setups, where source and build trees are different, are often called parallel builds or VPATH builds. The expression parallel build is misleading: the word parallel is a reference to the way the build tree shadows the source tree, it is not about some concurrency in the way build commands are run. For this reason we refer to such setups using the name VPATH builds in the following. VPATH is the name of the make feature used by the Makefiles to allow these builds (see Section “VPATH Search Path for All Prerequisites” in The GNU Make Manual).

VPATH builds have other interesting uses. One is to build the same sources with multiple configurations. For instance:

```
~ % tar zxf ~/amhello-1.0.tar.gz
~ % cd amhello-1.0
~/amhello-1.0 % mkdir debug optim && cd debug
~/amhello-1.0/debug % ../configure CFLAGS='-g -O0'
...
~/amhello-1.0/debug % make
...
~/amhello-1.0/debug % cd ../optim
~/amhello-1.0/optim % ../configure CFLAGS='-O3 -fomit-frame-pointer'
...
~/amhello-1.0/optim % make
...
```

With network file systems, a similar approach can be used to build the same sources on different machines. For instance, suppose that the sources are installed on a directory shared by two hosts: HOST1 and HOST2, which may be different platforms.

```
~ % cd /nfs/src
/nfs/src % tar zxf ~/amhello-1.0.tar.gz
On the first host, you could create a local build directory:

[HOST1] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST1] /tmp/amh % /nfs/src/amhello-1.0/configure
...
[HOST1] /tmp/amh % make && sudo make install
...
```

(Here we assume that the installer has configured sudo so it can execute make install with root privileges; it is more convenient than using su like in Section 2.2.1 [Basic Installation], page 2).

On the second host, you would do exactly the same, possibly at the same time:

```
[HOST2] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST2] /tmp/amh % /nfs/src/amhello-1.0/configure
...
[HOST2] /tmp/amh % make && sudo make install
```
In this scenario, nothing forbids the /nfs/src/amhello-1.0 directory from being read-only. In fact VPATH builds are also a means of building packages from a read-only medium such as a CD-ROM. (The FSF used to sell CD-ROM with unpacked source code, before the GNU project grew so big.)

2.2.7 Two-Part Installation

In our last example (see Section 2.2.6 [VPATH Builds], page 6), a source tree was shared by two hosts, but compilation and installation were done separately on each host.

The GNU Build System also supports networked setups where part of the installed files should be shared amongst multiple hosts. It does so by distinguishing architecture-dependent files from architecture-independent files, and providing two `Makefile` targets to install each of these classes of files.

These targets are `install-exec` for architecture-dependent files and `install-data` for architecture-independent files. The command we used up to now, `make install`, can be thought of as a shorthand for `make install-exec install-data`.

From the GNU Build System point of view, the distinction between architecture-dependent files and architecture-independent files is based exclusively on the directory variable used to specify their installation destination. In the list of directory variables we provided earlier (see Section 2.2.3 [Standard Directory Variables], page 4), all the variables based on `exec-prefix` designate architecture-dependent directories whose files will be installed by `make install-exec`. The others designate architecture-independent directories and will serve files installed by `make install-data`. See Section 12.2 [The Two Parts of Install], page 95, for more details.

Here is how we could revisit our two-host installation example, assuming that (1) we want to install the package directly in `/usr`, and (2) the directory `/usr/share` is shared by the two hosts.

On the first host we would run

```
[HOST1] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST1] /tmp/amh % /nfs/src/amhello-1.0/configure --prefix /usr
...
[HOST1] /tmp/amh % make && sudo make install
...
```

On the second host, however, we need only install the architecture-specific files.

```
[HOST2] ~ % mkdir /tmp/amh && cd /tmp/amh
[HOST2] /tmp/amh % /nfs/src/amhello-1.0/configure --prefix /usr
...
[HOST2] /tmp/amh % make && sudo make install-exec
...
```

In packages that have installation checks, it would make sense to run `make installcheck` (see Section 2.2.1 [Basic Installation], page 2) to verify that the package works correctly despite the apparent partial installation.
2.2.8 Cross-Compilation

To *cross-compile* is to build on one platform a binary that will run on another platform. When speaking of cross- compilation, it is important to distinguish between the *build platform* on which the compilation is performed, and the *host platform* on which the resulting executable is expected to run. The following *configure* options are used to specify each of them:

```
--build=build
  The system on which the package is built.

--host=host
  The system where built programs and libraries will run.
```

When the `--host` is used, *configure* will search for the cross-compiling suite for this platform. Cross-compilation tools commonly have their target architecture as prefix of their name. For instance my cross-compiler for MinGW32 has its binaries called `i586-mingw32msvc-gcc`, `i586-mingw32msvc-ld`, `i586-mingw32msvc-as`, etc.

Here is how we could build `amhello-1.0` for `i586-mingw32msvc` on a GNU/Linux PC.

```
~/amhello-1.0 % ./configure --build i686-pc-linux-gnu --host i586-mingw32msvc
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for i586-mingw32msvc-strip... i586-mingw32msvc-strip
checking for i586-mingw32msvc-gcc... i586-mingw32msvc-gcc
checking for C compiler default output file name... a.exe
checking whether the C compiler works... yes
checking whether we are cross compiling... yes
checking for suffix of executables... .exe
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether i586-mingw32msvc-gcc accepts -g... yes
checking for i586-mingw32msvc-gcc option to accept ANSI C...
.
```

```
~/amhello-1.0 % make
.
```

```
~/amhello-1.0 % cd src; file hello.exe
hello.exe: MS Windows PE 32-bit Intel 80386 console executable not relocatable
```

The `--host` and `--build` options are usually all we need for cross-compiling. The only exception is if the package being built is itself a cross-compiler: we need a third option to specify its target architecture.

```
--target=target
  When building compiler tools: the system for which the tools will create output.
```

For instance when installing GCC, the GNU Compiler Collection, we can use `--target=target` to specify that we want to build GCC as a cross-compiler for `target`. Mixing `--build` and `--target`, we can actually cross-compile a cross-compiler; such a three-way cross-compilation is known as a *Canadian cross*.

See Section “Specifying the System Type” in *The Autoconf Manual*, for more information about these *configure* options.
2.2.9 Renaming Programs at Install Time

The GNU Build System provides means to automatically rename executables and man- pages before they are installed (see Section 11.2 [Man Pages], page 93). This is especially convenient when installing a GNU package on a system that already has a proprietary implementation you do not want to overwrite. For instance, you may want to install GNU tar as gtar so you can distinguish it from your vendor’s tar.

This can be done using one of these three configure options.

--program-prefix=prefix
    Prepend prefix to installed program names.

--program-suffix=suffix
    Append suffix to installed program names.

--program-transform-name=program
    Run sed program on installed program names.

The following commands would install hello as /usr/local/bin/test-hello, for instance.

```
~/amhello-1.0 % ./configure --program-prefix test-
...  
~/amhello-1.0 % make
...
~/amhello-1.0 % sudo make install
...
```

2.2.10 Building Binary Packages Using DESTDIR

The GNU Build System’s make install and make uninstall interface does not exactly fit the needs of a system administrator who has to deploy and upgrade packages on lots of hosts. In other words, the GNU Build System does not replace a package manager.

Such package managers usually need to know which files have been installed by a package, so a mere make install is inappropriate.

The DESTDIR variable can be used to perform a staged installation. The package should be configured as if it was going to be installed in its final location (e.g., --prefix /usr), but when running make install, the DESTDIR should be set to the absolute name of a directory into which the installation will be diverted. From this directory it is easy to review which files are being installed where, and finally copy them to their final location by some means.

For instance here is how we could create a binary package containing a snapshot of all the files to be installed.

```
~/amhello-1.0 % ./configure --prefix /usr
...
~/amhello-1.0 % make
...
~/amhello-1.0 % make DESTDIR=$HOME/inst install
...
~/amhello-1.0 % cd ~/inst
~/inst % find . -type f -print > ../files.lst
```
~/inst % tar zcvf ~/amhello-1.0-i686.tar.gz 'cat ../files.lst'
./usr/bin/hello
./usr/share/doc/amhello/README

After this example, amhello-1.0-i686.tar.gz is ready to be uncompressed in / on many hosts. (Using ‘cat ../files.lst’ instead of ‘.’ as argument for tar avoids entries for each subdirectory in the archive: we would not like tar to restore the modification time of /, /usr/, etc.)

Note that when building packages for several architectures, it might be convenient to use make install-data and make install-exec (see Section 2.2.7 [Two-Part Install], page 8) to gather architecture-independent files in a single package.

See Chapter 12 [Install], page 94, for more information.

2.2.11 Preparing Distributions

We have already mentioned make dist. This target collects all your source files and the necessary parts of the build system to create a tarball named package-version.tar.gz.

Another, more useful command is make distcheck. The distcheck target constructs package-version.tar.gz just as well as dist, but it additionally ensures most of the use cases presented so far work:

- It attempts a full compilation of the package (see Section 2.2.1 [Basic Installation], page 2), unpacking the newly constructed tarball, running make, make check, make install, as well as make installcheck, and even make dist,
- it tests VPATH builds with read-only source tree (see Section 2.2.6 [VPATH Builds], page 6),
- it makes sure make clean, make distclean, and make uninstall do not omit any file (see Section 2.2.2 [Standard Targets], page 4),
- and it checks that DESTDIR installations work (see Section 2.2.10 [DESTDIR], page 10).

All of these actions are performed in a temporary subdirectory, so that no root privileges are required.

Releasing a package that fails make distcheck means that one of the scenarios we presented will not work and some users will be disappointed. Therefore it is a good practice to release a package only after a successful make distcheck. This of course does not imply that the package will be flawless, but at least it will prevent some of the embarrassing errors you may find in packages released by people who have never heard about distcheck (like DESTDIR not working because of a typo, or a distributed file being erased by make clean, or even VPATH builds not working).

See Section 2.4.1 [Creating amhello], page 13, to recreate amhello-1.0.tar.gz using make distcheck. See Section 14.4 [Checking the Distribution], page 99, for more information about distcheck.

2.2.12 Automatic Dependency Tracking

Dependency tracking is performed as a side-effect of compilation. Each time the build system compiles a source file, it computes its list of dependencies (in C these are the header files included by the source being compiled). Later, any time make is run and a dependency appears to have changed, the dependent files will be rebuilt.
Automake generates code for automatic dependency tracking by default, unless the developer chooses to override it; for more information, see Section 8.19 [Dependencies], page 80.

When `configure` is executed, you can see it probing each compiler for the dependency mechanism it supports (several mechanisms can be used):

```
~/amhello-1.0 % ./configure --prefix /usr
...
checking dependency style of gcc... gcc3
...
```

Because dependencies are only computed as a side-effect of the compilation, no dependency information exists the first time a package is built. This is OK because all the files need to be built anyway: `make` does not have to decide which files need to be rebuilt. In fact, dependency tracking is completely useless for one-time builds and there is a `configure` option to disable this:

`--disable-dependency-tracking`

Speed up one-time builds.

Some compilers do not offer any practical way to derive the list of dependencies as a side-effect of the compilation, requiring a separate run (maybe of another tool) to compute these dependencies. The performance penalty implied by these methods is important enough to disable them by default. The option `--enable-dependency-tracking` must be passed to `configure` to activate them.

`--enable-dependency-tracking`

Do not reject slow dependency extractors.

See Section “Dependency Tracking Evolution” in Brief History of Automake, for some discussion about the different dependency tracking schemes used by Automake over the years.

### 2.2.13 Nested Packages

Although nesting packages isn’t something we would recommend to someone who is discovering the Autotools, it is a nice feature worthy of mention in this small advertising tour.

Autoconfiscated packages (that means packages whose build system have been created by Autoconf and friends) can be nested to arbitrary depth.

A typical setup is that package A will distribute one of the libraries it needs in a subdirectory. This library B is a complete package with its own GNU Build System. The `configure` script of A will run the `configure` script of B as part of its execution, building and installing A will also build and install B. Generating a distribution for A will also include B.

It is possible to gather several packages like this. GCC is a heavy user of this feature. This gives installers a single package to configure, build and install, while it allows developers to work on subpackages independently.

When configuring nested packages, the `configure` options given to the top-level `configure` are passed recursively to nested `configures`. A package that does not understand an option will ignore it, assuming it is meaningful to some other package.
The command `configure --help=recursive` can be used to display the options supported by all the included packages.

See Section 7.4 [Subpackages], page 52, for an example setup.

## 2.3 How Autotools Help

There are several reasons why you may not want to implement the GNU Build System yourself (read: write a `configure` script and `Makefile` yourself).

- As we have seen, the GNU Build System has a lot of features (see Section 2.2 [Use Cases], page 2). Some users may expect features you have not implemented because you did not need them.
- Implementing these features portably is difficult and exhausting. Think of writing portable shell scripts, and portable `Makefile`s, for systems you may not have handy. See Section “Portable Shell Programming” in *The Autoconf Manual*, to convince yourself.
- You will have to upgrade your setup to follow changes to the GNU Coding Standards.

The GNU Autotools take all this burden off your back and provide:

- Tools to create a portable, complete, and self-contained GNU Build System, from simple instructions. *Self-contained* meaning the resulting build system does not require the GNU Autotools.
- A central place where fixes and improvements are made: a bug-fix for a portability issue will benefit every package.

Yet there also exist reasons why you may want NOT to use the Autotools... For instance you may be already using (or used to) another incompatible build system. Autotools will only be useful if you do accept the concepts of the GNU Build System. People who have their own idea of how a build system should work will feel frustrated by the Autotools.

## 2.4 A Small Hello World

In this section we recreate the `amhello-1.0` package from scratch. The first subsection shows how to call the Autotools to instantiate the GNU Build System, while the second explains the meaning of the `configure.ac` and `Makefile.am` files read by the Autotools.

### 2.4.1 Creating `amhello-1.0.tar.gz`

Here is how we can recreate `amhello-1.0.tar.gz` from scratch. The package is simple enough so that we will only need to write 5 files. (You may copy them from the final `amhello-1.0.tar.gz` that is distributed with Automake if you do not want to write them.)

Create the following files in an empty directory.

- `src/main.c` is the source file for the `hello` program. We store it in the `src/` subdirectory, because later, when the package evolves, it will ease the addition of a `man/` directory for man pages, a `data/` directory for data files, etc.

  ```
  ~/amhello % cat src/main.c
  #include <config.h>
  #include <stdio.h>
  ```
int
main (void)
{
   puts ("Hello World!");
   puts ("This is " PACKAGE_STRING ".");
   return 0;
}

• README contains some very limited documentation for our little package.

`~/amhello % cat README`

This is a demonstration package for GNU Automake.
Type 'info Automake' to read the Automake manual.

• Makefile.am and src/Makefile.am contain Automake instructions for these two directories.

`~/amhello % cat src/Makefile.am`
bin_PROGRAMS = hello
hello_SOURCES = main.c

`~/amhello % cat Makefile.am`
SUBDIRS = src
dist_doc_DATA = README

• Finally, configure.ac contains Autoconf instructions to create the configure script.

`~/amhello % cat configure.ac`
AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([
   Makefile
   src/Makefile
])
AC_OUTPUT

Once you have these five files, it is time to run the Autotools to instantiate the build system. Do this using the autoreconf command as follows:

`~/amhello % autoreconf --install`
configure.ac: installing './install-sh'
configure.ac: installing './missing'
src/Makefile.am: installing './depcomp'

At this point the build system is complete.

In addition to the three scripts mentioned in its output, you can see that autoreconf created four other files: configure, config.h.in, Makefile.in, and src/Makefile.in. The latter three files are templates that will be adapted to the system by configure under the names config.h, Makefile, and src/Makefile. Let’s do this:

`~/amhello % ./configure`
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: executing depfiles commands

You can see Makefile, src/Makefile, and config.h being created at the end after configure has probed the system. It is now possible to run all the targets we wish (see Section 2.2.2 [Standard Targets], page 4). For instance:

```
~/amhello % make
...
~/amhello % src/hello
Hello World!
This is amhello 1.0.
~/amhello % make distcheck
...
==================================================================
amhello-1.0 archives ready for distribution:
amhello-1.0.tar.gz
==================================================================
```

Note that running autoreconf is only needed initially when the GNU Build System does not exist. When you later change some instructions in a Makefile.am or configure.ac, the relevant part of the build system will be regenerated automatically when you execute make.

autoreconf is a script that calls autoconf, automake, and a bunch of other commands in the right order. If you are beginning with these tools, it is not important to figure out in which order all of these tools should be invoked and why. However, because Autoconf and Automake have separate manuals, the important point to understand is that autoconf is in charge of creating configure from configure.ac, while automake is in charge of creating Makefile.ins from Makefile.ams and configure.ac. This should at least direct you to the right manual when seeking answers.
2.4.2 amhello’s configure.ac Setup Explained

Let us begin with the contents of configure.ac.

```plaintext
AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([
    Makefile
    src/Makefile
])
AC_OUTPUT
```

This file is read by both autoconf (to create configure) and automake (to create the various Makefile.in). It contains a series of M4 macros that will be expanded as shell code to finally form the configure script. We will not elaborate on the syntax of this file, because the Autoconf manual has a whole section about it (see Section “Writing configure.ac” in The Autoconf Manual).

The macros prefixed with AC_ are Autoconf macros, documented in the Autoconf manual (see Section “Autoconf Macro Index” in The Autoconf Manual). The macros that start with AM_ are Automake macros, documented later in this manual (see Section B.1 [Macro Index], page 166).

The first two lines of configure.ac initialize Autoconf and Automake. AC_INIT takes in as parameters the name of the package, its version number, and a contact address for bug-reports about the package (this address is output at the end of ./configure --help, for instance). When adapting this setup to your own package, by all means please do not blindly copy Automake’s address: use the mailing list of your package, or your own mail address.

The argument to AM_INIT_AUTOMAKE is a list of options for automake (see Chapter 17 [Options], page 117). -Wall and -Werror ask automake to turn on all warnings and report them as errors. We are speaking of Automake warnings here, such as dubious instructions in Makefile.am. This has absolutely nothing to do with how the compiler will be called, even though it may support options with similar names. Using -Wall -Werror is a safe setting when starting to work on a package: you do not want to miss any issues. Later you may decide to relax things a bit. The foreign option tells Automake that this package will not follow the GNU Standards. GNU packages should always distribute additional files such as ChangeLog, AUTHORS, etc. We do not want automake to complain about these missing files in our small example.

The AC_PROG_CC line causes the configure script to search for a C compiler and define the variable CC with its name. The src/Makefile.in file generated by Automake uses the variable CC to build hello, so when configure creates src/Makefile from src/Makefile.in, it will define CC with the value it has found. If Automake is asked to create a Makefile.in that uses CC but configure.ac does not define it, it will suggest you add a call to AC_PROG_CC.

The AC_CONFIG_HEADERS([config.h]) invocation causes the configure script to create a config.h file gathering ‘#define’s defined by other macros in configure.ac. In our case,
the ACINIT macro already defined a few of them. Here is an excerpt of config.h after configure has run:

```c
... /* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "bug-automake@gnu.org"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "amhello 1.0"
... 
```

As you probably noticed, src/main.c includes config.h so it can use PACKAGE_STRING. In a real-world project, config.h can grow really big, with one ‘#define’ per feature probed on the system.

The AC_CONFIG_FILES macro declares the list of files that configure should create from their *.in templates. Automake also scans this list to find the Makefile.am files it must process. (This is important to remember: when adding a new directory to your project, you should add its Makefile to this list, otherwise Automake will never process the new Makefile.am you wrote in that directory.)

Finally, the AC_OUTPUT line is a closing command that actually produces the part of the script in charge of creating the files registered with AC_CONFIG_HEADERS and AC_CONFIG_FILES.

When starting a new project, we suggest you start with such a simple configure.ac, and gradually add the other tests it requires. The command autoscan can also suggest a few of the tests your package may need (see Section “Using autoscan to Create configure.ac” in The Autoconf Manual).

### 2.4.3 amhello’s Makefile.am Setup Explained

We now turn to src/Makefile.am. This file contains Automake instructions to build and install hello.

```make
bin_PROGRAMS = hello
hello_SOURCES = main.c
```

A Makefile.am has the same syntax as an ordinary Makefile. When automake processes a Makefile.am it copies the entire file into the output Makefile.in (that will be later turned into Makefile by configure) but will react to certain variable definitions by generating some build rules and other variables. Often Makefile.am files contain only a list of variable definitions as above, but they can also contain other variable and rule definitions that automake will pass along without interpretation.

Variables that end with _PROGRAMS are special variables that list programs that the resulting Makefile should build. In Automake speak, this _PROGRAMS suffix is called a primary; Automake recognizes other primaries such as _SCRIPTS, _DATA, _LIBRARIES, etc. corresponding to different types of files.

The ‘bin’ part of the bin_PROGRAMS tells automake that the resulting programs should be installed in bindir. Recall that the GNU Build System uses a set of variables to denote destination directories and allow users to customize these locations (see Section 2.2.3 [Standard Directory Variables], page 4). Any such directory variable can be put in front of a primary (omitting the dir suffix) to tell automake where to install the listed files.
Programs need to be built from source files, so for each program `prog` listed in a `_PROGRAMS` variable, `automake` will look for another variable named `prog_SOURCES` listing its source files. There may be more than one source file: they will all be compiled and linked together.

Automake also knows that source files need to be distributed when creating a tarball (unlike built programs). So a side-effect of this `hello_SOURCES` declaration is that `main.c` will be part of the tarball created by `make dist`.

Finally here are some explanations regarding the top-level `Makefile.am`.

```
SUBDIRS = src
dist_doc_DATA = README
```

`SUBDIRS` is a special variable listing all directories that `make` should recurse into before processing the current directory. So this line is responsible for `make` building `src/hello` even though we run it from the top-level. This line also causes `make install` to install `src/hello` before installing `README` (not that this order matters).

The line `dist_doc_DATA = README` causes `README` to be distributed and installed in `docdir`. Files listed with the `_DATA` primary are not automatically part of the tarball built with `make dist`, so we add the `dist_` prefix so they get distributed. However, for `README` it would not have been necessary: `automake` automatically distributes any `README` file it encounters (the list of other files automatically distributed is presented by `automake --help`). The only important effect of this second line is therefore to install `README` during `make install`.

One thing not covered in this example is accessing the installation directory values (see Section 2.2.3 [Standard Directory Variables], page 4) from your program code, that is, converting them into defined macros. For this, see Section “Defining Directories” in *The Autoconf Manual*.

### 3 General ideas

The following sections cover a few basic ideas that will help you understand how Automake works.

#### 3.1 General Operation

Automake works by reading a `Makefile.am` and generating a `Makefile.in`. Certain variables and rules defined in the `Makefile.am` instruct Automake to generate more specialized code; for instance, a `bin_PROGRAMS` variable definition will cause rules for compiling and linking programs to be generated.

The variable definitions and rules in the `Makefile.am` are copied mostly verbatim into the generated file, with all variable definitions preceding all rules. This allows you to add almost arbitrary code into the generated `Makefile.in`. For instance, the Automake distribution includes a non-standard rule for the `git-dist` target, which the Automake maintainer uses to make distributions from the source control system.

Note that most GNU make extensions are not recognized by Automake. Using such extensions in a `Makefile.am` will lead to errors or confusing behavior.
A special exception is that the GNU make append operator, ‘+=’, is supported. This operator appends its right hand argument to the variable specified on the left. Automake will translate the operator into an ordinary ‘=’ operator; ‘+=’ will thus work with any make program.

Automake tries to keep comments grouped with any adjoining rules or variable definitions.

Generally, Automake is not particularly smart in the parsing of unusual Makefile constructs, so you’re advised to avoid fancy constructs or “creative” use of whitespaces. For example, TAB characters cannot be used between a target name and the following “;” character, and variable assignments shouldn’t be indented with TAB characters. Also, using more complex macro in target names can cause trouble:

```bash
% cat Makefile.am
$(FOO:=x): bar
% automake
Makefile.am:1: bad characters in variable name '$(FOO'
Makefile.am:1: ':'-style assignments are not portable
```

A rule defined in `Makefile.am` generally overrides any such rule of a similar name that would be automatically generated by `automake`. Although this is a supported feature, it is generally best to avoid making use of it, as sometimes the generated rules are very particular.

Similarly, a variable defined in `Makefile.am` or AC_SUBSTed from `configure.ac` will override any definition of the variable that `automake` would ordinarily create. This feature is more often useful than the ability to override a rule. Be warned that many of the variables generated by `automake` are considered to be for internal use only, and their names might change in future releases.

When examining a variable definition, Automake will recursively examine variables referenced in the definition. For example, if Automake is looking at the content of `foo_SOURCES` in this snippet

```bash
xs = a.c b.c
foo_SOURCES = c.c $(xs)
```

it would use the files `a.c`, `b.c`, and `c.c` as the contents of `foo_SOURCES`.

Automake also allows a form of comment that is not copied into the output; all lines beginning with ‘##’ (leading spaces allowed) are completely ignored by Automake.

It is customary to make the first line of `Makefile.am` read:

```bash
## Process this file with automake to produce Makefile.in
```

### 3.2 Strictness

While Automake is intended to be used by maintainers of GNU packages, it does make some effort to accommodate those who wish to use it, but do not want to use all the GNU conventions.

To this end, Automake supports three levels of strictness—the strictness indicating how stringently Automake should check standards conformance.
The valid strictness levels are:

**foreign**  Automake will check for only those things that are absolutely required for proper operations. For instance, whereas GNU standards dictate the existence of a NEWS file, it will not be required in this mode. This strictness will also turn off some warnings by default (among them, portability warnings). The name comes from the fact that Automake is intended to be used for GNU programs; these relaxed rules are not the standard mode of operation.

**gnu**  Automake will check—as much as possible—for compliance to the GNU standards for packages. This is the default.

**gnits**  Automake will check for compliance to the as-yet-unwritten Gnits standards. These are based on the GNU standards, but are even more detailed. Unless you are a Gnits standards contributor, it is recommended that you avoid this option until such time as the Gnits standard is actually published (which may never happen).

See Chapter 22 [Gnits], page 130, for more information on the precise implications of the strictness level.

Automake also has a special (and today deprecated) “cygnus” mode that is similar to strictness but handled differently. This mode is useful for packages that are put into a “Cygnus” style tree (e.g., older versions of the GCC and gdb trees). See Chapter 23 [Cygnus], page 131, for more information on this mode. Please note that this mode is deprecated and will be removed in the next major Automake release (1.13); you must avoid its use in new packages, and should stop using it in existing packages as well.

### 3.3 The Uniform Naming Scheme

Automake variables generally follow a uniform naming scheme that makes it easy to decide how programs (and other derived objects) are built, and how they are installed. This scheme also supports configure time determination of what should be built.

At make time, certain variables are used to determine which objects are to be built. The variable names are made of several pieces that are concatenated together.

The piece that tells automake what is being built is commonly called the primary. For instance, the primary PROGRAMS holds a list of programs that are to be compiled and linked.

A different set of names is used to decide where the built objects should be installed. These names are prefixes to the primary, and they indicate which standard directory should be used as the installation directory. The standard directory names are given in the GNU standards (see Section “Directory Variables” in The GNU Coding Standards). Automake extends this list with pkgdatadir, pkgincludedir, pkglibdir, and pkglibexecdir; these are the same as the non-‘pkg’ versions, but with ‘$(PACKAGE)’ appended. For instance, pkglibdir is defined as ‘$(libdir)/$(PACKAGE)’.

For each primary, there is one additional variable named by prepending ‘EXTRA_’ to the primary name. This variable is used to list objects that may or may not be built, depending on what configure decides. This variable is required because Automake must statically know the entire list of objects that may be built in order to generate a Makefile.in that will work in all cases.
For instance, `cpio` decides at configure time which programs should be built. Some of the programs are installed in `bindir`, and some are installed in `sbindir`:

```plaintext
EXTRA_PROGRAMS = mt rmt
bin_PROGRAMS = cpio pax
sbin_PROGRAMS = $(MORE_PROGRAMS)
```

Defining a primary without a prefix as a variable, e.g., `PROGRAMS`, is an error.

Note that the common `dir` suffix is left off when constructing the variable names; thus one writes `bin_PROGRAMS` and not `bindir_PROGRAMS`.

Not every sort of object can be installed in every directory. Automake will flag those attempts it finds in error (but see below how to override the check if you really need to). Automake will also diagnose obvious misspellings in directory names.

Sometimes the standard directories—even as augmented by Automake—are not enough. In particular it is sometimes useful, for clarity, to install objects in a subdirectory of some predefined directory. To this end, Automake allows you to extend the list of possible installation directories. A given prefix (e.g., `zar`) is valid if a variable of the same name with `dir` appended is defined (e.g., `zardir`).

For instance, the following snippet will install `file.xml` into `$(datadir)/xml`.

```plaintext
xmldir = $(datadir)/xml
xml_DATA = file.xml
```

This feature can also be used to override the sanity checks Automake performs to diagnose suspicious directory/primary couples (in the unlikely case these checks are undesirable, and you really know what you’re doing). For example, Automake would error out on this input:

```plaintext
# Forbidden directory combinations, automake will error out on this.
pkglib_PROGRAMS = foo
doc_LIBRARIES = libquux.a
```

but it will succeed with this:

```plaintext
# Work around forbidden directory combinations. Do not use this
# without a very good reason!
my_execbindir = $(pkglibdir)
my_doclibdir = $(docdir)
my_execbin_PROGRAMS = foo
my_doclib_LIBRARIES = libquux.a
```

The `exec` substring of the `my_execbindir` variable lets the files be installed at the right time (see Section 12.2 [The Two Parts of Install], page 95).

The special prefix `noinst_` indicates that the objects in question should be built but not installed at all. This is usually used for objects required to build the rest of your package, for instance static libraries (see Section 8.2 [A Library], page 57), or helper scripts.

The special prefix `check_` indicates that the objects in question should not be built until the `make check` command is run. Those objects are not installed either.

The current primary names are `PROGRAMS`, `LIBRARIES`, `LTLIBRARIES`, `LISP`, `PYTHON`, `JAVA`, `SCRIPTS`, `DATA`, `HEADERS`, `MANS`, and `TEXINFOS`.

Some primaries also allow additional prefixes that control other aspects of Automake’s behavior. The currently defined prefixes are `dist_`, `nondist_`, `nobase_`, and `notrans_`. 
3.4 Staying below the command line length limit

Traditionally, most unix-like systems have a length limitation for the command line arguments and environment contents when creating new processes (see for example http://www.in-ulm.de/~mascheck/various/argmax/ for an overview on this issue), which of course also applies to commands spawned by make. POSIX requires this limit to be at least 4096 bytes, and most modern systems have quite high limits (or are unlimited).

In order to create portable Makefiles that do not trip over these limits, it is necessary to keep the length of file lists bounded. Unfortunately, it is not possible to do so fully transparently within Automake, so your help may be needed. Typically, you can split long file lists manually and use different installation directory names for each list. For example,

```
data_DATA = file1 ... fileN fileN+1 ... file2N
```

may also be written as

```
data_DATA = file1 ... fileN
data2dir = $(datadir)
data2_DATA = fileN+1 ... file2N
```

and will cause Automake to treat the two lists separately during make install. See Section 12.2 [The Two Parts of Install], page 95, for choosing directory names that will keep the ordering of the two parts of installation. Note that make dist may still only work on a host with a higher length limit in this example.

Automake itself employs a couple of strategies to avoid long command lines. For example, when `${srcdir}/` is prepended to file names, as can happen with above `${data_DATA}` lists, it limits the amount of arguments passed to external commands.

Unfortunately, some system’s make commands may prepend VPATH prefixes like `${srcdir}/` to file names from the source tree automatically (see Section “Automatic Rule Rewriting” in The Autoconf Manual). In this case, the user may have to switch to use GNU Make, or refrain from using VPATH builds, in order to stay below the length limit.

For libraries and programs built from many sources, convenience archives may be used as intermediates in order to limit the object list length (see Section 8.3.5 [Libtool Convenience Libraries], page 61).

3.5 How derived variables are named

Sometimes a Makefile variable name is derived from some text the maintainer supplies. For instance, a program name listed in ‘_PROGRAMS’ is rewritten into the name of a ‘_SOURCES’ variable. In cases like this, Automake canonicalizes the text, so that program names and the like do not have to follow Makefile variable naming rules. All characters in the name except for letters, numbers, the strudel (@), and the underscore are turned into underscores when making variable references.

For example, if your program is named sniff-glue, the derived variable name would be ‘sniff_glue_SOURCES’, not ‘sniff-glue_SOURCES’. Similarly the sources for a library named libmumble++.a should be listed in the ‘libmumble___a_SOURCES’ variable.
The strudel is an addition, to make the use of Autoconf substitutions in variable names less obfuscating.

### 3.6 Variables reserved for the user

Some *Makefile* variables are reserved by the GNU Coding Standards for the use of the “user”—the person building the package. For instance, `CFLAGS` is one such variable.

Sometimes package developers are tempted to set user variables such as `CFLAGS` because it appears to make their job easier. However, the package itself should never set a user variable, particularly not to include switches that are required for proper compilation of the package. Since these variables are documented as being for the package builder, that person rightfully expects to be able to override any of these variables at build time.

To get around this problem, Automake introduces an automake-specific shadow variable for each user flag variable. (Shadow variables are not introduced for variables like `CC`, where they would make no sense.) The shadow variable is named by prepending ‘`AM_`’ to the user variable’s name. For instance, the shadow variable for `YFLAGS` is `AM_YFLAGS`. The package maintainer—that is, the author(s) of the `Makefile.am` and `configure.ac` files—may adjust these shadow variables however necessary.

See Section 28.6 [Flag Variables Ordering], page 145, for more discussion about these variables and how they interact with per-target variables.

### 3.7 Programs automake might require

Automake sometimes requires helper programs so that the generated *Makefile* can do its work properly. There are a fairly large number of them, and we list them here.

Although all of these files are distributed and installed with Automake, a couple of them are maintained separately. The Automake copies are updated before each release, but we mention the original source in case you need more recent versions.

- **ar-lib**
  This is a wrapper primarily for the Microsoft lib archiver, to make it more POSIX-like.

- **compile**
  This is a wrapper for compilers that do not accept options `-c` and `-o` at the same time. It is only used when absolutely required. Such compilers are rare, with the Microsoft C/C++ Compiler as the most notable exception. This wrapper also makes the following common options available for that compiler, while performing file name translation where needed: `-I`, `-L`, `-l`, `-Wl`, and `-Xlinker`.

- **config.guess**
- **config.sub**
  These two programs compute the canonical triplets for the given build, host, or target architecture. These programs are updated regularly to support new architectures and fix probes broken by changes in new kernel versions. Each new release of Automake comes with up-to-date copies of these programs. If your copy of Automake is getting old, you are encouraged to fetch the latest versions of these files from [http://savannah.gnu.org/git/?group=config](http://savannah.gnu.org/git/?group=config) before making a release.
depcomp  This program understands how to run a compiler so that it will generate not
only the desired output but also dependency information that is then used by
the automatic dependency tracking feature (see Section 8.19 [Dependencies],
page 80).

elisp-comp  This program is used to byte-compile Emacs Lisp code.

install-sh  This is a replacement for the install program that works on platforms where
install is unavailable or unusable.

mdate-sh    This script is used to generate a version.texi file. It examines a file and prints
some date information about it.

missing    This wraps a number of programs that are typically only required by maintainers. If
the program in question doesn’t exist, missing prints an informative
warning and attempts to fix things so that the build can continue.

minstalldirs
This script used to be a wrapper around ‘mkdir -p’, which is not portable. Now
we prefer to use ‘install-sh -d’ when configure finds that ‘mkdir -p’ does
not work, this makes one less script to distribute.

For backward compatibility minstalldirs is still used and distributed when
automake finds it in a package. But it is no longer installed automatically, and
it should be safe to remove it.

py-compile  This is used to byte-compile Python scripts.

test-driver  This implements the default test driver offered by the parallel testsuite harness.

texinfo.tex Not a program, this file is required for ‘make dvi’, ‘make ps’ and ‘make pdf’
to work when Texinfo sources are in the package. The latest version can be

ylwrap    This program wraps lex and yacc to rename their output files. It also ensures
that, for instance, multiple yacc instances can be invoked in a single directory
in parallel.

4 Some example packages

This section contains two small examples.

The first example (see Section 4.1 [Complete], page 25) assumes you have an existing
project already using Autoconf, with handcrafted Makefiles, and that you want to convert
it to using Automake. If you are discovering both tools, it is probably better that you look
at the Hello World example presented earlier (see Section 2.4 [Hello World], page 13).

The second example (see Section 4.2 [true], page 25) shows how two programs can be
built from the same file, using different compilation parameters. It contains some technical
digressions that are probably best skipped on first read.
4.1 A simple example, start to finish

Let’s suppose you just finished writing zardoz, a program to make your head float from vortex to vortex. You’ve been using Autoconf to provide a portability framework, but your Makefile.ins have been ad-hoc. You want to make them bulletproof, so you turn to Automake.

The first step is to update your configure.ac to include the commands that automake needs. The way to do this is to add an AM_INIT_AUTOMAKE call just after AC_INIT:

```latex
AC_INIT([zardoz], [1.0])
AM_INIT_AUTOMAKE
...
```

Since your program doesn’t have any complicating factors (e.g., it doesn’t use gettext, it doesn’t want to build a shared library), you’re done with this part. That was easy!

Now you must regenerate configure. But to do that, you’ll need to tell autoconf how to find the new macro you’ve used. The easiest way to do this is to use the aclocal program to generate your aclocal.m4 for you. But wait... maybe you already have an aclocal.m4, because you had to write some hairy macros for your program. The aclocal program lets you put your own macros into acinclude.m4, so simply rename and then run:

```bash
mv aclocal.m4 acinclude.m4
aclocal
autoconf
```

Now it is time to write your Makefile.am for zardoz. Since zardoz is a user program, you want to install it where the rest of the user programs go: bindir. Additionally, zardoz has some Texinfo documentation. Your configure.ac script uses AC_REPLACE_FUNCS, so you need to link against ‘$(LIBOBJS)’. So here’s what you’d write:

```latex
bin_PROGRAMS = zardoz
zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c
zardoz_LDADD = $(LIBOBJS)

info_TEXINFOS = zardoz.texi
```

Now you can run ‘automake --add-missing’ to generate your Makefile.in and grab any auxiliary files you might need, and you’re done!

4.2 Building true and false

Here is another, trickier example. It shows how to generate two programs (true and false) from the same source file (true.c). The difficult part is that each compilation of true.c requires different cpp flags.

```bash
bin_PROGRAMS = true false
false_SOURCES =
false_LDADD = false.o

true.o: true.c
   $(COMPILE) -DEXIT_CODE=0 -c true.c

false.o: true.c
```


Note that there is no `true_SOURCES` definition. Automake will implicitly assume that there is a source file named `true.c` (see Section 8.5 [Default_SOURCES], page 68), and define rules to compile `true.o` and link `true`. The `true: true.c` rule supplied by the above `Makefile.am`, will override the Automake generated rule to build `true.o`.

`false_SOURCES` is defined to be empty—that way no implicit value is substituted. Because we have not listed the source of `false`, we have to tell Automake how to link the program. This is the purpose of the `false_LDADD` line. A `false_DEPENDENCIES` variable, holding the dependencies of the `false` target will be automatically generated by Automake from the content of `false_LDADD`.

The above rules won’t work if your compiler doesn’t accept both `-c` and `-o`. The simplest fix for this is to introduce a bogus dependency (to avoid problems with a parallel `make`):

```
  true.o: true.c false.o
  $(COMPILE) -DEXIT_CODE=0 -c true.c
```

```
  false.o: true.c
  $(COMPILE) -DEXIT_CODE=1 -c true.c && mv true.o false.o
```

As it turns out, there is also a much easier way to do this same task. Some of the above technique is useful enough that we’ve kept the example in the manual. However if you were to build `true` and `false` in real life, you would probably use per-program compilation flags, like so:

```
bin_PROGRAMS = false true
false_SOURCES = true.c
false_CPPFLAGS = -DEXIT_CODE=1
true_SOURCES = true.c
true_CPPFLAGS = -DEXIT_CODE=0
```

In this case Automake will cause `true.c` to be compiled twice, with different flags. In this instance, the names of the object files would be chosen by autoconf; they would be `false-true.o` and `true-true.o`. (The name of the object files rarely matters.)

## 5 Creating a `Makefile.in`

To create all the `Makefile.ins` for a package, run the `automake` program in the top level directory, with no arguments. `automake` will automatically find each appropriate `Makefile.am` (by scanning `configure.ac`; see Chapter 6 [configure], page 29) and generate the corresponding `Makefile.in`. Note that `automake` has a rather simplistic view of what constitutes a package; it assumes that a package has only one `configure.ac`, at the top. If your package has multiple `configure.ac`s, then you must run `automake` in each directory holding a `configure.ac`. (Alternatively, you may rely on Autoconf’s `autoreconf`, which is able to recurse your package tree and run `automake` where appropriate.)

You can optionally give `automake` an argument; `.am` is appended to the argument and the result is used as the name of the input file. This feature is generally only used to
automatically rebuild an out-of-date Makefile.in. Note that automake must always be run from the topmost directory of a project, even if being used to regenerate the Makefile.in in some subdirectory. This is necessary because automake must scan configure.ac, and because automake uses the knowledge that a Makefile.in is in a subdirectory to change its behavior in some cases.

Automake will run autoconf to scan configure.ac and its dependencies (i.e., aclocal.m4 and any included file), therefore autoconf must be in your PATH. If there is an AUTOCONF variable in your environment it will be used instead of autoconf, this allows you to select a particular version of Autoconf. By the way, don’t misunderstand this paragraph: automake runs autoconf to scan your configure.ac, this won’t build configure and you still have to run autoconf yourself for this purpose.

automake accepts the following options:

-a
--add-missing
Automake requires certain common files to exist in certain situations; for instance, config.guess is required if configure.ac invokes AC_CANONICAL_HOST. Automake is distributed with several of these files (see Section 3.7 [Auxiliary Programs], page 23); this option will cause the missing ones to be automatically added to the package, whenever possible. In general if Automake tells you a file is missing, try using this option. By default Automake tries to make a symbolic link pointing to its own copy of the missing file; this can be changed with --copy.

Many of the potentially-missing files are common scripts whose location may be specified via the AC_CONFIG_AUX_DIR macro. Therefore, AC_CONFIG_AUX_DIR’s setting affects whether a file is considered missing, and where the missing file is added (see Section 6.2 [Optional], page 31).

In some strictness modes, additional files are installed, see Chapter 22 [Gnits], page 130, for more information.

--libdir=dir
Look for Automake data files in directory dir instead of in the installation directory. This is typically used for debugging.

--print-libdir
Print the path of the installation directory containing Automake-provided scripts and data files (like e.g., texinfo.texi and install-sh).

c
--copy
When used with --add-missing, causes installed files to be copied. The default is to make a symbolic link.

--cygnus
Causes the generated Makefile.ins to follow Cygnus rules, instead of GNU or Gnits rules. For more information, see Chapter 23 [Cygnus], page 131. Note that this mode of operation is deprecated, and will be removed in the next major Automake release (1.13).
-f
--force-missing
When used with --add-missing, causes standard files to be reinstalled even if they already exist in the source tree. This involves removing the file from the source tree before creating the new symlink (or, with --copy, copying the new file).

--foreign
Set the global strictness to foreign. For more information, see Section 3.2 [Strictness], page 19.

--gnits
Set the global strictness to gnits. For more information, see Chapter 22 [Gnits], page 130.

--gnu
Set the global strictness to gnu. For more information, see Chapter 22 [Gnits], page 130. This is the default strictness.

--help
Print a summary of the command line options and exit.

-i
--ignore-deps
This disables the dependency tracking feature in generated Makefiles; see Section 8.19 [Dependencies], page 80.

--include-deps
This enables the dependency tracking feature. This feature is enabled by default. This option is provided for historical reasons only and probably should not be used.

--no-force
Ordinarily automake creates all Makefile.ins mentioned in configure.ac. This option causes it to only update those Makefile.ins that are out of date with respect to one of their dependents.

-o dir
--output-dir=dir
Put the generated Makefile.in in the directory dir. Ordinarily each Makefile.in is created in the directory of the corresponding Makefile.am. This option is deprecated and will be removed in a future release.

-v
--verbose
Cause Automake to print information about which files are being read or created.

--version
Print the version number of Automake and exit.

-W CATEGORY
--warnings=category
Output warnings falling in category. category can be one of:
gnu warnings related to the GNU Coding Standards (see The GNU Coding Standards).
Chapter 6: Scanning configure.ac, using aclocal

Automake scans the package’s configure.ac to determine certain information about the package. Some autoconf macros are required and some variables must be defined in configure.ac. Automake will also use information from configure.ac to further tailor its output.

Automake also supplies some Autoconf macros to make the maintenance easier. These macros can automatically be put into your aclocal.m4 using the aclocal program.
6.1 Configuration requirements

The one real requirement of Automake is that your configure.ac call AM_INIT_AUTOMAKE. This macro does several things that are required for proper Automake operation (see Section 6.4 [Macros], page 44).

Here are the other macros that Automake requires but which are not run by AM_INIT_AUTOMAKE:

AC_CONFIG_FILES
AC_OUTPUT

These two macros are usually invoked as follows near the end of configure.ac.

...  
AC_CONFIG_FILES([  
  Makefile  
  doc/Makefile  
  src/Makefile  
  src/lib/Makefile  
  ...  
])  
AC_OUTPUT

Automake uses these to determine which files to create (see Section “Creating Output Files” in The Autoconf Manual). A listed file is considered to be an Automake generated Makefile if there exists a file with the same name and the .am extension appended. Typically, ‘AC_CONFIG_FILES([foo/Makefile])’ will cause Automake to generate foo/Makefile.in if foo/Makefile.am exists.

When using AC_CONFIG_FILES with multiple input files, as in

AC_CONFIG_FILES([Makefile:top.in:Makefile.in:bot.in])

automake will generate the first .in input file for which a .am file exists. If no such file exists the output file is not considered to be generated by Automake.

Files created by AC_CONFIG_FILES, be they Automake Makefiles or not, are all removed by ‘make distclean’. Their inputs are automatically distributed, unless they are the output of prior AC_CONFIG_FILES commands. Finally, rebuild rules are generated in the Automake Makefile existing in the subdirectory of the output file, if there is one, or in the top-level Makefile otherwise.

The above machinery (cleaning, distributing, and rebuilding) works fine if the AC_CONFIG_FILES specifications contain only literals. If part of the specification uses shell variables, automake will not be able to fulfill this setup, and you will have to complete the missing bits by hand. For instance, on

file=input  
...  
AC_CONFIG_FILES([output:$file],, [file=$file])

automake will output rules to clean output, and rebuild it. However the rebuild rule will not depend on input, and this file will not be distributed either. (You must add ‘EXTRA_DIST = input’ to your Makefile.am if input is a source file.) Similarly

file=output
file2=out:in
...
AC_CONFIG_FILES([$file:input],, [file=$file])
AC_CONFIG_FILES($file2),, [file2=$file2])

will only cause input to be distributed. No file will be cleaned automatically (add ‘DISTCLEANFILES = output out’ yourself), and no rebuild rule will be output.

Obviously automake cannot guess what value ‘$file’ is going to hold later when configure is run, and it cannot use the shell variable ‘$file’ in a Makefile. However, if you make reference to ‘$file’ as ‘${file}’ (i.e., in a way that is compatible with make’s syntax) and furthermore use AC_SUBST to ensure that ‘${file}’ is meaningful in a Makefile, then automake will be able to use ‘${file}’ to generate all of these rules. For instance, here is how the Automake package itself generates versioned scripts for its test suite:

```
AC_SUBST([APIVERSION], ...)
...
AC_CONFIG_FILES(
    [tests/aclocal-${APIVERSION}:tests/aclocal.in],
    [chmod +x tests/aclocal-${APIVERSION}],
    [APIVERSION=${APIVERSION}])
AC_CONFIG_FILES(
    [tests/automake-${APIVERSION}:tests/automake.in],
    [chmod +x tests/automake-${APIVERSION}])
```

Here cleaning, distributing, and rebuilding are done automatically, because ‘${APIVERSION}’ is known at make-time.

Note that you should not use shell variables to declare Makefile files for which automake must create Makefile.in. Even AC_SUBST does not help here, because automake needs to know the file name when it runs in order to check whether Makefile.am exists. (In the very hairy case that your setup requires such use of variables, you will have to tell Automake which Makefile.ins to generate on the command-line.)

It is possible to let automake emit conditional rules for AC_CONFIG_FILES with the help of AM_COND_IF (see Section 6.2 [Optional], page 31).

To summarize:

- Use literals for Makefiles, and for other files whenever possible.
- Use ‘$file’ (or ‘${file}’) without ‘AC_SUBST([file])’ for files that automake should ignore.
- Use ‘${file}’ and ‘AC_SUBST([file])’ for files that automake should not ignore.

### 6.2 Other things Automake recognizes

Every time Automake is run it calls Autoconf to trace configure.ac. This way it can recognize the use of certain macros and tailor the generated Makefile.in appropriately. Currently recognized macros and their effects are:
AC_CANONICAL_BUILD
AC_CANONICAL_HOST
AC_CANONICAL_TARGET
Automake will ensure that config.guess and config.sub exist. Also, the Makefile variables build_triplet, host_triplet and target_triplet are introduced. See Section “Getting the Canonical System Type” in The Autoconf Manual.

AC_CONFIG_AUX_DIR
Automake will look for various helper scripts, such as install-sh, in the directory named in this macro invocation. (The full list of scripts is: ar-lib, config.guess, config.sub, depcomp, elisp-comp, compile, install-sh, ltmain.sh, mdate-sh, missing, mkinstalldirs, py-compile, test-driver, texinfo.tex, ylwrap.) Not all scripts are always searched for; some scripts will only be sought if the generated Makefile.in requires them.
If AC_CONFIG_AUX_DIR is not given, the scripts are looked for in their standard locations. For mdate-sh, texinfo.tex, and ylwrap, the standard location is the source directory corresponding to the current Makefile.am. For the rest, the standard location is the first one of .., ..., or ../.. (relative to the top source directory) that provides any one of the helper scripts. See Section “Finding ‘configure’ Input” in The Autoconf Manual.
Required files from AC_CONFIG_AUX_DIR are automatically distributed, even if there is no Makefile.am in this directory.

AC_CONFIG_LIBOBJ_DIR
Automake will require the sources file declared with AC_LIBSOURCE (see below) in the directory specified by this macro.

AC_CONFIG_HEADERS
Automake will generate rules to rebuild these headers. Older versions of Automake required the use of AM_CONFIG_HEADER (see Section 6.4 [Macros], page 44); this is no longer the case.
As with AC_CONFIG_FILES (see Section 6.1 [Requirements], page 30), parts of the specification using shell variables will be ignored as far as cleaning, distributing, and rebuilding is concerned.

AC_CONFIG_LINKS
Automake will generate rules to remove configure generated links on ‘make distclean’ and to distribute named source files as part of ‘make dist’.
As for AC_CONFIG_FILES (see Section 6.1 [Requirements], page 30), parts of the specification using shell variables will be ignored as far as cleaning and distributing is concerned. (There are no rebuild rules for links.)

AC_LIBOBJ
AC_LIBSOURCE
AC_LIBSOURCES
Automake will automatically distribute any file listed in AC_LIBSOURCE or AC_LIBSOURCES.
Chapter 6: Scanning configure.ac, using aclocal

Note that the AC_LIBOBJ macro calls AC_LIBSOURCE. So if an Autoconf macro is documented to call ‘AC_LIBOBJ([file])’, then file.c will be distributed automatically by Automake. This encompasses many macros like AC_FUNC_ALLOC, AC_FUNC_MEMCMP, AC_REPLACE_FUNCS, and others.

By the way, direct assignments to LIBOBJS are no longer supported. You should always use AC_LIBOBJ for this purpose. See Section “AC_LIBOBJ vs. LIBOBJS” in The Autoconf Manual.

AC_PROG_RANLIB
This is required if any libraries are built in the package. See Section “Particular Program Checks” in The Autoconf Manual.

AC_PROG_CXX
This is required if any C++ source is included. See Section “Particular Program Checks” in The Autoconf Manual.

AC_PROG_OBJC
This is required if any Objective C source is included. See Section “Particular Program Checks” in The Autoconf Manual.

AC_PROG_OBJCXX
This is required if any Objective C++ source is included. See Section “Particular Program Checks” in The Autoconf Manual.

AC_PROG_F77
This is required if any Fortran 77 source is included. See Section “Particular Program Checks” in The Autoconf Manual.

AC_F77_LIBRARY_LDFLAGS
This is required for programs and shared libraries that are a mixture of languages that include Fortran 77 (see Section 8.14.3 [Mixing Fortran 77 With C and C++], page 77). See Section 6.4 [Autoconf macros supplied with Automake], page 44.

AC_FC_SRCEXT
Automake will add the flags computed by AC_FC_SRCEXT to compilation of files with the respective source extension (see Section “Fortran Compiler Characteristics” in The Autoconf Manual).

AC_PROG_FC
This is required if any Fortran 90/95 source is included. This macro is distributed with Autoconf version 2.58 and later. See Section “Particular Program Checks” in The Autoconf Manual.

AC_PROG_LIBTOOL
Automake will turn on processing for libtool (see Section “Introduction” in The Libtool Manual).

AC_PROG_YACC
If a Yacc source file is seen, then you must either use this macro or define the variable YACC in configure.ac. The former is preferred (see Section “Particular Program Checks” in The Autoconf Manual).
AC_PROG_LEX
If a Lex source file is seen, then this macro must be used. See Section “Particular Program Checks” in The Autoconf Manual.

AC_REQUIRE_AUX_FILE
For each AC_REQUIRE_AUX_FILE([file]), automake will ensure that file exists in the aux directory, and will complain otherwise. It will also automatically distribute the file. This macro should be used by third-party Autoconf macros that require some supporting files in the aux directory specified with AC_CONFIG_AUX_DIR above. See Section “Finding configure Input” in The Autoconf Manual.

AC_SUBST
The first argument is automatically defined as a variable in each generated Makefile.in, unless AM_SUBST_NOTMAKE is also used for this variable. See Section “Setting Output Variables” in The Autoconf Manual.

For every substituted variable var, automake will add a line var = value to each Makefile.in file. Many Autoconf macros invoke AC_SUBST to set output variables this way, e.g., AC_PATH_XTRA defines X_CFLAGS and X_LIBS. Thus, you can access these variables as $(X_CFLAGS) and $(X_LIBS) in any Makefile.am if AC_PATH_XTRA is called.

AM_CONDITIONAL
This introduces an Automake conditional (see Chapter 20 [Conditionals], page 124).

AM_COND_IF
This macro allows automake to detect subsequent access within configure.ac to a conditional previously introduced with AM_CONDITIONAL, thus enabling conditional AC_CONFIG_FILES (see Section 20.1 [Usage of Conditionals], page 124).

AM_GNU_GETTEXT
This macro is required for packages that use GNU gettext (see Section 10.2 [gettext], page 88). It is distributed with gettext. If Automake sees this macro it ensures that the package meets some of gettext’s requirements.

AM_GNU_GETTEXT_INTL_SUBDIR
This macro specifies that the intl/ subdirectory is to be built, even if the AM_GNU_GETTEXT macro was invoked with a first argument of ‘external’.

AM_MAINTAINER_MODE([default-mode])
This macro adds an --enable-maintainer-mode option to configure. If this is used, automake will cause “maintainer-only” rules to be turned off by default in the generated Makefile.ins, unless default-mode is ‘enable’. This macro defines the MAINTAINER_MODE conditional, which you can use in your own Makefile.am. See Section 28.2 [maintainer-mode], page 140.

AM_SUBST_NOTMAKE(var)
Prevent Automake from defining a variable var, even if it is substituted by config.status. Normally, Automake defines a make variable for each configure substitution, i.e., for each AC_SUBST([var]). This macro prevents that definition from Automake. If AC_SUBST has not been called
for this variable, then AM_SUBST_NOTMAKE has no effects. Preventing variable definitions may be useful for substitution of multi-line values, where \texttt{var = @value@} might yield unintended results.

\textbf{m4\_include}

Files included by \texttt{configure.ac} using this macro will be detected by Automake and automatically distributed. They will also appear as dependencies in Makefile rules.

\texttt{m4\_include} is seldom used by \texttt{configure.ac} authors, but can appear in \texttt{aclocal.m4} when \texttt{aclocal} detects that some required macros come from files local to your package (as opposed to macros installed in a system-wide directory, see Section 6.3 [aclocal Invocation], page 35).

### 6.3 Auto-generating aclocal.m4

Automake includes a number of Autoconf macros that can be used in your package (see Section 6.4 [Macros], page 44); some of them are actually required by Automake in certain situations. These macros must be defined in your \texttt{aclocal.m4}; otherwise they will not be seen by \texttt{autoconf}.

The \texttt{aclocal} program will automatically generate \texttt{aclocal.m4} files based on the contents of \texttt{configure.ac}. This provides a convenient way to get Automake-provided macros, without having to search around. The \texttt{aclocal} mechanism allows other packages to supply their own macros (see Section 6.3.3 [Extending aclocal], page 39). You can also use it to maintain your own set of custom macros (see Section 6.3.4 [Local Macros], page 41).

At startup, \texttt{aclocal} scans all the .\texttt{m4} files it can find, looking for macro definitions (see Section 6.3.2 [Macro Search Path], page 37). Then it scans \texttt{configure.ac}. Any mention of one of the macros found in the first step causes that macro, and any macros it in turn requires, to be put into \texttt{aclocal.m4}.

Putting the file that contains the macro definition into \texttt{aclocal.m4} is usually done by copying the entire text of this file, including unused macro definitions as well as both ‘#’ and ‘dnl’ comments. If you want to make a comment that will be completely ignored by \texttt{aclocal}, use ‘##’ as the comment leader.

When a file selected by \texttt{aclocal} is located in a subdirectory specified as a relative search path with \texttt{aclocal}'s \texttt{-I} argument, \texttt{aclocal} assumes the file belongs to the package and uses \texttt{m4\_include} instead of copying it into \texttt{aclocal.m4}. This makes the package smaller, eases dependency tracking, and cause the file to be distributed automatically. (See Section 6.3.4 [Local Macros], page 41, for an example.) Any macro that is found in a system-wide directory, or via an absolute search path will be copied. So use ‘-I \texttt{pwd}/reldir’ instead of ‘-I reldir’ whenever some relative directory should be considered outside the package.

The contents of \texttt{acinclude.m4}, if this file exists, are also automatically included in \texttt{aclocal.m4}. We recommend against using \texttt{acinclude.m4} in new packages (see Section 6.3.4 [Local Macros], page 41).

While computing \texttt{aclocal.m4}, \texttt{aclocal} runs \texttt{autom4te} (see Section “Using Autom4te” in The Autoconf Manual) in order to trace the macros that are really used, and omit from \texttt{aclocal.m4} all macros that are mentioned but otherwise unexpanded (this can happen when a macro is called conditionally). \texttt{autom4te} is expected to be in the \texttt{PATH}, just as \texttt{autoconf}. Its location can be overridden using the \texttt{AUTOM4TE} environment variable.
6.3.1 aclocal Options

aclocal accepts the following options:

--automake-acdir=dir
Look for the automake-provided macro files in dir instead of in the installation directory. This is typically used for debugging.

--system-acdir=dir
Look for the system-wide third-party macro files (and the special dirlist file) in dir instead of in the installation directory. This is typically used for debugging.

--diff[=command]
Run command on M4 file that would be installed or overwritten by --install. The default command is ‘diff -u’. This option implies --install and --dry-run.

--dry-run
Do not actually overwrite (or create) aclocal.m4 and M4 files installed by --install.

--help
Print a summary of the command line options and exit.

-I dir
Add the directory dir to the list of directories searched for .m4 files.

--install
Install system-wide third-party macros into the first directory specified with ‘-I dir’ instead of copying them in the output file. Note that this will happen also if dir is an absolute path.

When this option is used, and only when this option is used, aclocal will also honor ‘#serial number’ lines that appear in macros: an M4 file is ignored if there exists another M4 file with the same basename and a greater serial number in the search path (see Section 6.3.5 [Serials], page 42).

--force
Always overwrite the output file. The default is to overwrite the output file only when really needed, i.e., when its contents changes or if one of its dependencies is younger.

This option forces the update of aclocal.m4 (or the file specified with --output below) and only this file, it has absolutely no influence on files that may need to be installed by --install.

--output=file
Cause the output to be put into file instead of aclocal.m4.

--print-ac-dir
Prints the name of the directory that aclocal will search to find third-party .m4 files. When this option is given, normal processing is suppressed. This option was used in the past by third-party packages to determine where to install .m4 macro files, but this usage is today discouraged, since it causes ‘$(prefix)’ not to be thoroughly honoured (which violates the GNU Coding Standards), and a similar semantics can be better obtained with the ACLOCAL_PATH environment variable; see Section 6.3.3 [Extending aclocal], page 39.
--verbose
Print the names of the files it examines.

--version
Print the version number of Automake and exit.

-W CATEGORY
--warnings=category
Output warnings falling in category. category can be one of:
syntax  dubious syntactic constructs, underquoted macros, unused macros, etc.
unsupported unknown macros
all all the warnings, this is the default
none turn off all the warnings
error treat warnings as errors

All warnings are output by default.
The environment variable WARNINGS is honored in the same way as it is for automake (see Chapter 5 [automake Invocation], page 26).

6.3.2 Macro Search Path
By default, aclocal searches for .m4 files in the following directories, in this order:

acdir-APIVERSION
This is where the .m4 macros distributed with Automake itself are stored. APIVERSION depends on the Automake release used; for example, for Automake 1.11.x, $APIVERSION = 1.11$.

acdir This directory is intended for third party .m4 files, and is configured when automake itself is built. This is @datadir@/aclocal/, which typically expands to ${prefix}/share/aclocal/. To find the compiled-in value of acdir, use the --print-ac-dir option (see Section 6.3.1 [aclocal Options], page 36).

As an example, suppose that $automake-1.11.2$ was configured with $-p$ /[usr/local]. Then, the search path would be:
1. /usr/local/share/aclocal-1.11.2/
2. /usr/local/share/aclocal/

The paths for the acdir and acdir-APIVERSION directories can be changed respectively through aclocal options --system-acdir and --automake-acdir (see Section 6.3.1 [aclocal Options], page 36). Note however that these options are only intended for use by the internal Automake test suite, or for debugging under highly unusual situations; they are not ordinarily needed by end-users.

As explained in (see Section 6.3.1 [aclocal Options], page 36), there are several options that can be used to change or extend this search path.
Modifying the Macro Search Path: ‘-I dir’

Any extra directories specified using -I options (see Section 6.3.1 [aclocal Options], page 36) are prepended to this search list. Thus, `aclocal -I /foo -I /bar` results in the following search path:

1. /foo
2. /bar
3. acdir-APIVERSION
4. acdir

Modifying the Macro Search Path: dirlist

There is a third mechanism for customizing the search path. If a dirlist file exists in acdir, then that file is assumed to contain a list of directory patterns, one per line. aclocal expands these patterns to directory names, and adds them to the search list after all other directories. dirlist entries may use shell wildcards such as ‘*’, ‘?’, or [...]).

For example, suppose acdir/dirlist contains the following:

```
/test1
/test2
/test3*
```

and that aclocal was called with the ‘-I /foo -I /bar’ options. Then, the search path would be

1. /foo
2. /bar
3. acdir-APIVERSION
4. acdir
5. /test1
6. /test2

and all directories with path names starting with /test3.

If the --system-acdir=dir option is used, then aclocal will search for the dirlist file in dir; but remember the warnings above against the use of --system-acdir.

dirlist is useful in the following situation: suppose that automake version 1.11.2 is installed with ‘--prefix=/usr’ by the system vendor. Thus, the default search directories are

1. /usr/share/aclocal-1.11/
2. /usr/share/aclocal/

However, suppose further that many packages have been manually installed on the system, with $prefix=/usr/local, as is typical. In that case, many of these “extra” .m4 files are in /usr/local/share/aclocal. The only way to force /usr/bin/aclocal to find these “extra” .m4 files is to always call ‘aclocal -I /usr/local/share/aclocal’. This is inconvenient. With dirlist, one may create a file /usr/share/aclocal/dirlist containing only the single line

```
/usr/local/share/aclocal
```
Now, the “default” search path on the affected system is
1. /usr/share/aclocal-1.11/
2. /usr/share/aclocal/
3. /usr/local/share/aclocal/

without the need for -I options; -I options can be reserved for project-specific needs (my-source-dir/m4/), rather than using it to work around local system-dependent tool installation directories.

Similarly, dirlist can be handy if you have installed a local copy of Automake in your account and want aclocal to look for macros installed at other places on the system.

**Modifying the Macro Search Path: ACLOCAL_PATH**

The fourth and last mechanism to customize the macro search path is also the simplest. Any directory included in the colon-separated environment variable ACLOCAL_PATH is added to the search path and takes precedence over system directories (including those found via dirlist), with the exception of the versioned directory acdir-APIVERSION (see Section 6.3.2 [Macro Search Path], page 37). However, directories passed via -I will take precedence over directories in ACLOCAL_PATH.

Also note that, if the --install option is used, any .m4 file containing a required macro that is found in a directory listed in ACLOCAL_PATH will be installed locally. In this case, serial numbers in .m4 are honoured too, see Section 6.3.5 [Serials], page 42.

Conversely to dirlist, ACLOCAL_PATH is useful if you are using a global copy of Automake and want aclocal to look for macros somewhere under your home directory.

**Planned future incompatibilities**

The order in which the directories in the macro search path are currently looked up is confusing and/or suboptimal in various aspects, and is probably going to be changed in the future Automake release. In particular, directories in ACLOCAL_PATH and acdir might end up taking precedence over acdir-APIVERSION, and directories in acdir/dirlst might end up taking precedence over acdir. This is a possible future incompatibility!

**6.3.3 Writing your own aclocal macros**

The aclocal program doesn’t have any built-in knowledge of any macros, so it is easy to extend it with your own macros.

This can be used by libraries that want to supply their own Autoconf macros for use by other programs. For instance, the gettext library supplies a macro AM_GNU_GETTEXT that should be used by any package using gettext. When the library is installed, it installs this macro so that aclocal will find it.

A macro file’s name should end in .m4. Such files should be installed in $(datadir)/aclocal. This is as simple as writing:

```
aclocaldir = $(datadir)/aclocal
aclocal_DATA = mymacro.m4 myothermacro.m4
```

Please do use $(datadir)/aclocal, and not something based on the result of ‘aclocal --print-ac-dir’ (see Section 28.10 [Hard-Coded Install Paths], page 154, for arguments). It might also be helpful to suggest to the user to add the $(datadir)/aclocal directory
to his ACLOCAL_PATH variable (see [ACLOCAL_PATH], page 39) so that aclocal will find the .m4 files installed by your package automatically.

A file of macros should be a series of properly quoted AC_DEFUN’s (see Section “Macro Definitions” in The Autoconf Manual). The aclocal programs also understands AC_REQUIRE (see Section “Prerequisite Macros” in The Autoconf Manual), so it is safe to put each macro in a separate file. Each file should have no side effects but macro definitions. Especially, any call to AC_PREREQ should be done inside the defined macro, not at the beginning of the file.

Starting with Automake 1.8, aclocal will warn about all underquoted calls to AC_DEFUN. We realize this will annoy a lot of people, because aclocal was not so strict in the past and many third party macros are underquoted; and we have to apologize for this temporary inconvenience. The reason we have to be stricter is that a future implementation of aclocal (see Section 6.3.6 [Future of aclocal], page 44) will have to temporarily include all of these third party .m4 files, maybe several times, including even files that are not actually needed. Doing so should alleviate many problems of the current implementation, however it requires a stricter style from the macro authors. Hopefully it is easy to revise the existing macros. For instance,

```
# bad style
AC_PREREQ(2.68)
AC_DEFUN(AX_FOOBAR, [AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

should be rewritten as

```
AC_DEFUN([AX_FOOBAR], [AC_PREREQ([2.68])dnl
AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

Wrapping the AC_PREREQ call inside the macro ensures that Autoconf 2.68 will not be required if AX_FOOBAR is not actually used. Most importantly, quoting the first argument of AC_DEFUN allows the macro to be redefined or included twice (otherwise this first argument would be expanded during the second definition). For consistency we like to quote even arguments such as 2.68 that do not require it.

If you have been directed here by the aclocal diagnostic but are not the maintainer of the implicated macro, you will want to contact the maintainer of that macro. Please make sure you have the latest version of the macro and that the problem hasn’t already been reported before doing so: people tend to work faster when they aren’t flooded by mails.

Another situation where aclocal is commonly used is to manage macros that are used locally by the package, Section 6.3.4 [Local Macros], page 41.
6.3.4 Handling Local Macros

Feature tests offered by Autoconf do not cover all needs. People often have to supplement existing tests with their own macros, or with third-party macros.

There are two ways to organize custom macros in a package.

The first possibility (the historical practice) is to list all your macros in acinclude.m4. This file will be included in aclocal.m4 when you run aclocal, and its macro(s) will henceforth be visible to autoconf. However if it contains numerous macros, it will rapidly become difficult to maintain, and it will be almost impossible to share macros between packages.

The second possibility, which we do recommend, is to write each macro in its own file and gather all of these files in a directory. This directory is usually called m4/. To build aclocal.m4, one should therefore instruct aclocal to scan m4/. From the command line, this is done with ’aclocal -I m4’. The top-level Makefile.am should also be updated to define

\texttt{ACLOCAL_AMFLAGS = -I m4}

\texttt{ACLOCAL_AMFLAGS} contains options to pass to aclocal when aclocal.m4 is to be rebuilt by make. This line is also used by autoreconf (see Section “Using autoreconf to Update configure Scripts” in The Autoconf Manual) to run aclocal with suitable options, or by autopoint (see Section “Invoking the autopoint Program” in GNU gettext tools) and gettextize (see Section “Invoking the gettextize Program” in GNU gettext tools) to locate the place where Gettext’s macros should be installed. So even if you do not really care about the rebuild rules, you should define \texttt{ACLOCAL_AMFLAGS}.

When ‘aclocal -I m4’ is run, it will build an aclocal.m4 that m4_includes any file from m4/ that defines a required macro. Macros not found locally will still be searched in system-wide directories, as explained in Section 6.3.2 [Macro Search Path], page 37.

Custom macros should be distributed for the same reason that configure.ac is: so that other people have all the sources of your package if they want to work on it. Actually, this distribution happens automatically because all m4_included files are distributed.

However there is no consensus on the distribution of third-party macros that your package may use. Many libraries install their own macro in the system-wide aclocal directory (see Section 6.3.3 [Extending aclocal], page 39). For instance, Guile ships with a file called guile.m4 that contains the macro GUILE_FLAGS that can be used to define setup compiler and linker flags appropriate for using Guile. Using GUILE_FLAGS in configure.ac will cause aclocal to copy guile.m4 into aclocal.m4, but as guile.m4 is not part of the project, it will not be distributed. Technically, that means a user who needs to rebuild aclocal.m4 will have to install Guile first. This is probably OK, if Guile already is a requirement to build the package. However, if Guile is only an optional feature, or if your package might run on architectures where Guile cannot be installed, this requirement will hinder development. An easy solution is to copy such third-party macros in your local m4/ directory so they get distributed.

Since Automake 1.10, aclocal offers an option to copy these system-wide third-party macros in your local macro directory, solving the above problem. Simply use:

\texttt{ACLOCAL_AMFLAGS = -I m4 --install}
With this setup, system-wide macros will be copied to m4/ the first time you run autoreconf. Then the locally installed macros will have precedence over the system-wide installed macros each time aclocal is run again.

One reason why you should keep --install in the flags even after the first run is that when you later edit configure.ac and depend on a new macro, this macro will be installed in your m4/ automatically. Another one is that serial numbers (see Section 6.3.5 [Serials], page 42) can be used to update the macros in your source tree automatically when new system-wide versions are installed. A serial number should be a single line of the form

```
#serial nnn
```

where nnn contains only digits and dots. It should appear in the M4 file before any macro definition. It is a good practice to maintain a serial number for each macro you distribute, even if you do not use the --install option of aclocal: this allows other people to use it.

### 6.3.5 Serial Numbers

Because third-party macros defined in *.m4 files are naturally shared between multiple projects, some people like to version them. This makes it easier to tell which of two M4 files is newer. Since at least 1996, the tradition is to use a ‘#serial’ line for this.

A serial number should be a single line of the form

```
# serial version
```

where version is a version number containing only digits and dots. Usually people use a single integer, and they increment it each time they change the macro (hence the name of “serial”). Such a line should appear in the M4 file before any macro definition.

The ‘#’ must be the first character on the line, and it is OK to have extra words after the version, as in

```
#serial version garbage
```

Normally these serial numbers are completely ignored by aclocal and autoconf, like any genuine comment. However when using aclocal’s --install feature, these serial numbers will modify the way aclocal selects the macros to install in the package: if two files with the same basename exist in your search path, and if at least one of them uses a ‘#serial’ line, aclocal will ignore the file that has the older ‘#serial’ line (or the file that has none).

Note that a serial number applies to a whole M4 file, not to any macro it contains. A file can contains multiple macros, but only one serial.

Here is a use case that illustrates the use of --install and its interaction with serial numbers. Let’s assume we maintain a package called MyPackage, the configure.ac of which requires a third-party macro AX_THIRD_PARTY defined in /usr/share/aclocal/thirdparty.m4 as follows:

```
# serial 1
AC_DEFUN([AX_THIRD_PARTY], [...] )
```

MyPackage uses an m4/ directory to store local macros as explained in Section 6.3.4 [Local Macros], page 41, and has

```bash
ACLOCAL_AMFLAGS = -I m4 --install
```

in its top-level Makefile.am.
Initially the m4/ directory is empty. The first time we run autoreconf, it will fetch the options to pass to aclocal in Makefile.am, and run ‘aclocal -I m4 --install’. aclocal will notice that

- configure.ac uses AX_THIRD_PARTY
- No local macros define AX_THIRD_PARTY
- /usr/share/aclocal/thirdparty.m4 defines AX_THIRD_PARTY with serial 1.

Because /usr/share/aclocal/thirdparty.m4 is a system-wide macro and aclocal was given the --install option, it will copy this file in m4/thirdparty.m4, and output an aclocal.m4 that contains ‘m4_include([m4/thirdparty.m4])’.

The next time ‘aclocal -I m4 --install’ is run (either via autoreconf, by hand, or from the Makefile rebuild rules) something different happens. aclocal notices that

- configure.ac uses AX_THIRD_PARTY
- m4/thirdparty.m4 defines AX_THIRD_PARTY with serial 1.
- /usr/share/aclocal/thirdparty.m4 defines AX_THIRD_PARTY with serial 1.

Because both files have the same serial number, aclocal uses the first it found in its search path order (see Section 6.3.2 [Macro Search Path], page 37). aclocal therefore ignores /usr/share/aclocal/thirdparty.m4 and outputs an aclocal.m4 that contains ‘m4_include([m4/thirdparty.m4])’.

Local directories specified with -I are always searched before system-wide directories, so a local file will always be preferred to the system-wide file in case of equal serial numbers.

Now suppose the system-wide third-party macro is changed. This can happen if the package installing this macro is updated. Let’s suppose the new macro has serial number 2. The next time ‘aclocal -I m4 --install’ is run the situation is the following:

- configure.ac uses AX_THIRD_PARTY
- m4/thirdparty.m4 defines AX_THIRD_PARTY with serial 1.
- /usr/share/aclocal/thirdparty.m4 defines AX_THIRD_PARTY with serial 2.

When aclocal sees a greater serial number, it immediately forgets anything it knows from files that have the same basename and a smaller serial number. So after it has found /usr/share/aclocal/thirdparty.m4 with serial 2, aclocal will proceed as if it had never seen m4/thirdparty.m4. This brings us back to a situation similar to that at the beginning of our example, where no local file defined the macro. aclocal will install the new version of the macro in m4/thirdparty.m4, in this case overriding the old version. MyPackage just had its macro updated as a side effect of running aclocal.

If you are leery of letting aclocal update your local macro, you can run ‘aclocal -I m4 --diff’ to review the changes ‘aclocal -I m4 --install’ would perform on these macros.

Finally, note that the --force option of aclocal has absolutely no effect on the files installed by --install. For instance, if you have modified your local macros, do not expect --install --force to replace the local macros by their system-wide versions. If you want to do so, simply erase the local macros you want to revert, and run ‘aclocal -I m4 --install’.
6.3.6 The Future of aclocal

aclocal is expected to disappear. This feature really should not be offered by Automake. Automake should focus on generating Makefiles; dealing with M4 macros really is Autoconf’s job. The fact that some people install Automake just to use aclocal, but do not use automake otherwise is an indication of how that feature is misplaced.

The new implementation will probably be done slightly differently. For instance, it could enforce the m4/-style layout discussed in Section 6.3.4 [Local Macros], page 41.

We have no idea when and how this will happen. This has been discussed several times in the past, but someone still has to commit to that non-trivial task.

From the user point of view, aclocal’s removal might turn out to be painful. There is a simple precaution that you may take to make that switch more seamless: never call aclocal yourself. Keep this guy under the exclusive control of autoreconf and Automake’s rebuild rules. Hopefully you won’t need to worry about things breaking, when aclocal disappears, because everything will have been taken care of. If otherwise you used to call aclocal directly yourself or from some script, you will quickly notice the change.

Many packages come with a script called bootstrap.sh or autogen.sh, that will just call aclocal, libtoolize, gettextize or autopoint, autoconf, autoheader, and automake in the right order. Actually this is precisely what autoreconf can do for you. If your package has such a bootstrap.sh or autogen.sh script, consider using autoreconf. That should simplify its logic a lot (less things to maintain, yun!), it’s even likely you will not need the script anymore, and more to the point you will not call aclocal directly anymore.

For the time being, third-party packages should continue to install public macros into /usr/share/aclocal/. If aclocal is replaced by another tool it might make sense to rename the directory, but supporting /usr/share/aclocal/ for backward compatibility should be really easy provided all macros are properly written (see Section 6.3.3 [Extending aclocal], page 39).

6.4 Autoconf macros supplied with Automake

Automake ships with several Autoconf macros that you can use from your configure.ac. When you use one of them it will be included by aclocal in aclocal.m4.

6.4.1 Public Macros

\texttt{AM\_INIT\_AUTOMAKE([OPTIONS])}

Runs many macros required for proper operation of the generated Makefiles.

Today, \texttt{AM\_INIT\_AUTOMAKE} is called with a single argument: a space-separated list of Automake options that should be applied to every Makefile.am in the tree. The effect is as if each option were listed in AUTOMAKE\_OPTIONS (see Chapter 17 [Options], page 117).

This macro can also be called in another, deprecated form (support for which will be removed in the next major Automake release (1.13)): \texttt{AM\_INIT\_AUTOMAKE(PACKAGE, VERSION, [NO-DEFINE])}. In this form, there are two required arguments: the package and the version number. This form is obsolete because the package and version can be obtained from Autoconf’s AC\_INIT macro (which itself has an old and a new form).
If your `configure.ac` has:

```plaintext
AC_INIT([src/foo.c])
AM_INIT_AUTOMAKE([mumble], [1.5])
```

you should modernize it as follows:

```plaintext
AC_INIT([mumble], [1.5])
AC_CONFIG_SRCDIR([src/foo.c])
AM_INIT_AUTOMAKE
```

Note that if you’re upgrading your `configure.ac` from an earlier version of Automake, it is not always correct to simply move the package and version arguments from `AM_INIT_AUTOMAKE` directly to `AC_INIT`, as in the example above. The first argument to `AC_INIT` should be the name of your package (e.g., ‘GNU Automake’), not the tarball name (e.g., ‘automake’) that you used to pass to `AM_INIT_AUTOMAKE`. Autoconf tries to derive a tarball name from the package name, which should work for most but not all package names. (If it doesn’t work for yours, you can use the four-argument form of `AC_INIT` to provide the tarball name explicitly).

By default this macro `AC_DEFINE`’s `PACKAGE` and `VERSION`. This can be avoided by passing the `no-define` option:

```plaintext
AM_INIT_AUTOMAKE([gnits 1.5 no-define dist-bzip2])
```

`AM_PATH_LISPDIR`

Searches for the program `emacs`, and, if found, sets the output variable `lispdir` to the full path to Emacs’ site-lisp directory.

Note that this test assumes the `emacs` found to be a version that supports Emacs Lisp (such as GNU Emacs or XEmacs). Other emacs can cause this test to hang (some, like old versions of MicroEmacs, start up in interactive mode, requiring `C-x C-c` to exit, which is hardly obvious for a non-emacs user).

In most cases, however, you should be able to use `C-c` to kill the test. In order to avoid problems, you can set `EMACS` to “no” in the environment, or use the `--with-lispdir` option to `configure` to explicitly set the correct path (if you’re sure you have an `emacs` that supports Emacs Lisp).

`AM_PROG_AR([act-if-fail])`

You must use this macro when you use the archiver in your project, if you want support for unusual archivers such as Microsoft `lib`. The content of the optional argument is executed if the archiver interface is not recognized; the default action is to abort configure with an error message.

`AM_PROG_AS`

Use this macro when you have assembly code in your project. This will choose the assembler for you (by default the C compiler) and set `CCAS`, and will also set `CCASFLAGS` if required.

`AM_PROG_CC_C_O`

This is like `AC_PROG_CC_C_O`, but it generates its results in the manner required by Automake. You must use this instead of `AC_PROG_CC_C_O` when you need this functionality, that is, when using per-target flags or subdir-objects with C sources.
AM_PROG_LEX
Like AC_PROG_LEX (see Section “Particular Program Checks” in The Autoconf Manual), but uses the missing script on systems that do not have lex. HP-UX 10 is one such system.

AM_PROG_GCJ
This macro finds the gcj program or causes an error. It sets GCJ and GCJFLAGS. gcj is the Java front-end to the GNU Compiler Collection.

AM_PROG_UPC([compiler-search-list])
Find a compiler for Unified Parallel C and define the UPC variable. The default compiler-search-list is ‘upcc upc’. This macro will abort configure if no Unified Parallel C compiler is found.

AM_MISSING_PROG(name, program)
Find a maintainer tool program and define the name environment variable with its location. If program is not detected, then name will instead invoke the missing script, in order to give useful advice to the user about the missing maintainer tool. See Section 28.2 [maintainer-mode], page 140, for more information on when the missing script is appropriate.

AM_SILENT_RULES
Enable the machinery for less verbose build output (see Chapter 17 [Options], page 117).

AM_WITH_DMALLOC
Add support for the Dmalloc package (http://dmalloc.com/). If the user runs configure with --with-dmalloc, then define WITH_DMALLOC and add -ldmalloc to LIBS.

6.4.2 Obsolete Macros
Although using some of the following macros was required in past releases, you should not use any of them in new code. All these macros will be removed in the next major Automake version; if you are still using them, running autoupdate should adjust your configure.ac automatically (see Section “Using autoupdate to Modernize configure.ac” in The Autoconf Manual). Do it NOW!

AM_CONFIG_HEADER
Automake will generate rules to automatically regenerate the config header. This obsolete macro is a synonym of AC_CONFIG_HEADERS today (see Section 6.2 [Optional], page 31).

AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL
If the use of TIOCGWINSZ requires <sys/ioctl.h>, then define GWINSZ_IN_SYS_IOCTL. Otherwise TIOCGWINSZ can be found in <termios.h>. This macro is obsolete, you should use Autoconf’s AC_HEADER_TIOCGWINSZ instead.

AM_PROG_MKDIR_P
From Automake 1.8 to 1.9.6 this macro used to define the output variable mkdir_p to one of mkdir -p, install-sh -d, or mkinstalldirs. Nowadays Autoconf provides a similar functionality with AC_PROG_MKDIR_P (see Section “Particular Program Checks” in The Autoconf Manual), however this
defines the output variable `MKDIR_P` instead. In case you are still using the `AM_PROG_MKDIR_P` macro in your `configure.ac`, or its provided variable `${mkdir_p}` in your `Makefile.am`, you are advised to switch ASAP to the more modern Autoconf-provided interface instead; both the macro and the variable will be removed in the next major Automake release.

**AM_SYS_POSIX_TERMIOS**
Check to see if POSIX termios headers and functions are available on the system. If so, set the shell variable `am_cv_sys_posix_termios` to 'yes'. If not, set the variable to 'no'. This macro is obsolete, you should use Autoconf's `AC_SYS_POSIX_TERMIOS` instead.

### 6.4.3 Private Macros

The following macros are private macros you should not call directly. They are called by the other public macros when appropriate. Do not rely on them, as they might be changed in a future version. Consider them as implementation details; or better, do not consider them at all: skip this section!

- `_AM_DEPENDENCIES`
- `AM_SET_DEPDIR`
- `AM_DEP_TRACK`
- `AM_OUTPUT_DEPENDENCY_COMMANDS`
These macros are used to implement Automake’s automatic dependency tracking scheme. They are called automatically by Automake when required, and there should be no need to invoke them manually.

- `AM_MAKE_INCLUDE`
This macro is used to discover how the user’s `make` handles `include` statements. This macro is automatically invoked when needed; there should be no need to invoke it manually.

- `AM_PROG_INSTALL_STRIP`
This is used to find a version of `install` that can be used to strip a program at installation time. This macro is automatically included when required.

- `AM_SANITY_CHECK`
This checks to make sure that a file created in the build directory is newer than a file in the source directory. This can fail on systems where the clock is set incorrectly. This macro is automatically run from `AM_INIT_AUTOMAKE`.

### 7 Directories

For simple projects that distribute all files in the same directory it is enough to have a single `Makefile.am` that builds everything in place.

In larger projects, it is common to organize files in different directories, in a tree. For example, there could be a directory for the program’s source, one for the testsuite, and one for the documentation; or, for very large projects, there could be one directory per program, per library or per module.
The traditional approach is to build these subdirectories recursively, employing *make recursion*: each directory contains its own *Makefile*, and when *make* is run from the top-level directory, it enters each subdirectory in turn, and invokes there a new *make* instance to build the directory’s contents.

Because this approach is very widespread, Automake offers built-in support for it. However, it is worth noting that the use of make recursion has its own serious issues and drawbacks, and that it’s well possible to have packages with a multi directory layout that make little or no use of such recursion (examples of such packages are GNU Bison and GNU Automake itself); see also the Section 7.3 [Alternative], page 51, section below.

### 7.1 Recursing subdirectories

In packages using make recursion, the top level *Makefile.am* must tell Automake which subdirectories are to be built. This is done via the `SUBDIRS` variable.

The `SUBDIRS` variable holds a list of subdirectories in which building of various sorts can occur. The rules for many targets (e.g., `all`) in the generated *Makefile* will run commands both locally and in all specified subdirectories. Note that the directories listed in `SUBDIRS` are not required to contain *Makefile.am*s; only *Makefiles* (after configuration). This allows inclusion of libraries from packages that do not use Automake (such as *gettext*; see also Section 24.2 [Third-Party *Makefiles*], page 133).

In packages that use subdirectories, the top-level *Makefile.am* is often very short. For instance, here is the *Makefile.am* from the GNU Hello distribution:

```
EXTRA_DIST = BUGS ChangeLog.O README-alpha
SUBDIRS = doc intl po src tests
```

When Automake invokes *make* in a subdirectory, it uses the value of the `MAKE` variable. It passes the value of the variable `AM_MAKEFLAGS` to the *make* invocation; this can be set in *Makefile.am* if there are flags you must always pass to *make*.

The directories mentioned in `SUBDIRS` are usually direct children of the current directory, each subdirectory containing its own *Makefile.am* with a `SUBDIRS` pointing to deeper subdirectories. Automake can be used to construct packages of arbitrary depth this way.

By default, Automake generates *Makefiles* that work depth-first in postfix order: the subdirectories are built before the current directory. However, it is possible to change this ordering. You can do this by putting ‘.’ into `SUBDIRS`. For instance, putting ‘.’ first will cause a prefix ordering of directories.

Using

```
SUBDIRS = lib src . test
```

will cause `lib/` to be built before `src/`, then the current directory will be built, finally the `test/` directory will be built. It is customary to arrange test directories to be built after everything else since they are meant to test what has been constructed.

### 7.2 Conditional Subdirectories

It is possible to define the `SUBDIRS` variable conditionally if, like in the case of GNU Inetutils, you want to only build a subset of the entire package.

To illustrate how this works, let’s assume we have two directories `src/` and `opt/`. `src/` should always be built, but we want to decide in `configure` whether `opt/` will be built
or not. (For this example we will assume that opt/ should be built when the variable ‘$want_opt’ was set to ‘yes’.)

Running make should thus recurse into src/ always, and then maybe in opt/.

However ‘make dist’ should always recurse into both src/ and opt/. Because opt/ should be distributed even if it is not needed in the current configuration. This means opt/Makefile should be created unconditionally.

There are two ways to setup a project like this. You can use Automake conditionals (see Chapter 20 [Conditionals], page 124) or use Autoconf AC_SUBST variables (see Section “Setting Output Variables” in The Autoconf Manual). Using Automake conditionals is the preferred solution. Before we illustrate these two possibilities, let’s introduce DIST_SUBDIRS.

### 7.2.1 SUBDIRS vs. DIST_SUBDIRS

Automake considers two sets of directories, defined by the variables SUBDIRS and DIST_SUBDIRS.

**SUBDIRS** contains the subdirectories of the current directory that must be built (see Section 7.1 [Subdirectories], page 48). It must be defined manually; Automake will never guess a directory is to be built. As we will see in the next two sections, it is possible to define it conditionally so that some directory will be omitted from the build.

**DIST_SUBDIRS** is used in rules that need to recurse in all directories, even those that have been conditionally left out of the build. Recall our example where we may not want to build subdirectory opt/, but yet we want to distribute it? This is where DIST_SUBDIRS comes into play: ‘opt’ may not appear in SUBDIRS, but it must appear in DIST_SUBDIRS.

Precisely, DIST_SUBDIRS is used by ‘make maintainer-clean’, ‘make distclean’ and ‘make dist’. All other recursive rules use SUBDIRS.

If SUBDIRS is defined conditionally using Automake conditionals, Automake will define DIST_SUBDIRS automatically from the possible values of SUBDIRS in all conditions.

If SUBDIRS contains AC_SUBST variables, DIST_SUBDIRS will not be defined correctly because Automake does not know the possible values of these variables. In this case DIST_SUBDIRS needs to be defined manually.

### 7.2.2 Subdirectories with AM_CONDITIONAL

configure should output the Makefile for each directory and define a condition into which opt/ should be built.

...  
AM_CONDITIONAL([COND_OPT], [test "$want_opt" = yes])  
AC_CONFIG_FILES([Makefile src/Makefile opt/Makefile])  
...

Then SUBDIRS can be defined in the top-level Makefile.am as follows.

```bash
if COND_OPT  
  MAYBE_OPT = opt
endif
SUBDIRS = src $(MAYBE_OPT)
```

As you can see, running make will rightly recurse into src/ and maybe opt/.
As you can’t see, running `make dist` will recurse into both `src/` and `opt/` directories because `make dist`, unlike `make all`, doesn’t use the `SUBDIRS` variable. It uses the `DIST_SUBDIRS` variable.

In this case Automake will define `DIST_SUBDIRS = src opt` automatically because it knows that `MAYBE_OPT` can contain `opt` in some condition.

### 7.2.3 Subdirectories with AC_SUBST

Another possibility is to define `MAYBE_OPT` from `.configure` using `AC_SUBST`:

```bash
... if test "\$want_opt" = yes; then
  MAYBE_OPT=opt
else
  MAYBE_OPT=
fi
AC_SUBST([MAYBE_OPT])
AC_CONFIG_FILES([Makefile src/Makefile opt/Makefile])
...
```

In this case the top-level `Makefile.am` should look as follows.

```makefile
SUBDIRS = src $(MAYBE_OPT)
DIST_SUBDIRS = src opt
```

The drawback is that since Automake cannot guess what the possible values of `MAYBE_OPT` are, it is necessary to define `DIST_SUBDIRS`.

### 7.2.4 Unconfigured Subdirectories

The semantics of `DIST_SUBDIRS` are often misunderstood by some users that try to `configure` and `build` subdirectories conditionally. Here by configuring we mean creating the `Makefile` (it might also involve running a nested `configure` script: this is a costly operation that explains why people want to do it conditionally, but only the `Makefile` is relevant to the discussion).

The above examples all assume that every `Makefile` is created, even in directories that are not going to be built. The simple reason is that we want `make dist` to distribute even the directories that are not being built (e.g., platform-dependent code), hence `make dist` must recurse into the subdirectory, hence this directory must be configured and appear in `DIST_SUBDIRS`.

Building packages that do not configure every subdirectory is a tricky business, and we do not recommend it to the novice as it is easy to produce an incomplete tarball by mistake. We will not discuss this topic in depth here, yet for the adventurous here are a few rules to remember.
• **SUBDIRS** should always be a subset of **DIST_SUBDIRS**.

  It makes little sense to have a directory in **SUBDIRS** that is not in **DIST_SUBDIRS**. Think of the former as a way to tell which directories listed in the latter should be built.

• Any directory listed in **DIST_SUBDIRS** and **SUBDIRS** must be configured.

  I.e., the **Makefile** must exists or the recursive **make** rules will not be able to process the directory.

• Any configured directory must be listed in **DIST_SUBDIRS**.

  So that the cleaning rules remove the generated **Makefiles**. It would be correct to see **DIST_SUBDIRS** as a variable that lists all the directories that have been configured.

In order to prevent recursion in some unconfigured directory you must therefore ensure that this directory does not appear in **DIST_SUBDIRS** (and **SUBDIRS**). For instance, if you define **SUBDIRS** conditionally using **AC_SUBST** and do not define **DIST_SUBDIRS** explicitly, it will be default to ‘$(SUBDIRS)’; another possibility is to force **DIST_SUBDIRS = $(SUBDIRS)**.

Of course, directories that are omitted from **DIST_SUBDIRS** will not be distributed unless you make other arrangements for this to happen (for instance, always running ‘make dist’ in a configuration where all directories are known to appear in **DIST_SUBDIRS**; or writing a **dist-hook** target to distribute these directories).

In few packages, unconfigured directories are not even expected to be distributed. Although these packages do not require the aforementioned extra arrangements, there is another pitfall. If the name of a directory appears in **SUBDIRS** or **DIST_SUBDIRS**, **automake** will make sure the directory exists. Consequently **automake** cannot be run on such a distribution when one directory has been omitted. One way to avoid this check is to use the **AC_SUBST** method to declare conditional directories; since **automake** does not know the values of **AC_SUBST** variables it cannot ensure the corresponding directory exists.

### 7.3 An Alternative Approach to Subdirectories

If you’ve ever read Peter Miller’s excellent paper, Recursive Make Considered Harmful ([http://miller.emu.id.au/pmiller/books/rmch/](http://miller.emu.id.au/pmiller/books/rmch/)), the preceding sections on the use of make recursion will probably come as unwelcome advice. For those who haven’t read the paper, Miller’s main thesis is that recursive **make** invocations are both slow and error-prone.

Automake provides sufficient cross-directory support\(^2\) to enable you to write a single **Makefile.am** for a complex multi-directory package.

By default an installable file specified in a subdirectory will have its directory name stripped before installation. For instance, in this example, the header file will be installed as **$(includedir)/stdio.h**:

```
include_HEADERS = inc/stdio.h
```

However, the ‘nobase_’ prefix can be used to circumvent this path stripping. In this example, the header file will be installed as **$(includedir)/sys/types.h**:

```
nobase_include_HEADERS = sys/types.h
```

\(^2\) We believe. This work is new and there are probably warts. See Chapter 1 [Introduction], page 1, for information on reporting bugs.
'nobase_' should be specified first when used in conjunction with either 'dist_' or 'nodist_' (see Section 14.2 [Fine-grained Distribution Control], page 98). For instance:

```
nobase_dist_pkgdata_DATA = images/vortex.pgm sounds/whirl.ogg
```

Finally, note that a variable using the 'nobase_' prefix can often be replaced by several variables, one for each destination directory (see Section 3.3 [Uniform], page 20). For instance, the last example could be rewritten as follows:

```
imagesdir = $(pkgdatadir)/images
soundsdir = $(pkgdatadir)/sounds
dist_images_DATA = images/vortex.pgm
dist_sounds_DATA = sounds/whirl.ogg
```

This latter syntax makes it possible to change one destination directory without changing the layout of the source tree.

Currently, 'nobase_*_LTLIBRARIES' are the only exception to this rule, in that there is no particular installation order guarantee for an otherwise equivalent set of variables without 'nobase_' prefix.

### 7.4 Nesting Packages

In the GNU Build System, packages can be nested to arbitrary depth. This means that a package can embed other packages with their own configure, Makefiles, etc.

These other packages should just appear as subdirectories of their parent package. They must be listed in SUBDIRS like other ordinary directories. However the subpackage’s Makefiles should be output by its own configure script, not by the parent’s configure. This is achieved using the AC_CONFIG_SUBDIRS Autoconf macro (see Section “Configuring Other Packages in Subdirectories” in The Autoconf Manual).

Here is an example package for an arm program that links with a hand library that is a nested package in subdirectory hand/.

arm’s configure.ac:

```
AC_INIT([arm], [1.0])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
# Call hand’s ./configure script recursively.
AC_CONFIG_SUBDIRS([hand])
AC_OUTPUT
```

arm’s Makefile.am:

```
# Build the library in the hand subdirectory first.
SUBDIRS = hand

# Include hand’s header when compiling this directory.
AM_CPPFLAGS = -I$(srcdir)/hand

bin_PROGRAMS = arm
arm_SOURCES = arm.c
```
# link with the hand library.
arm_LDADD = hand/libhand.a

Now here is hand's hand/configure.ac:

AC_INIT([hand], [1.2])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CC
AM_PROG_AR
AC_PROG_RANLIB
AC_CONFIG_FILES([Makefile])
AC_OUTPUT

and its hand/Makefile.am:

lib_LIBRARIES = libhand.a
libhand_a_SOURCES = hand.c

When 'make dist' is run from the top-level directory it will create an archive arm-1.0.tar.gz that contains the arm code as well as the hand subdirectory. This package can be built and installed like any ordinary package, with the usual './configure && make && make install' sequence (the hand subpackage will be built and installed by the process).

When 'make dist' is run from the hand directory, it will create a self-contained hand-1.2.tar.gz archive. So although it appears to be embedded in another package, it can still be used separately.

The purpose of the 'AC_CONFIG_AUX_DIR([.])' instruction is to force Automake and Autoconf to search for auxiliary scripts in the current directory. For instance, this means that there will be two copies of install-sh: one in the top-level of the arm package, and another one in the hand/ subdirectory for the hand package.

The historical default is to search for these auxiliary scripts in the parent directory and the grandparent directory. So if the 'AC_CONFIG_AUX_DIR([.])' line was removed from hand/configure.ac, that subpackage would share the auxiliary script of the arm package. This may looks like a gain in size (a few kilobytes), but it is actually a loss of modularity as the hand subpackage is no longer self-contained ('make dist' in the subdirectory will not work anymore).

Packages that do not use Automake need more work to be integrated this way. See Section 24.2 [Third-Party Makefiles], page 133.

8 Building Programs and Libraries

A large part of Automake's functionality is dedicated to making it easy to build programs and libraries.

8.1 Building a program

In order to build a program, you need to tell Automake which sources are part of it, and which libraries it should be linked with.
This section also covers conditional compilation of sources or programs. Most of the comments about these also apply to libraries (see Section 8.2 [A Library], page 57) and libtool libraries (see Section 8.3 [A Shared Library], page 58).

### 8.1.1 Defining program sources

In a directory containing source that gets built into a program (as opposed to a library or a script), the `PROGRAMS` primary is used. Programs can be installed in `bindir`, `sbindir`, `libexecdir`, `pkglibexecdir`, or not at all (`noinst_`). They can also be built only for `make check`, in which case the prefix is `check_`.

For instance:

```
bin_PROGRAMS = hello
```

In this simple case, the resulting `Makefile.in` will contain code to generate a program named `hello`.

Associated with each program are several assisting variables that are named after the program. These variables are all optional, and have reasonable defaults. Each variable, its use, and default is spelled out below; we use the “hello” example throughout.

The variable `hello_SOURCES` is used to specify which source files get built into an executable:

```
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
```

This causes each mentioned `.c` file to be compiled into the corresponding `.o`. Then all are linked to produce `hello`.

If `hello_SOURCES` is not specified, then it defaults to the single file `hello.c` (see Section 8.5 [Default _SOURCES], page 68).

Multiple programs can be built in a single directory. Multiple programs can share a single source file, which must be listed in each `_SOURCES` definition.

Header files listed in a `_SOURCES` definition will be included in the distribution but otherwise ignored. In case it isn’t obvious, you should not include the header file generated by `configure` in a `_SOURCES` variable; this file should not be distributed. Lex (.l) and Yacc (.y) files can also be listed; see Section 8.8 [Yacc and Lex], page 72.

### 8.1.2 Linking the program

If you need to link against libraries that are not found by `configure`, you can use `LDADD` to do so. This variable is used to specify additional objects or libraries to link with; it is inappropriate for specifying specific linker flags, you should use `AM_LDFLAGS` for this purpose.

Sometimes, multiple programs are built in one directory but do not share the same link-time requirements. In this case, you can use the `prog_LDADD` variable (where prog is the name of the program as it appears in some `PROGRAMS` variable, and usually written in lowercase) to override `LDADD`. If this variable exists for a given program, then that program is not linked using `LDADD`.

For instance, in GNU `cpio`, `pax`, `cpio` and `mt` are linked against the library `libcpio.a`. However, `rmt` is built in the same directory, and has no such link requirement. Also, `mt` and
rmt are only built on certain architectures. Here is what cpio's `src/Makefile.am` looks like (abridged):

```plaintext
bin_PROGRAMS = cpio pax $(MT)
libexec_PROGRAMS = $(RMT)
EXTRA_PROGRAMS = mt rmt

LDADD = ../lib/libcpio.a $(INTLLIBS)
rmt_LDADD =

cpio_SOURCES = ...
pax_SOURCES = ...
mt_SOURCES = ...
rmt_SOURCES = ...

prog_LDADD is inappropriate for passing program-specific linker flags (except for -l, -L, -dlopen and -dlpreopen). So, use the prog_LDFLAGS variable for this purpose.

It is also occasionally useful to have a program depend on some other target that is not actually part of that program. This can be done using either the prog_DEPENDENCIES or the EXTRA_prog_DEPENDENCIES variable. Each program depends on the contents both variables, but no further interpretation is done.

Since these dependencies are associated to the link rule used to create the programs they should normally list files used by the link command. That is *.$(OBJEXT), *a, or *.la files. In rare cases you may need to add other kinds of files such as linker scripts, but listing a source file in _DEPENDENCIES is wrong. If some source file needs to be built before all the components of a program are built, consider using the BUILT_SOURCES variable instead (see Section 9.4 [Sources], page 84).

If prog_DEPENDENCIES is not supplied, it is computed by Automake. The automatically-assigned value is the contents of prog_LDADD, with most configure substitutions, -l, -L, -dlopen and -dlpreopen options removed. The configure substitutions that are left in are only '$(LIBOBJS)' and '$(ALLOCA)'; these are left because it is known that they will not cause an invalid value for prog_DEPENDENCIES to be generated.

Section 8.1.3 [Conditional Sources], page 55, shows a situation where _DEPENDENCIES may be used.

The EXTRA_prog_DEPENDENCIES may be useful for cases where you merely want to augment the automake-generated prog_DEPENDENCIES rather than replacing it.

We recommend that you avoid using -l options in LDADD or prog_LDADD when referring to libraries built by your package. Instead, write the file name of the library explicitly as in the above cpio example. Use -l only to list third-party libraries. If you follow this rule, the default value of prog_DEPENDENCIES will list all your local libraries and omit the other ones.

8.1.3 Conditional compilation of sources

You can't put a configure substitution (e.g., '@FOO@' or '$(FOO)' where FOO is defined via AC_SUBST) into a _SOURCES variable. The reason for this is a bit hard to explain, but suffice to say that it simply won't work. Automake will give an error if you try to do this.
Fortunately there are two other ways to achieve the same result. One is to use configure substitutions in `_LDADD` variables, the other is to use an Automake conditional.

**Conditional Compilation using `_LDADD` Substitutions**

Automake must know all the source files that could possibly go into a program, even if not all the files are built in every circumstance. Any files that are only conditionally built should be listed in the appropriate `EXTRA_` variable. For instance, if `hello-linux.c` or `hello-generic.c` were conditionally included in `hello`, the `Makefile.am` would contain:

```makefile
bin_PROGRAMS = hello
hello_SOURCES = hello-common.c
EXTRA_hello_SOURCES = hello-linux.c hello-generic.c
hello_LDADD = $(HELLO_SYSTEM)
hello_DEPENDENCIES = $(HELLO_SYSTEM)
```

You can then setup the `$(HELLO_SYSTEM)` substitution from `configure.ac`:

```makefile
... case $host in
  *linux*) HELLO_SYSTEM='hello-linux.$(OBJEXT)';
  *) HELLO_SYSTEM='hello-generic.$(OBJEXT)';
esac AC_SUBST([HELLO_SYSTEM])
...
```

In this case, the variable HELLO_SYSTEM should be replaced by either `hello-linux.o` or `hello-generic.o`, and added to both `hello_DEPENDENCIES` and `hello_LDADD` in order to be built and linked in.

**Conditional Compilation using Automake Conditionals**

An often simpler way to compile source files conditionally is to use Automake conditionals. For instance, you could use this `Makefile.am` construct to build the same `hello` example:

```makefile
bin_PROGRAMS = hello
if LINUX
  hello_SOURCES = hello-linux.c hello-common.c
else
  hello_SOURCES = hello-generic.c hello-common.c
endif
```

In this case, `configure.ac` should setup the LINUX conditional using `AM_CONDITIONAL` (see Chapter 20 [Conditionals], page 124).

When using conditionals like this you don’t need to use the `EXTRA_` variable, because Automake will examine the contents of each variable to construct the complete list of source files.

If your program uses a lot of files, you will probably prefer a conditional `+=`:

```makefile
bin_PROGRAMS = hello
hello_SOURCES = hello-common.c
if LINUX
  hello_SOURCES += hello-linux.c
else
```

hello_SOURCES += hello-generic.c
endif

8.1.4 Conditional compilation of programs

Sometimes it is useful to determine the programs that are to be built at configure time. For instance, GNU cpio only builds mt and rmt under special circumstances. The means to achieve conditional compilation of programs are the same you can use to compile source files conditionally: substitutions or conditionals.

Conditional Programs using configure Substitutions

In this case, you must notify Automake of all the programs that can possibly be built, but at the same time cause the generated Makefile.in to use the programs specified by configure. This is done by having configure substitute values into each _PROGRAMS definition, while listing all optionally built programs in EXTRA_PROGRAMS.

    bin_PROGRAMS = cpio pax $(MT)
    libexec_PROGRAMS = $(RMT)
    EXTRA_PROGRAMS = mt rmt

As explained in Section 8.20 [EXEEXT], page 81, Automake will rewrite bin_PROGRAMS, libexec_PROGRAMS, and EXTRA_PROGRAMS, appending ‘$(EXEEXT)’ to each binary. Obviously it cannot rewrite values obtained at run-time through configure substitutions, therefore you should take care of appending ‘$(EXEEXT)’ yourself, as in ‘AC_SUBST([MT], ['mt${EXEEXT}'])’.

Conditional Programs using Automake Conditionals

You can also use Automake conditionals (see Chapter 20 [Conditionals], page 124) to select programs to be built. In this case you don’t have to worry about ‘$(EXEEXT)’ or EXTRA_PROGRAMS.

    bin_PROGRAMS = cpio pax
    if WANT_MT
        bin_PROGRAMS += mt
    endif
    if WANT_RMT
        libexec_PROGRAMS = rmt
    endif

8.2 Building a library

Building a library is much like building a program. In this case, the name of the primary is LIBRARIES. Libraries can be installed in libdir or pkglibdir.

See Section 8.3 [A Shared Library], page 58, for information on how to build shared libraries using libtool and the LTLIBRARIES primary.

Each _LIBRARIES variable is a list of the libraries to be built. For instance, to create a library named libcpio.a, but not install it, you would write:

    noinst_LIBRARIES = libcpio.a
    libcpio_a_SOURCES = ...
Chapter 8: Building Programs and Libraries

The sources that go into a library are determined exactly as they are for programs, via the _SOURCES variables. Note that the library name is canonicalized (see Section 3.5 [Canonicalization], page 22), so the _SOURCES variable corresponding to libcpio.a is 'libcpio_a_SOURCES', not 'libcpio.a_SOURCES'.

Extra objects can be added to a library using the library_LIBADD variable. This should be used for objects determined by configure. Again from cpio:

```
libcpio_a_LIBADD = $(LIBOBJS) $(ALLOCA)
```

In addition, sources for extra objects that will not exist until configure-time must be added to the BUILT_SOURCES variable (see Section 9.4 [Sources], page 84).

Building a static library is done by compiling all object files, then by invoking `$(AR) $(ARFLAGS)` followed by the name of the library and the list of objects, and finally by calling `$RANLIB` on that library. You should call AC_PROG_RANLIB from your configure.ac to define RANLIB (Automake will complain otherwise). You should also call AM_PROG_AR to define AR, in order to support unusual archivers such as Microsoft lib. ARFLAGS will default to cru; you can override this variable by setting it in your Makefile.am or by AC_SUBSTing it from your configure.ac. You can override the AR variable by defining a per-library maude_AR variable (see Section 8.4 [Program and Library Variables], page 64).

Be careful when selecting library components conditionally. Because building an empty library is not portable, you should ensure that any library always contains at least one object.

To use a static library when building a program, add it to LDADD for this program. In the following example, the program cpio is statically linked with the library libcpio.a.

```
noinst_LIBRARIES = libcpio.a
libcpio_a_SOURCES = ...

bin_PROGRAMS = cpio
cpio_SOURCES = cpio.c ...
cpio_LDADD = libcpio.a
```

8.3 Building a Shared Library

Building shared libraries portably is a relatively complex matter. For this reason, GNU Libtool (see Section “Introduction” in The Libtool Manual) was created to help build shared libraries in a platform-independent way.

8.3.1 The Libtool Concept

Libtool abstracts shared and static libraries into a unified concept henceforth called libtool libraries. Libtool libraries are files using the .la suffix, and can designate a static library, a shared library, or maybe both. Their exact nature cannot be determined until ./configure is run: not all platforms support all kinds of libraries, and users can explicitly select which libraries should be built. (However the package’s maintainers can tune the default, see Section “The AC_PROG_LIBTOOL macro” in The Libtool Manual.)

Because object files for shared and static libraries must be compiled differently, libtool is also used during compilation. Object files built by libtool are called libtool objects: these are files using the .lo suffix. Libtool libraries are built from these libtool objects.
You should not assume anything about the structure of .la or .lo files and how libtool constructs them: this is libtool’s concern, and the last thing one wants is to learn about libtool’s guts. However the existence of these files matters, because they are used as targets and dependencies in Makefiles rules when building libtool libraries. There are situations where you may have to refer to these, for instance when expressing dependencies for building source files conditionally (see Section 8.3.4 [Conditional Libtool Sources], page 60).

People considering writing a plug-in system, with dynamically loaded modules, should look into libltdl: libtool’s dlopening library (see Section “Using libltdl” in The Libtool Manual). This offers a portable dlopening facility to load libtool libraries dynamically, and can also achieve static linking where unavoidable.

Before we discuss how to use libtool with Automake in details, it should be noted that the libtool manual also has a section about how to use Automake with libtool (see Section “Using Automake with Libtool” in The Libtool Manual).

8.3.2 Building Libtool Libraries

Automake uses libtool to build libraries declared with the LTLIBRARIES primary. Each _LTLIBRARIES variable is a list of libtool libraries to build. For instance, to create a libtool library named libgettext.la, and install it in libdir, write:

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c gettext.h ...
```

Automake predefines the variable pkglibdir, so you can use pkglib_LTLIBRARIES to install libraries in `$(libdir)/@PACKAGE@/`.

If gettext.h is a public header file that needs to be installed in order for people to use the library, it should be declared using a _HEADERS variable, not in libgettext_la_SOURCES. Headers listed in the latter should be internal headers that are not part of the public interface.

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c ...
include_HEADERS = gettext.h ...
```

A package can build and install such a library along with other programs that use it. This dependency should be specified using LDADD. The following example builds a program named hello that is linked with libgettext.la.

```
lib_LTLIBRARIES = libgettext.la
libgettext_la_SOURCES = gettext.c ...
bin_PROGRAMS = hello
hello_SOURCES = hello.c ...
hello_LDADD = libgettext.la
```

Whether hello is statically or dynamically linked with libgettext.la is not yet known: this will depend on the configuration of libtool and the capabilities of the host.

8.3.3 Building Libtool Libraries Conditionally

Like conditional programs (see Section 8.1.4 [Conditional Programs], page 57), there are two main ways to build conditional libraries: using Automake conditionals or using Autoconf AC_SUBSTiutions.
The important implementation detail you have to be aware of is that the place where a
library will be installed matters to libtool: it needs to be indicated at link-time using the
-rpath option.

For libraries whose destination directory is known when Automake runs, Automake will
automatically supply the appropriate -rpath option to libtool. This is the case for libraries
listed explicitly in some installable _LTLIBRARIES variables such as lib_LTLIBRARIES.

However, for libraries determined at configure time (and thus mentioned in EXTRA_-
LTLIBRARIES), Automake does not know the final installation directory. For such libraries
you must add the -rpath option to the appropriate _LDFLAGS variable by hand.

The examples below illustrate the differences between these two methods.

Here is an example where WANTEDLIBS is an AC_SUBSTed variable set at ./configure-
time to either libfoo.la, libbar.la, both, or none. Although ‘$(WANTEDLIBS)’ appears
in the lib_LTLIBRARIES, Automake cannot guess it relates to libfoo.la or libbar.la at
the time it creates the link rule for these two libraries. Therefore the -rpath argument
must be explicitly supplied.

EXTRA_LTLIBRARIES = libfoo.la libbar.la
lib_LTLIBRARIES = $(WANTEDLIBS)
libfoo_la_SOURCES = foo.c ...
libfoo_la_LDFLAGS = -rpath '$(libdir)'
libbar_la_SOURCES = bar.c ...
libbar_la_LDFLAGS = -rpath '$(libdir)'

Here is how the same Makefile.am would look using Automake conditionals named
WANT_LIBFOO and WANT_LIBBAR. Now Automake is able to compute the -rpath setting
itself, because it’s clear that both libraries will end up in ‘$(libdir)’ if they are installed.

lib_LTLIBRARIES =
if WANT_LIBFOO
  lib_LTLIBRARIES += libfoo.la
endif
if WANT_LIBBAR
  lib_LTLIBRARIES += libbar.la
endif
libfoo_la_SOURCES = foo.c ...
libbar_la_SOURCES = bar.c ...

8.3.4 Libtool Libraries with Conditional Sources

Conditional compilation of sources in a library can be achieved in the same way as condi-
tional compilation of sources in a program (see Section 8.1.3 [Conditional Sources], page 55).
The only difference is that _LIBADD should be used instead of _LDADD and that it should
mention libtool objects (.lo files).

So, to mimic the hello example from Section 8.1.3 [Conditional Sources], page 55, we
could build a libhello.la library using either hello-linux.c or hello-generic.c with
the following Makefile.am.

lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello-common.c
EXTRA_libhello_la_SOURCES = hello-linux.c hello-generic.c
libhello_la_LIBADD = $(HELLO_SYSTEM)
libhello_la_DEPENDENCIES = $(HELLO_SYSTEM)

And make sure configure defines HELLO_SYSTEM as either hello-linux.lo or hello-generic.lo.

Or we could simply use an Automake conditional as follows.

lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello-common.c
if LINUX
libhello_la_SOURCES += hello-linux.c
else
libhello_la_SOURCES += hello-generic.c
endif

8.3.5 Libtool Convenience Libraries

Sometimes you want to build libtool libraries that should not be installed. These are called libtool convenience libraries and are typically used to encapsulate many sublibraries, later gathered into one big installed library.

Libtool convenience libraries are declared by directory-less variables such as noinst_LTLIBRARIES, check_LTLIBRARIES, or even EXTRA_LTLIBRARIES. Unlike installed libtool libraries they do not need an -rpath flag at link time (actually this is the only difference).

Convenience libraries listed in noinst_LTLIBRARIES are always built. Those listed in check_LTLIBRARIES are built only upon 'make check'. Finally, libraries listed in EXTRA_LTLIBRARIES are never built explicitly: Automake outputs rules to build them, but if the library does not appear as a Makefile dependency anywhere it won’t be built (this is why EXTRA_LTLIBRARIES is used for conditional compilation).

Here is a sample setup merging libtool convenience libraries from subdirectories into one main libtop.la library.

# -- Top-level Makefile.am --
SUBDIRS = sub1 sub2 ... 
lib_LTLIBRARIES = libtop.la
libtop_la_SOURCES = 
libtop_la_LIBADD = \ 
  sub1/libsub1.la \ 
  sub2/libsub2.la \ 
  ...

# -- sub1/Makefile.am --
noinst_LTLIBRARIES = libsub1.la
libsub1_la_SOURCES = ...

# -- sub2/Makefile.am --
# showing nested convenience libraries
SUBDIRS = sub2.1 sub2.2 ... 
noinst_LTLIBRARIES = libsub2.la 
libsub2_la_SOURCES =
libsub2_la_LIBADD = \
sub21/libsub21.la \
sub22/libsub22.la \
...

When using such setup, beware that automake will assume libtop.la is to be linked with the C linker. This is because libtop_la_SOURCES is empty, so automake picks C as default language. If libtop_la_SOURCES was not empty, automake would select the linker as explained in Section 8.14.3.1 [How the Linker is Chosen], page 78.

If one of the sublibraries contains non-C source, it is important that the appropriate linker be chosen. One way to achieve this is to pretend that there is such a non-C file among the sources of the library, thus forcing automake to select the appropriate linker. Here is the top-level Makefile of our example updated to force C++ linking.

SUBDIRS = sub1 sub2 ...  
lib_LTLIBRARIES = libtop.la 
libtop_la_SOURCES = 
  # Dummy C++ source to cause C++ linking. 
  nodist_EXTRA_libtop_la_SOURCES = dummy.cxx 
libtop_la_LIBADD = \
  sub1/libsub1.la \
  sub2/libsub2.la \
...

'EXTRA_*.SOURCES' variables are used to keep track of source files that might be compiled (this is mostly useful when doing conditional compilation using AC_SUBST, see Section 8.3.4 [Conditional Libtool Sources], page 60), and the nodist_ prefix means the listed sources are not to be distributed (see Section 8.4 [Program and Library Variables], page 64). In effect the file dummy.cxx does not need to exist in the source tree. Of course if you have some real source file to list in libtop_la_SOURCES there is no point in cheating with nodist_EXTRA_libtop_la_SOURCES.

8.3.6 Libtool Modules
These are libtool libraries meant to be dlopened. They are indicated to libtool by passing -module at link-time.

pkglib_LTLIBRARIES = mymodule.la 
mymodule_la_SOURCES = doit.c 
mymodule_la_LDFLAGS = -module

Ordinarily, Automake requires that a library’s name start with lib. However, when building a dynamically loadable module you might wish to use a "nonstandard" name. Automake will not complain about such nonstandard names if it knows the library being built is a libtool module, i.e., if -module explicitly appears in the library’s _LDFLAGS variable (or in the common AM_LDFLAGS variable when no per-library _LDFLAGS variable is defined).

As always, AC_SUBST variables are black boxes to Automake since their values are not yet known when automake is run. Therefore if -module is set via such a variable, Automake cannot notice it and will proceed as if the library was an ordinary libtool library, with strict naming.
If `mymodule_la_SOURCES` is not specified, then it defaults to the single file `mymodule.c` (see Section 8.5 [Default _SOURCES], page 68).

### 8.3.7 _LIBADD, _LDFLAGS, and _LIBTOOLFLAGS

As shown in previous sections, the ‘library_LIBRARY_ADD’ variable should be used to list extra libtool objects (.lo files) or libtool libraries (.la) to add to library.

The ‘library_LIBRARY_FLAGS’ variable is the place to list additional libtool linking flags, such as -version-info, -static, and a lot more. See Section “Link mode” in The Libtool Manual.

The libtool command has two kinds of options: mode-specific options and generic options. Mode-specific options such as the aforementioned linking flags should be lumped with the other flags passed to the tool invoked by libtool (hence the use of ‘library_LIBRARY_FLAGS’ for libtool linking flags). Generic options include --tag=tag and --silent (see Section “Invoking libtool” in The Libtool Manual for more options) should appear before the mode selection on the command line; in Makefile.am they should be listed in the ‘library_LIBRARY_FLAGS’ variable.

If ‘library_LIBRARY_FLAGS’ is not defined, then the variable `AM_LIBRARY_FLAGS` is used instead.

These flags are passed to libtool after the --tag=tag option computed by Automake (if any), so ‘library_LIBRARY_FLAGS’ (or `AM_LIBRARY_FLAGS`) is a good place to override or supplement the --tag=tag setting.

The libtool rules also use a LIBTOOLFLAGS variable that should not be set in Makefile.am: this is a user variable (see Section 28.6 [Flag Variables Ordering], page 145. It allows users to run ‘make LIBTOOLFLAGS=--silent’, for instance. Note that the verbosity of libtool can also be influenced with the Automake silent-rules option (see Chapter 17 [Options], page 117).

### 8.3.8 LTLIBOBJS and LTALLOCA

Where an ordinary library might include ‘$(LIBOBSJS)’ or ‘$(ALLOCA)’ (see Section 8.6 [LIBOBSJS], page 69), a libtool library must use ‘$(LTLIBOBSJS)’ or ‘$(LTALLOCA)’. This is required because the object files that libtool operates on do not necessarily end in .o.

Nowadays, the computation of LTLIBOBSJS from LIBOBSJS is performed automatically by Autoconf (see Section “AC_LIBOBJ vs. LIBOBSJS” in The Autoconf Manual).

### 8.3.9 Common Issues Related to Libtool’s Use

#### 8.3.9.1 Error: ‘required file ‘./ltmain.sh’ not found’

Libtool comes with a tool called libtoolize that will install libtool’s supporting files into a package. Running this command will install ltmain.sh. You should execute it before aclocal and automake.

People upgrading old packages to newer autotools are likely to face this issue because older Automake versions used to call libtoolize. Therefore old build scripts do not call libtoolize.

Since Automake 1.6, it has been decided that running libtoolize was none of Automake’s business. Instead, that functionality has been moved into the autoreconf com-
mand (see Section “Using autoreconf” in The Autoconf Manual). If you do not want to remember what to run and when, just learn the autoreconf command. Hopefully, replacing existing bootstrap.sh or autogen.sh scripts by a call to autoreconf should also free you from any similar incompatible change in the future.

8.3.9.2 Objects ‘created with both libtool and without’

Sometimes, the same source file is used both to build a libtool library and to build another non-libtool target (be it a program or another library).

Let’s consider the following Makefile.am.

```makefile
bin_PROGRAMS = prog
prog_SOURCES = prog.c foo.c ...
lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.c ...
```

(In this trivial case the issue could be avoided by linking libfoo.la with prog instead of listing foo.c in prog_SOURCES. But let’s assume we really want to keep prog and libfoo.la separate.)

Technically, it means that we should build foo.$(OBJEXT) for prog, and foo.lo for libfoo.la. The problem is that in the course of creating foo.lo, libtool may erase (or replace) foo.$(OBJEXT), and this cannot be avoided.

Therefore, when Automake detects this situation it will complain with a message such as

```
object 'foo.$(OBJEXT)' created both with libtool and without
```

A workaround for this issue is to ensure that these two objects get different basenames.

As explained in Section 28.7 [Renamed Objects], page 147, this happens automatically when per-targets flags are used.

```makefile
bin_PROGRAMS = prog
prog_SOURCES = prog.c foo.c ...
prog_CFLAGS = $(AM_CFLAGS)
```

```
lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.c ...
```

Adding ‘prog_CFLAGS = $(AM_CFLAGS)’ is almost a no-op, because when the prog_CFLAGS is defined, it is used instead of AM_CFLAGS. However as a side effect it will cause prog.c and foo.c to be compiled as prog-prog.$(OBJEXT) and prog-foo.$(OBJEXT), which solves the issue.

8.4 Program and Library Variables

Associated with each program is a collection of variables that can be used to modify how that program is built. There is a similar list of such variables for each library. The canonical name of the program (or library) is used as a base for naming these variables.

In the list below, we use the name “maude” to refer to the program or library. In your Makefile.am you would replace this with the canonical name of your program. This list also refers to “maude” as a program, but in general the same rules apply for both static and
dynamic libraries; the documentation below notes situations where programs and libraries differ.

**maude\_SOURCES**

This variable, if it exists, lists all the source files that are compiled to build the program. These files are added to the distribution by default. When building the program, Automake will cause each source file to be compiled to a single `.o` file (or `.lo` when using libtool). Normally these object files are named after the source file, but other factors can change this. If a file in the \_SOURCES variable has an unrecognized extension, Automake will do one of two things with it. If a suffix rule exists for turning files with the unrecognized extension into `.o` files, then automake will treat this file as it will any other source file (see Section 8.18 [Support for Other Languages], page 80). Otherwise, the file will be ignored as though it were a header file.

The prefixes dist_ and nodist_ can be used to control whether files listed in a \_SOURCES variable are distributed. dist_ is redundant, as sources are distributed by default, but it can be specified for clarity if desired.

It is possible to have both dist_ and nodist_ variants of a given \_SOURCES variable at once; this lets you easily distribute some files and not others, for instance:

```latex
nodist_maude\_SOURCES = nodist.c
dist_maude\_SOURCES = dist-me.c
```

By default the output file (on Unix systems, the `.o` file) will be put into the current build directory. However, if the option subdir-objects is in effect in the current directory then the `.o` file will be put into the subdirectory named after the source file. For instance, with subdir-objects enabled, `sub/dir/file.c` will be compiled to `sub/dir/file.o`. Some people prefer this mode of operation. You can specify subdir-objects in AUTOMAKE_OPTIONS (see Chapter 17 [Options], page 117).

**EXTRA\_maude\_SOURCES**

Automake needs to know the list of files you intend to compile *statically*. For one thing, this is the only way Automake has of knowing what sort of language support a given `Makefile.in` requires.\(^3\) This means that, for example, you can’t put a configure substitution like `@my\_sources@` into a \_SOURCES variable. If you intend to conditionally compile source files and use configure to substitute the appropriate object names into, e.g., `\_LDADD` (see below), then you should list the corresponding source files in the EXTRA_ variable.

This variable also supports dist_ and nodist_ prefixes. For instance, nodist_ EXTRA\_maude\_SOURCES would list extra sources that may need to be built, but should not be distributed.

**maude\_AR**

A static library is created by default by invoking `\$\(\text{AR} \) \$\(\text{ARFLAGS}\)` followed by the name of the library and then the objects being put into the library. You can override this by setting the \_AR variable. This is usually used with C++; some C++ compilers require a special invocation in order to instantiate all the

---

\(^3\) There are other, more obscure reasons for this limitation as well.
templates that should go into a library. For instance, the SGI C++ compiler likes this variable set like so:

\[
\text{libmaude\_a\_AR} = $(CXX) -ar -o
\]

**maude\_LIBADD**

Extra objects can be added to a library using the \_LIBADD variable. For instance, this should be used for objects determined by configure (see Section 8.2 [A Library], page 57).

In the case of libtool libraries, **maude\_LIBADD** can also refer to other libtool libraries.

**maude\_LDADD**

Extra objects (*.\$(OBJEXT)) and libraries (*.a, *.la) can be added to a program by listing them in the \_LDADD variable. For instance, this should be used for objects determined by configure (see Section 8.1.2 [Linking], page 54).

\_LDADD and \_LIBADD are inappropriate for passing program-specific linker flags (except for -l, -L, -dlopen and -dlpreopen). Use the \_LDFLAGS variable for this purpose.

For instance, if your configure.ac uses AC\_PATH\_XTRA, you could link your program against the X libraries like so:

\[
\text{maude\_LDADD} = $(X\_PRE\_LIBS) $(X\_LIBS) $(X\_EXTRA\_LIBS)
\]

We recommend that you use -l and -L only when referring to third-party libraries, and give the explicit file names of any library built by your package. Doing so will ensure that **maude\_DEPENDENCIES** (see below) is correctly defined by default.

**maude\_LDFLAGS**

This variable is used to pass extra flags to the link step of a program or a shared library. It overrides the AM\_LDFLAGS variable.

**maude\_LIBTOOLFLAGS**

This variable is used to pass extra options to libtool. It overrides the AM\_LIBTOOLFLAGS variable. These options are output before libtool's \(--mode=mode\) option, so they should not be mode-specific options (those belong to the compiler or linker flags). See Section 8.3.7 [Libtool Flags], page 63.

**maude\_DEPENDENCIES**

It is also occasionally useful to have a target (program or library) depend on some other file that is not actually part of that target. This can be done using the \_DEPENDENCIES variable. Each target depends on the contents of such a variable, but no further interpretation is done.

Since these dependencies are associated to the link rule used to create the programs they should normally list files used by the link command. That is *.\$(OBJEXT), *.a, or *.la files for programs; *.lo and *.la files for Libtool libraries; and *.\$(OBJEXT) files for static libraries. In rare cases you may need to add other kinds of files such as linker scripts, but **listing a source file in \_DEPENDENCIES is wrong**. If some source file needs to be built before all the
components of a program are built, consider using the BUILT_SOURCES variable (see Section 9.4 [Sources], page 84).

If _DEPENDENCIES is not supplied, it is computed by Automake. The automatically-assigned value is the contents of _LDADD or _LIBADD, with most configure substitutions, -l, -L, -dlopen and -dlpreopen options removed. The configure substitutions that are left in are only '$(LIBOBS)' and '$(ALLOCA)'; these are left because it is known that they will not cause an invalid value for _DEPENDENCIES to be generated.

DEPENDENCIES is more likely used to perform conditional compilation using an AC_SUBST variable that contains a list of objects. See Section 8.1.3 [Conditional Sources], page 55, and Section 8.3.4 [Conditional Libtool Sources], page 60.

The EXTRA_*_DEPENDENCIES variable may be useful for cases where you merely want to augment the `automake`-generated _DEPENDENCIES variable rather than replacing it.

maude_LINK

You can override the linker on a per-program basis. By default the linker is chosen according to the languages used by the program. For instance, a program that includes C++ source code would use the C++ compiler to link. The _LINK variable must hold the name of a command that can be passed all the .o file names and libraries to link against as arguments. Note that the name of the underlying program is not passed to _LINK; typically one uses `$@':

\[
\text{maude\_LINK} = \$(CCLD) -magic -o $@
\]

If a _LINK variable is not supplied, it may still be generated and used by Automake due to the use of per-target link flags such as _CFLAGS, _LDFLAGS or _LIBTOOLFLAGS, in cases where they apply.

maude_CCASFLAGS
maude_CFLAGS
maude_CPPFLAGS
maude_CXXFLAGS
maude_FFLAGS
maude_GCJFLAGS
maude_LFLAGS
maude_OBJCFLAGS
maude_OBJCXXFLAGS
maude_RFLAGS
maude_UPCFLAGS
maude_YFLAGS

Automake allows you to set compilation flags on a per-program (or per-library) basis. A single source file can be included in several programs, and it will potentially be compiled with different flags for each program. This works for any language directly supported by Automake. These per-target compilation flags are '_CCASFLAGS', '_CFLAGS', '_CPPFLAGS', '_CXXFLAGS', '_FFLAGS', '_GCJFLAGS', '_LFLAGS', '_OBJCFLAGS', '_OBJCXXFLAGS', '_RFLAGS', '_UPCFLAGS', and '_YFLAGS'.
When using a per-target compilation flag, Automake will choose a different name for the intermediate object files. Ordinarily a file like `sample.c` will be compiled to produce `sample.o`. However, if the program’s `_CFLAGS` variable is set, then the object file will be named, for instance, `maude-sample.o`. (See also Section 28.7 [Renamed Objects], page 147.) The use of per-target compilation flags with C sources requires that the macro `AM_PROG_CC_C_O` be called from `configure.ac`.

In compilations with per-target flags, the ordinary ‘`AM_`’ form of the flags variable is not automatically included in the compilation (however, the user form of the variable is included). So for instance, if you want the hypothetical `maude` compilations to also use the value of `AM_CFLAGS`, you would need to write:

```
maude_CFLAGS = ... your flags ... $(AM_CFLAGS)
```

See Section 28.6 [Flag Variables Ordering], page 145, for more discussion about the interaction between user variables, ‘`AM_`’ shadow variables, and per-target variables.

**maude_SHORTNAME**

On some platforms the allowable file names are very short. In order to support these systems and per-target compilation flags at the same time, Automake allows you to set a ‘‘short name’’ that will influence how intermediate object files are named. For instance, in the following example,

```
bin_PROGRAMS = maude
maude_CPPFLAGS = -DSOMEFLAG
maude_SHORTNAME = m
maude_SOURCES = sample.c ...
```

the object file would be named `m-sample.o` rather than `maude-sample.o`. This facility is rarely needed in practice, and we recommend avoiding it until you find it is required.

### 8.5 Default _SOURCES

_SOURCES_ variables are used to specify source files of programs (see Section 8.1 [A Program], page 53), libraries (see Section 8.2 [A Library], page 57), and Libtool libraries (see Section 8.3 [A Shared Library], page 58).

When no such variable is specified for a target, Automake will define one itself. The default is to compile a single C file whose base name is the name of the target itself, with any extension replaced by `AM_DEFAULT_SOURCE_EXT`, which defaults to `.c`.

For example if you have the following somewhere in your `Makefile.am` with no corresponding `libfoo_a_SOURCES`:

```
lib_LIBRARIES = libfoo.a sub/libc++.a
```

`libfoo.a` will be built using a default source file named `libfoo.c`, and `sub/libc++.a` will be built from `sub/libc++.c`. (In older versions `sub/libc++.a` would be built from `sub_libc__a.c`, i.e., the default source was the canonized name of the target, with `.c` appended. We believe the new behavior is more sensible, but for backward compatibility _automake_ will use the old name if a file or a rule with that name exists and `AM_DEFAULT_SOURCE_EXT` is not used.)
Default sources are mainly useful in test suites, when building many test programs each from a single source. For instance, in

```c
check_PROGRAMS = test1 test2 test3
AM_DEFAULT_SOURCE_EXT = .cpp
```
test1, test2, and test3 will be built from test1.cpp, test2.cpp, and test3.cpp. Without the last line, they will be built from test1.c, test2.c, and test3.c.

Another case where this is convenient is building many Libtool modules (module\textunderscore n.la), each defined in its own file (module\textunderscore n.c).

```c
AM_LDFLAGS = -module
lib_LTLIBRARIES = module1.la module2.la module3.la
```

Finally, there is one situation where this default source computation needs to be avoided: when a target should not be built from sources. We already saw such an example in Section 4.2 [true], page 25; this happens when all the constituents of a target have already been compiled and just need to be combined using a \_LDADD variable. Then it is necessary to define an empty \_SOURCES variable, so that automake does not compute a default.

```c
bin_PROGRAMS = target
target\_SOURCES =
target\_LDADD = libmain.a libmisc.a
```

### 8.6 Special handling for LIBOBS and ALLOCA

The \texttt{\$(LIBOBS)} and \texttt{\$(ALLOCA)} variables list object files that should be compiled into the project to provide an implementation for functions that are missing or broken on the host system. They are substituted by configure.

These variables are defined by Autoconf macros such as \texttt{AC\_LIBOBJ}, \texttt{AC\_REPLACE\_FUNCS} (see Section “Generic Function Checks” in The Autoconf Manual), or \texttt{AC\_FUNC\_ALLOCA} (see Section “Particular Function Checks” in The Autoconf Manual). Many other Autoconf macros call \texttt{AC\_LIBOBJ} or \texttt{AC\_REPLACE\_FUNCS} to populate \texttt{\$(LIBOBS)}.

Using these variables is very similar to doing conditional compilation using \texttt{AC\_SUBST} variables, as described in Section 8.1.3 [Conditional Sources], page 55. That is, when building a program, \texttt{\$(LIBOBS)} and \texttt{\$(ALLOCA)} should be added to the associated \texttt{\_LDADD} variable, or to the \texttt{\_LIBADD} variable when building a library. However there is no need to list the corresponding sources in \texttt{\_EXTRA\_SOURCES} nor to define \texttt{\_DEPENDENCIES}. Automake automatically adds \texttt{\$(LIBOBS)} and \texttt{\$(ALLOCA)} to the dependencies, and it will discover the list of corresponding source files automatically (by tracing the invocations of the \texttt{AC\_LIBSOURCE} Autoconf macros). If you have already defined \texttt{\_DEPENDENCIES} explicitly for an unrelated reason, then you either need to add these variables manually, or use \texttt{\_EXTRA\_DEPENDENCIES} instead of \texttt{\_DEPENDENCIES}.

These variables are usually used to build a portability library that is linked with all the programs of the project. We now review a sample setup. First, configure.ac contains some checks that affect either LIBOBS or ALLOCA.

```c
# configure.ac
...
AC_CONFIG_LIBOBJ_DIR([lib])
...
```
AC_FUNC_MALLOC dnl May add malloc.$(OBJEXT) to LIBOBJS
AC_FUNC_MEMCMP dnl May add memcmp.$(OBJEXT) to LIBOBJS
AC_REPLACE_FUNCS([strdup]) dnl May add strdup.$(OBJEXT) to LIBOBJS
AC_FUNC_ALLOCA dnl May add alloca.$(OBJEXT) to ALLOCA
...
AC_CONFIG_FILES(
    lib/Makefile
    src/Makefile
)
AC_OUTPUT

The AC_CONFIG_LIBOBJ_DIR tells Autoconf that the source files of these object files are to be found in the lib/ directory. Automake can also use this information, otherwise it expects the source files are to be in the directory where the ‘$(LIBOBJS)’ and ‘$(ALLOCA)’ variables are used.

The lib/ directory should therefore contain malloc.c, memcmp.c, strdup.c, alloca.c. Here is its Makefile.am:

# lib/Makefile.am

noinst_LIBRARIES = libcompat.a
libcompat_a_SOURCES =
libcompat_a_LIBADD = $(LIBOBJS) $(ALLOCA)

The library can have any name, of course, and anyway it is not going to be installed: it just holds the replacement versions of the missing or broken functions so we can later link them in. Many projects also include extra functions, specific to the project, in that library: they are simply added on the _SOURCES line.

There is a small trap here, though: ‘$(LIBOBJS)’ and ‘$(ALLOCA)’ might be empty, and building an empty library is not portable. You should ensure that there is always something to put in libcompat.a. Most projects will also add some utility functions in that directory, and list them in libcompat_a_SOURCES, so in practice libcompat.a cannot be empty.

Finally here is how this library could be used from the src/ directory.

# src/Makefile.am

# Link all programs in this directory with libcompat.a
LDADD = ../lib/libcompat.a

bin_PROGRAMS = tool1 tool2 ...

tool1_SOURCES = ...
tool2_SOURCES = ...

When option subdir-objects is not used, as in the above example, the variables ‘$(LIBOBJS)’ or ‘$(ALLOCA)’ can only be used in the directory where their sources lie. E.g., here it would be wrong to use ‘$(LIBOBJS)’ or ‘$(ALLOCA)’ in src/Makefile.am. However if both subdir-objects and AC_CONFIG_LIBOBJ_DIR are used, it is OK to use these variables in other directories. For instance src/Makefile.am could be changed as follows.

# src/Makefile.am
Chapter 8: Building Programs and Libraries

AUTOMAKE_OPTIONS = subdir-objects
LDADD = $(LIBOBJS) $(ALLOCA)

bin_PROGRAMS = tool1 tool2 ...

tool1_SOURCES = ...
tool2_SOURCES = ...

Because ‘$(LIBOBJS)’ and ‘$(ALLOCA)’ contain object file names that end with ‘.$(OBJEXT)’, they are not suitable for Libtool libraries (where the expected object extension is .lo): LTLIBOBJS and LTALLOCA should be used instead.

LTLIBOBJS is defined automatically by Autoconf and should not be defined by hand (as in the past), however at the time of writing LTALLOCA still needs to be defined from ALLOCA manually. See Section “AC_LIBOBJ vs. LIBOBJS” in The Autoconf Manual.

8.7 Variables used when building a program

Occasionally it is useful to know which Makefile variables Automake uses for compilations, and in which order (see Section 28.6 [Flag Variables Ordering], page 145); for instance, you might need to do your own compilation in some special cases.

Some variables are inherited from Autoconf; these are CC, CFLAGS, CPPFLAGS, DEFS, LDFLAGS, and LIBS.

There are some additional variables that Automake defines on its own:

AM_CPPFLAGS

The contents of this variable are passed to every compilation that invokes the C preprocessor; it is a list of arguments to the preprocessor. For instance, -I and -D options should be listed here.

Automake already provides some -I options automatically, in a separate variable that is also passed to every compilation that invokes the C preprocessor. In particular it generates ‘-I.’, ‘-I$(srcdir)’, and a -I pointing to the directory holding config.h (if you’ve used AC_CONFIG_HEADERS or AM_CONFIG_HEADER). You can disable the default -I options using the nostdinc option.

When a file to be included is generated during the build and not part of a distribution tarball, its location is under $(builddir), not under $(srcdir). This matters especially for packages that use header files placed in sub-directories and want to allow builds outside the source tree (see Section 2.2.6 [VPATH Builds], page 6). In that case we recommend to use a pair of -I options, such as, e.g., ‘-I$(some/subdir) -I$(srcdir)/some/subdir’ or ‘-I$(top_builddir)/some/subdir -I$(top_srcdir)/some/subdir’. Note that the reference to the build tree should come before the reference to the source tree, so that accidentally leftover generated files in the source directory are ignored.

AM_CPPFLAGS is ignored in preference to a per-executable (or per-library) _CPPFLAGS variable if it is defined.
INCLUDES  This does the same job as AM_CPPFLAGS (or any per-target _CPPFLAGS variable if it is used). It is an older name for the same functionality. This variable is deprecated; we suggest using AM_CPPFLAGS and per-target _CPPFLAGS instead.

AM_CFLAGS  This is the variable the Makefile.am author can use to pass in additional C compiler flags. It is more fully documented elsewhere. In some situations, this is not used, in preference to the per-executable (or per-library) _CFLAGS.

COMPILE  This is the command used to actually compile a C source file. The file name is appended to form the complete command line.

AM_LDFLAGS  This is the variable the Makefile.am author can use to pass in additional linker flags. In some situations, this is not used, in preference to the per-executable (or per-library) _LDFLAGS.

LINK  This is the command used to actually link a C program. It already includes ‘-o $@’ and the usual variable references (for instance, CFLAGS); it takes as “arguments” the names of the object files and libraries to link in. This variable is not used when the linker is overridden with a per-target _LINK variable or per-target flags cause Automake to define such a _LINK variable.

8.8 Yacc and Lex support

Automake has somewhat idiosyncratic support for Yacc and Lex.

Automake assumes that the .c file generated by yacc (or lex) should be named using the basename of the input file. That is, for a yacc source file foo.y, Automake will cause the intermediate file to be named foo.c (as opposed to y.tab.c, which is more traditional).

The extension of a yacc source file is used to determine the extension of the resulting C or C++ source and header files. Note that header files are generated only when the -d Yacc option is used; see below for more information about this flag, and how to specify it. Files with the extension .y will thus be turned into .c sources and .h headers; likewise, .yy will become .cc and .hh, .y+ will become c++ and h++, .yxx will become .cxx and .hxx, and .ypp will become .cpp and .hpp.

Similarly, lex source files can be used to generate C or C++; the extensions .1, .11, .1++, .1xx, and .1pp are recognized.

You should never explicitly mention the intermediate (C or C++) file in any SOURCES variable; only list the source file.

The intermediate files generated by yacc (or lex) will be included in any distribution that is made. That way the user doesn’t need to have yacc or lex.

If a yacc source file is seen, then your configure.ac must define the variable YACC. This is most easily done by invoking the macro AC_PROG_YACC (see Section “Particular Program Checks” in The Autoconf Manual).

When yacc is invoked, it is passed AM_YFLAGS and YFLAGS. The latter is a user variable and the former is intended for the Makefile.am author.

AM_YFLAGS is usually used to pass the -d option to yacc. Automake knows what this means and will automatically adjust its rules to update and distribute the header file built...
by `yacc -d`. What Automake cannot guess, though, is where this header will be used: it is up to you to ensure the header gets built before it is first used. Typically this is necessary in order for dependency tracking to work when the header is included by another file. The common solution is listing the header file in `BUILT_SOURCES` (see Section 9.4 [Sources], page 84) as follows.

```plaintext
BUILT_SOURCES = parser.h
AM_YFLAGS = -d
bin_PROGRAMS = foo
foo_SOURCES = ... parser.y ...
```

If a `lex` source file is seen, then your `configure.ac` must define the variable `LEX`. You can use `AC_PROG_LEX` to do this (see Section “Particular Program Checks” in The Autoconf Manual), but using `AM_PROG_LEX` macro (see Section 6.4 [Macros], page 44) is recommended.

When `lex` is invoked, it is passed `AM_LFLAGS` and `LFLAGS`. The latter is a user variable and the former is intended for the `Makefile.am` author.

When `AM_MAINTAINER_MODE` (see Section 28.2 [maintainer-mode], page 140) is used, the rebuild rule for distributed Yacc and Lex sources are only used when `maintainer-mode` is enabled, or when the files have been erased.

When `lex` or `yacc` sources are used, `automake -i` automatically installs an auxiliary program called `ylwrap` in your package (see Section 3.7 [Auxiliary Programs], page 23). This program is used by the build rules to rename the output of these tools, and makes it possible to include multiple `yacc` (or `lex`) source files in a single directory. (This is necessary because yacc’s output file name is fixed, and a parallel make could conceivably invoke more than one instance of yacc simultaneously.)

For `yacc`, simply managing locking is insufficient. The output of `yacc` always uses the same symbol names internally, so it isn’t possible to link two `yacc` parsers into the same executable.

We recommend using the following renaming hack used in `gdb`:

```plaintext
#define yynmaxdepth c_maxdepth
#define yynparse c_parse
#define yynlex c_lex
#define yynerror c_error
#define yynlval c_lval
#define yychar c_char
#define yyndebug c_debug
#define yynpact c_pact
#define yyr1 c_r1
#define yyr2 c_r2
#define yyndef c_def
#define yynchk c_chk
#define yynpg o c_pgo
#define yynact c_act
#define yynexca c_exca
```

---

4 Please note that `automake` recognizes `-d` in `AM_YFLAGS` only if it is not clustered with other options; for example, it won’t be recognized if `AM_YFLAGS` is `-dt`, but it will be if `AM_YFLAGS` is `-d` `-t` or `-t -d`.
For each define, replace the 'c_' prefix with whatever you like. These defines work for bison, byacc, and traditional yaccs. If you find a parser generator that uses a symbol not covered here, please report the new name so it can be added to the list.

8.9 C++ Support

Automake includes full support for C++.

Any package including C++ code must define the output variable CXX in configure.ac; the simplest way to do this is to use the AC_PROG_CXX macro (see Section “Particular Program Checks” in The Autoconf Manual).

A few additional variables are defined when a C++ source file is seen:

CXX The name of the C++ compiler.
CXXFLAGS Any flags to pass to the C++ compiler.
AM_CXXFLAGS The maintainer’s variant of CXXFLAGS.
CXXCOMPILE The command used to actually compile a C++ source file. The file name is appended to form the complete command line.
CXXLINK The command used to actually link a C++ program.
8.10 Objective C Support

Automake includes some support for Objective C.

Any package including Objective C code must define the output variable OBJC in configure.ac; the simplest way to do this is to use the AC_PROG_OBJC macro (see Section "Particular Program Checks" in The Autoconf Manual).

A few additional variables are defined when an Objective C source file is seen:

- **OBJC**  
The name of the Objective C compiler.
- **OBJCFLAGS**  
Any flags to pass to the Objective C compiler.
- **AM_OBJCFLAGS**  
The maintainer’s variant of OBJCFLAGS.
- **OBJCCOMPILE**  
The command used to actually compile an Objective C source file. The file name is appended to form the complete command line.
- **OBJCLINK**  
The command used to actually link an Objective C program.

8.11 Objective C++ Support

Automake includes some support for Objective C++.

Any package including Objective C++ code must define the output variable OBJCXX in configure.ac; the simplest way to do this is to use the AC_PROG_OBJCXX macro (see Section "Particular Program Checks" in The Autoconf Manual).

A few additional variables are defined when an Objective C++ source file is seen:

- **OBJCXX**  
The name of the Objective C++ compiler.
- **OBJCXXFLAGS**  
Any flags to pass to the Objective C++ compiler.
- **AM_OBJCXXFLAGS**  
The maintainer’s variant of OBJCXXFLAGS.
- **OBJCXXCOMPILE**  
The command used to actually compile an Objective C++ source file. The file name is appended to form the complete command line.
- **OBJCXXLINK**  
The command used to actually link an Objective C++ program.

8.12 Unified Parallel C Support

Automake includes some support for Unified Parallel C.

Any package including Unified Parallel C code must define the output variable UPC in configure.ac; the simplest way to do this is to use the AM_PROG_UPC macro (see Section 6.4.1 [Public Macros], page 44).

A few additional variables are defined when a Unified Parallel C source file is seen:

- **UPC**  
The name of the Unified Parallel C compiler.
Chapter 8: Building Programs and Libraries

UPCFLAGS Any flags to pass to the Unified Parallel C compiler.

AM_UPCFLAGS
The maintainer’s variant of UPCFLAGS.

UPCCOMPILE
The command used to actually compile a Unified Parallel C source file. The file name is appended to form the complete command line.

UPCLINK The command used to actually link a Unified Parallel C program.

8.13 Assembly Support

Automake includes some support for assembly code. There are two forms of assembler files: normal (*.s) and preprocessed by CPP (*.S or *.sx).

The variable CCAS holds the name of the compiler used to build assembly code. This compiler must work a bit like a C compiler; in particular it must accept -c and -o. The values of CCASFLAGS and AM_CCASFLAGS (or its per-target definition) is passed to the compilation. For preprocessed files, DEFS, DEFAULT_INCLUDES, INCLUDES, CPPFLAGS and AM_CPPFLAGS are also used.

The autoconf macro AM_PROG_AS will define CCAS and CCASFLAGS for you (unless they are already set, it simply sets CCAS to the C compiler and CCASFLAGS to the C compiler flags), but you are free to define these variables by other means.

Only the suffixes .s, .S, and .sx are recognized by automake as being files containing assembly code.

8.14 Fortran 77 Support

Automake includes full support for Fortran 77.

Any package including Fortran 77 code must define the output variable F77 in configure.ac; the simplest way to do this is to use the AC_PROG_F77 macro (see Section “Particular Program Checks” in The Autoconf Manual).

A few additional variables are defined when a Fortran 77 source file is seen:

F77 The name of the Fortran 77 compiler.

FFLAGS Any flags to pass to the Fortran 77 compiler.

AM_FFLAGS
The maintainer’s variant of FFLAGS.

RFLAGS Any flags to pass to the Ratfor compiler.

AM_RFLAGS
The maintainer’s variant of RFLAGS.

F77COMPILE
The command used to actually compile a Fortran 77 source file. The file name is appended to form the complete command line.

FLINK The command used to actually link a pure Fortran 77 program or shared library.
Automake can handle preprocessing Fortran 77 and Ratfor source files in addition to compiling them\(^5\). Automake also contains some support for creating programs and shared libraries that are a mixture of Fortran 77 and other languages (see Section 8.14.3 [Mixing Fortran 77 With C and C++, page 77]). These issues are covered in the following sections.

### 8.14.1 Preprocessing Fortran 77

N.f is made automatically from N.F or N.r. This rule runs just the preprocessor to convert a preprocessable Fortran 77 or Ratfor source file into a strict Fortran 77 source file. The precise command used is as follows:

\[
\text{.F} \quad $(F77) \ -F \ $(DEFS) \ $(INCLUDES) \ $(AM\_CPPFLAGS) \ $(CPPFLAGS) \\
\quad $(AM\_FFLAGS) \ $(FFLAGS) \\
\text{.r} \quad $(F77) \ -F \ $(AM\_FFLAGS) \ $(FFLAGS) \ $(AM\_RFLAGS) \ $(RFLAGS)
\]

### 8.14.2 Compiling Fortran 77 Files

N.o is made automatically from N.f, N.F or N.r by running the Fortran 77 compiler. The precise command used is as follows:

\[
\text{.f} \quad $(F77) \ -c \ $(AM\_FFLAGS) \ $(FFLAGS) \\
\text{.F} \quad $(F77) \ -c \ $(DEFS) \ $(INCLUDES) \ $(AM\_CPPFLAGS) \ $(CPPFLAGS) \\
\quad $(AM\_FFLAGS) \ $(FFLAGS) \ $(AM\_RFLAGS) \ $(RFLAGS) \\
\text{.r} \quad $(F77) \ -c \ $(AM\_FFLAGS) \ $(FFLAGS) \ $(AM\_RFLAGS) \ $(RFLAGS)
\]

### 8.14.3 Mixing Fortran 77 With C and C++

Automake currently provides limited support for creating programs and shared libraries that are a mixture of Fortran 77 and C and/or C++. However, there are many other issues related to mixing Fortran 77 with other languages that are not (currently) handled by Automake, but that are handled by other packages\(^6\).

Automake can help in two ways:

1. Automatic selection of the linker depending on which combinations of source code.
2. Automatic selection of the appropriate linker flags (e.g., -L and -l) to pass to the automatically selected linker in order to link in the appropriate Fortran 77 intrinsic and run-time libraries.

These extra Fortran 77 linker flags are supplied in the output variable FLIBS by the \texttt{AC\_F77\_LIBRARY\_LDFLAGS} Autoconf macro. See Section “Fortran Compiler Characteristics” in The Autoconf Manual.

If Automake detects that a program or shared library (as mentioned in some \_PROGRAMS or \_LTLIBRARIES primary) contains source code that is a mixture of Fortran 77 and C and/or

\(^5\) Much, if not most, of the information in the following sections pertaining to preprocessing Fortran 77 programs was taken almost verbatim from Section “Catalogue of Rules” in The GNU Make Manual.

\(^6\) For example, the cfortran package (http://www-zeus.desy.de/~burow/cfortran/) addresses all of these inter-language issues, and runs under nearly all Fortran 77, C and C++ compilers on nearly all platforms. However, cfortran is not yet Free Software, but it will be in the next major release.
C++, then it requires that the macro `AC_F77_LIBRARY_LDFLAGS` be called in `configure.ac`, and that either `$(FLIBS)` appear in the appropriate `_LDADD` (for programs) or `_LIBADD` (for shared libraries) variables. It is the responsibility of the person writing the `Makefile.am` to make sure that `$(FLIBS)` appears in the appropriate `_LDADD` or `_LIBADD` variable.

For example, consider the following `Makefile.am`:

```makefile
bin_PROGRAMS = foo
foo_SOURCES = main.cc foo.f
foo_LDADD = libfoo.la $(FLIBS)

pkglib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = bar.f baz.c zardoz.cc
libfoo_la_LIBADD = $(FLIBS)
```

In this case, Automake will insist that `AC_F77_LIBRARY_LDFLAGS` is mentioned in `configure.ac`. Also, if `$(FLIBS)` hadn’t been mentioned in `foo_LDADD` and `libfoo_la_LIBADD`, then Automake would have issued a warning.

### 8.14.3.1 How the Linker is Chosen

When a program or library mixes several languages, Automake choose the linker according to the following priorities. (The names in parentheses are the variables containing the link command.)

1. Native Java (GCJLINK)
2. Objective C++ (OBJCXXLINK)
3. C++ (CXXLINK)
4. Fortran 77 (F77LINK)
5. Fortran (FCLINK)
6. Objective C (OBJCLINK)
7. Unified Parallel C (UPCLINK)
8. C (LINK)

For example, if Fortran 77, C and C++ source code is compiled into a program, then the C++ linker will be used. In this case, if the C or Fortran 77 linkers required any special libraries that weren’t included by the C++ linker, then they must be manually added to an _LDADD or _LIBADD variable by the user writing the `Makefile.am`.

Automake only looks at the file names listed in _SOURCES variables to choose the linker, and defaults to the C linker. Sometimes this is inconvenient because you are linking against a library written in another language and would like to set the linker more appropriately. See Section 8.3.5 [Libtool Convenience Libraries], page 61, for a trick with `nodist_EXTRA_ ..._SOURCES`.

A per-target _LINK variable will override the above selection. Per-target link flags will cause Automake to write a per-target _LINK variable according to the language chosen as above.
8.15 Fortran 9x Support

Automake includes support for Fortran 9x.

Any package including Fortran 9x code must define the output variable FC in configure.ac; the simplest way to do this is to use the AC_PROG_FC macro (see Section “Particular Program Checks” in The Autoconf Manual).

A few additional variables are defined when a Fortran 9x source file is seen:

FC The name of the Fortran 9x compiler.

FCFLAGS Any flags to pass to the Fortran 9x compiler.

AM_FCFLAGS The maintainer’s variant of FCFLAGS.

FCCOMPILE The command used to actually compile a Fortran 9x source file. The file name is appended to form the complete command line.

FCLINK The command used to actually link a pure Fortran 9x program or shared library.

8.15.1 Compiling Fortran 9x Files

file.o is made automatically from file.f90, file.f95, file.f03, or file.f08 by running the Fortran 9x compiler. The precise command used is as follows:

```
.f90 $(FC) $(AM_FCFLAGS) $(FCFLAGS) -c $(FCFLAGS_f90) $<
.f95 $(FC) $(AM_FCFLAGS) $(FCFLAGS) -c $(FCFLAGS_f95) $<
.f03 $(FC) $(AM_FCFLAGS) $(FCFLAGS) -c $(FCFLAGS_f03) $<
.f08 $(FC) $(AM_FCFLAGS) $(FCFLAGS) -c $(FCFLAGS_f08) $<
```

8.16 Compiling Java sources using gcj

Automake includes support for natively compiled Java, using gcj, the Java front end to the GNU Compiler Collection (rudimentary support for compiling Java to bytecode using the javac compiler is also present, albeit deprecated; see Section 10.4 [Java], page 88).

Any package including Java code to be compiled must define the output variable GCJ in configure.ac; the variable GCJFLAGS must also be defined somehow (either in configure.ac or Makefile.am). The simplest way to do this is to use the AM_PROG_GCJ macro.

By default, programs including Java source files are linked with gcj.

As always, the contents of AM_GCJFLAGS are passed to every compilation invoking gcj (in its role as an ahead-of-time compiler, when invoking it to create .class files, AM_JAVACFLAGS is used instead). If it is necessary to pass options to gcj from Makefile.am, this variable, and not the user variable GCJFLAGS, should be used.

gcj can be used to compile .java, .class, .zip, or .jar files.

When linking, gcj requires that the main class be specified using the --main= option. The easiest way to do this is to use the _LDFLAGS variable for the program.
8.17 Vala Support

Automake provides initial support for Vala (http://www.vala-project.org/). This requires valac version 0.7.0 or later, and currently requires the user to use GNU make.

```bash
foo_SOURCES = foo.vala bar.vala zardoc.c
```

Any .vala file listed in a _SOURCES variable will be compiled into C code by the Vala compiler. The generated .c files are distributed. The end user does not need to have a Vala compiler installed.

Automake ships with an Autoconf macro called AM_PROG_VALAC that will locate the Vala compiler and optionally check its version number.

```bash
AM_PROG_VALAC ([minimum-version]) [Macro]
```

Try to find a Vala compiler in PATH. If it is found, the variable VALAC is set. Optionally a minimum release number of the compiler can be requested:

```bash
AM_PROG_VALAC([0.7.0])
```

There are a few variables that are used when compiling Vala sources:

- **VALAC** Path to the Vala compiler.
- **VALAFLAGS** Additional arguments for the Vala compiler.
- **AM_VALAFLAGS** The maintainer’s variant of VALAFLAGS.

```bash
lib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES = foo.vala
```

Note that currently, you cannot use per-target *_VALAFLAGS (see Section 28.7 [Renamed Objects], page 147) to produce different C files from one Vala source file.

8.18 Support for Other Languages

Automake currently only includes full support for C, C++ (see Section 8.9 [C++ Support], page 74), Objective C (see Section 8.10 [Objective C Support], page 75), Objective C++ (see Section 8.11 [Objective C++ Support], page 75), Fortran 77 (see Section 8.14 [Fortran 77 Support], page 76), Fortran 9x (see Section 8.15 [Fortran 9x Support], page 79), and Java (see Section 8.16 [Java Support with gcj], page 79). There is only rudimentary support for other languages, support for which will be improved based on user demand.

Some limited support for adding your own languages is available via the suffix rule handling (see Section 18.2 [Suffixes], page 123).

8.19 Automatic dependency tracking

As a developer it is often painful to continually update the Makefile.am whenever the include-file dependencies change in a project. Automake supplies a way to automatically track dependency changes (see Section 2.2.12 [Dependency Tracking], page 11).

Automake always uses complete dependencies for a compilation, including system headers. Automake’s model is that dependency computation should be a side effect of the build.
To this end, dependencies are computed by running all compilations through a special wrapper program called `depcomp`. `depcomp` understands how to coax many different C and C++ compilers into generating dependency information in the format it requires. ‘automake -a’ will install `depcomp` into your source tree for you. If `depcomp` can’t figure out how to properly invoke your compiler, dependency tracking will simply be disabled for your build.

Experience with earlier versions of Automake (see Section “Dependency Tracking Evolution” in Brief History of Automake) taught us that it is not reliable to generate dependencies only on the maintainer’s system, as configurations vary too much. So instead Automake implements dependency tracking at build time.

Automatic dependency tracking can be suppressed by putting `no-dependencies` in the variable `AUTOMAKE_OPTIONS`, or passing `no-dependencies` as an argument to `AM_INIT_AUTOMAKE` (this should be the preferred way). Or, you can invoke `automake` with the `-i` option. Dependency tracking is enabled by default.

The person building your package also can choose to disable dependency tracking by configuring with `--disable-dependency-tracking`.

### 8.20 Support for executable extensions

On some platforms, such as Windows, executables are expected to have an extension such as `.exe`. On these platforms, some compilers (GCC among them) will automatically generate `foo.exe` when asked to generate `foo`.

Automake provides mostly-transparent support for this. Unfortunately mostly doesn’t yet mean fully. Until the English dictionary is revised, you will have to assist Automake if your package must support those platforms.

One thing you must be aware of is that, internally, Automake rewrites something like this:

```plaintext
bin_PROGRAMS = liver
```

to this:

```plaintext
bin_PROGRAMS = liver$(EXEEXT)
```

The targets Automake generates are likewise given the ‘$(EXEEXT)’ extension.

The variables `TESTS` and `XFAIL_TESTS` (see Section 15.2 [Simple Tests], page 102) are also rewritten if they contain filenames that have been declared as programs in the same `Makefile`. (This is mostly useful when some programs from `check_PROGRAMS` are listed in `TESTS`.)

However, Automake cannot apply this rewriting to `configure` substitutions. This means that if you are conditionally building a program using such a substitution, then your `configure.ac` must take care to add ‘$(EXEEXT)’ when constructing the output variable.

Sometimes maintainers like to write an explicit link rule for their program. Without executable extension support, this is easy—you simply write a rule whose target is the name of the program. However, when executable extension support is enabled, you must instead add the ‘$(EXEEXT)’ suffix.

This might be a nuisance for maintainers who know their package will never run on a platform that has executable extensions. For those maintainers, the `no-exeext` option (see Chapter 17 [Options], page 117) will disable this feature. This works in a fairly ugly way; if
no-exeext is seen, then the presence of a rule for a target named foo in Makefile.am will override an automake-generated rule for ‘foo$(EXEEXT)’. Without the no-exeext option, this use will give a diagnostic.

9 Other Derived Objects

Automake can handle derived objects that are not C programs. Sometimes the support for actually building such objects must be explicitly supplied, but Automake will still automatically handle installation and distribution.

9.1 Executable Scripts

It is possible to define and install programs that are scripts. Such programs are listed using the SCRIPTS primary name. When the script is distributed in its final, installable form, the Makefile usually looks as follows:

```
# Install my_script in $(bindir) and distribute it.
dist_bin_SCRIPTS = my_script
```

Scripts are not distributed by default; as we have just seen, those that should be distributed can be specified using a dist_ prefix as with other primaries.

Scripts can be installed in bindir, sbindir, libexecdir, pkglibexecdir, or pkgdatadir.

Scripts that need not be installed can be listed in noinst_SCRIPTS, and among them, those which are needed only by `make check` should go in check_SCRIPTS.

When a script needs to be built, the Makefile.am should include the appropriate rules. For instance the automake program itself is a Perl script that is generated from automake.in. Here is how this is handled:

```
bin_SCRIPTS = automake
CLEANFILES = $(bin_SCRIPTS)
EXTRA_DIST = automake.in

do_subst = sed -e 's,[@]datadir[@],$(datadir),g' \ 
    -e 's,[@]PERL[@],$(PERL),g' \ 
    -e 's,[@]PACKAGE[@],$(PACKAGE),g' \ 
    -e 's,[@]VERSION[@],$(VERSION),g' \ 
    ...

automake: automake.in Makefile
    $(do_subst) < $(srcdir)/automake.in > automake
    chmod +x automake
```

Such scripts for which a build rule has been supplied need to be deleted explicitly using CLEANFILES (see Chapter 13 [Clean], page 96), and their sources have to be distributed, usually with EXTRA_DIST (see Section 14.1 [Basics of Distribution], page 97).

Another common way to build scripts is to process them from configure with AC_CONFIG_FILES. In this situation Automake knows which files should be cleaned and distributed, and what the rebuild rules should look like.
For instance if `configure.ac` contains

```plaintext
AC_CONFIG_FILES([src/my_script], [chmod +x src/my_script])
```

to build `src/my_script` from `src/my_script.in`, then a `src/Makefile.am` to install this script in `$(_bindir)` can be as simple as

```plaintext
bin_SCRIPTS = my_script
CLEANFILES = $(bin_SCRIPTS)
```

There is no need for `EXTRA_DIST` or any build rule: Automake infers them from `AC_CONFIG_FILES` (see Section 6.1 [Requirements], page 30). `CLEANFILES` is still useful, because by default Automake will clean targets of `AC_CONFIG_FILES` in `distclean`, not `clean`.

Although this looks simpler, building scripts this way has one drawback: directory variables such as `$(_datadir)` are not fully expanded and may refer to other directory variables.

### 9.2 Header files

Header files that must be installed are specified by the `HEADERS` family of variables. Headers can be installed in `includedir`, `oldincludedir`, `pkgincludedir` or any other directory you may have defined (see Section 3.3 [Uniform], page 20). For instance,

```plaintext
include_HEADERS = foo.h bar/bar.h
```

will install the two files as `$(_includedir)/foo.h` and `$(_includedir)/bar.h`.

The `nobase_` prefix is also supported,

```plaintext
nobase_include_HEADERS = foo.h bar/bar.h
```

will install the two files as `$(_includedir)/foo.h` and `$(_includedir)/bar/bar.h` (see Section 7.3 [Alternative], page 51).

Usually, only header files that accompany installed libraries need to be installed. Headers used by programs or convenience libraries are not installed. The `noinst_HEADERS` variable can be used for such headers. However when the header actually belongs to a single convenience library or program, we recommend listing it in the program's or library's `_SOURCES` variable (see Section 8.1.1 [Program Sources], page 54) instead of in `noinst_HEADERS`. This is clearer for the `Makefile.am` reader. `noinst_HEADERS` would be the right variable to use in a directory containing only headers and no associated library or program.

All header files must be listed somewhere; in a `_SOURCES` variable or in a `_HEADERS` variable. Missing ones will not appear in the distribution.

For header files that are built and must not be distributed, use the `nodist_` prefix as in `nodist_include_HEADERS` or `nodist_prog_SOURCES`. If these generated headers are needed during the build, you must also ensure they exist before they are used (see Section 9.4 [Sources], page 84).

### 9.3 Architecture-independent data files

Automake supports the installation of miscellaneous data files using the `DATA` family of variables.

Such data can be installed in the directories `datadir`, `sysconfdir`, `sharedstatedir`, `localstatedir`, or `pkgdatadir`.

By default, data files are not included in a distribution. Of course, you can use the `dist_` prefix to change this on a per-variable basis.
Here is how Automake declares its auxiliary data files:

```
dist_pkgdata_DATA = clean-kr.am clean.am ...
```

## 9.4 Built Sources

Because Automake's automatic dependency tracking works as a side-effect of compilation (see Section 8.19 [Dependencies], page 80) there is a bootstrap issue: a target should not be compiled before its dependencies are made, but these dependencies are unknown until the target is first compiled.

Ordinarily this is not a problem, because dependencies are distributed sources: they preexist and do not need to be built. Suppose that `foo.c` includes `foo.h`. When it first compiles `foo.o`, `make` only knows that `foo.o` depends on `foo.c`. As a side-effect of this compilation `depcomp` records the `foo.h` dependency so that following invocations of `make` will honor it. In these conditions, it's clear there is no problem: either `foo.o` doesn't exist and has to be built (regardless of the dependencies), or accurate dependencies exist and they can be used to decide whether `foo.o` should be rebuilt.

It's a different story if `foo.h` doesn't exist by the first `make` run. For instance, there might be a rule to build `foo.h`. This time `file.o`'s build will fail because the compiler can't find `foo.h`. `make` failed to trigger the rule to build `foo.h` first by lack of dependency information.

The `BUILT_SOURCES` variable is a workaround for this problem. A source file listed in `BUILT_SOURCES` is made on `make all` or `make check` (or even `make install`) before other targets are processed. However, such a source file is not compiled unless explicitly requested by mentioning it in some other `_SOURCES` variable.

So, to conclude our introductory example, we could use `BUILT_SOURCES = foo.h` to ensure `foo.h` gets built before any other target (including `foo.o`) during `make all` or `make check`.

`BUILT_SOURCES` is actually a bit of a misnomer, as any file which must be created early in the build process can be listed in this variable. Moreover, all built sources do not necessarily have to be listed in `BUILT_SOURCES`. For instance, a generated `.c` file doesn't need to appear in `BUILT_SOURCES` (unless it is included by another source), because it's a known dependency of the associated object.

It might be important to emphasize that `BUILT_SOURCES` is honored only by `make all`, `make check` and `make install`. This means you cannot build a specific target (e.g., `make foo`) in a clean tree if it depends on a built source. However it will succeed if you have run `make all` earlier, because accurate dependencies are already available.

The next section illustrates and discusses the handling of built sources on a toy example.

### 9.4.1 Built Sources Example

Suppose that `foo.c` includes `bindir.h`, which is installation-dependent and not distributed: it needs to be built. Here `bindir.h` defines the preprocessor macro `bindir` to the value of the `make` variable `bindir` (inherited from `configure`).

We suggest several implementations below. It's not meant to be an exhaustive listing of all ways to handle built sources, but it will give you a few ideas if you encounter this issue.
First Try

This first implementation will illustrate the bootstrap issue mentioned in the previous section (see Section 9.4 [Sources], page 84).

Here is a tentative Makefile.am.

```plaintext
# This won't work.
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
    echo '#define bindir "$(bindir)"' >$@
```

This setup doesn't work, because Automake doesn't know that `foo.c` includes `bindir.h`. Remember, automatic dependency tracking works as a side-effect of compilation, so the dependencies of `foo.o` will be known only after `foo.o` has been compiled (see Section 8.19 [Dependencies], page 80). The symptom is as follows.

```
% make
source='foo.c' object='foo.o' libtool=no \
depfile='deps/foo.Po' tmpdepfile='deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c 'test -f 'foo.c' || echo './''foo.c
foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

In this example `bindir.h` is not distributed nor installed, and it is not even being built on-time. One may wonder if the `nodist_foo_SOURCES = bindir.h` line has any use at all. This line simply states that `bindir.h` is a source of `foo`, so for instance, it should be inspected while generating tags (see Section 18.1 [Tags], page 122). In other words, it does not help our present problem, and the build would fail identically without it.

Using BUILT_SOURCES

A solution is to require `bindir.h` to be built before anything else. This is what BUILT_SOURCES is meant for (see Section 9.4 [Sources], page 84).

```plaintext
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
BUILT_SOURCES = bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
    echo '#define bindir "$(bindir)"' >$@
```

See how `bindir.h` gets built first:

```
% make
    echo '#define bindir "/usr/local/bin"' >bindir.h
make all-am
make[1]: Entering directory `/home/adl/tmp'
source='foo.c' object='foo.o' libtool=no \
    depfile='deps/foo.Po' tmpdepfile='deps/foo.TPo' \
```
Chapter 9: Other Derived Objects

```plaintext
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c 'test -f 'foo.c' || echo './''foo.c 
gcc -g -O2 -o foo foo.o 
make[1]: Leaving directory '/home/adl/tmp'

However, as said earlier, BUILT_SOURCES applies only to the all, check, and install targets. It still fails if you try to run 'make foo' explicitly:

```%
make clean
test -z "bindir.h" || rm -f bindir.h
test -z "foo" || rm -f foo
rm -f *.o
% : > .deps/foo.Po # Suppress previously recorded dependencies
% make foo
source='foo.c' object='foo.o' libtool=no 

depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' 

depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c 'test -f 'foo.c' || echo './'"''foo.c 

foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

Recording Dependencies manually

Usually people are happy enough with BUILT_SOURCES because they never build targets such as 'make foo' before 'make all', as in the previous example. However if this matters to you, you can avoid BUILT_SOURCES and record such dependencies explicitly in the Makefile.am.

```plaintext
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
foo.$(OBJEXT): bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
    echo '#define bindir "$(bindir)"' >$@
```

You don't have to list all the dependencies of foo.o explicitly, only those that might need to be built. If a dependency already exists, it will not hinder the first compilation and will be recorded by the normal dependency tracking code. (Note that after this first compilation the dependency tracking code will also have recorded the dependency between foo.o and bindir.h; so our explicit dependency is really useful to the first build only.)

Adding explicit dependencies like this can be a bit dangerous if you are not careful enough. This is due to the way Automake tries not to overwrite your rules (it assumes you know better than it). ‘foo.$(OBJEXT): bindir.h’ supersedes any rule Automake may want to output to build ‘foo.$(OBJEXT)’. It happens to work in this case because Automake doesn't have to output any ‘foo.$(OBJEXT)’ target: it relies on a suffix rule instead (i.e., ‘.c.$(OBJEXT)’). Always check the generated Makefile.in if you do this.

Build bindir.h from configure

It's possible to define this preprocessor macro from configure, either in config.h (see Section “Defining Directories” in The Autoconf Manual), or by processing a bindir.h.in file using AC_CONFIG_FILES (see Section “Configuration Actions” in The Autoconf Manual).
At this point it should be clear that building bindir.h from configure works well for this example. bindir.h will exist before you build any target, hence will not cause any dependency issue.

The Makefile can be shrunk as follows. We do not even have to mention bindir.h.

```makefile
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

However, it’s not always possible to build sources from configure, especially when these sources are generated by a tool that needs to be built first.

**Build bindir.c, not bindir.h.**

Another attractive idea is to define bindir as a variable or function exported from bindir.o, and build bindir.c instead of bindir.h.

```makefile
noinst_PROGRAMS = foo
foo_SOURCES = foo.c bindir.h
nodist_foo_SOURCES = bindir.c
CLEANFILES = bindir.c

bindir.c: Makefile
    echo 'const char bindir[] = "$(bindir)";'; > bindir.c

bindir.h contains just the variable’s declaration and doesn’t need to be built, so it won’t cause any trouble. bindir.o is always dependent on bindir.c, so bindir.c will get built first.

**Which is best?**

There is no panacea, of course. Each solution has its merits and drawbacks.

You cannot use BUILT_SOURCES if the ability to run ‘make foo’ on a clean tree is important to you.

You won’t add explicit dependencies if you are leery of overriding an Automake rule by mistake.

Building files from ./configure is not always possible, neither is converting .h files into .c files.

**10 Other GNU Tools**

Since Automake is primarily intended to generate Makefile.ins for use in GNU programs, it tries hard to interoperate with other GNU tools.

**10.1 Emacs Lisp**

Automake provides some support for Emacs Lisp. The LISP primary is used to hold a list of .el files. Possible prefixes for this primary are lisp_ and noinst_. Note that if lisp_.LISP is defined, then configure.ac must run AM_PATH_LISPDIR (see Section 6.4 [Macros], page 44).

Lisp sources are not distributed by default. You can prefix the LISP primary with dist_, as in dist_lisp_LISP or dist_noinst_LISP, to indicate that these files should be distributed.
Automake will byte-compile all Emacs Lisp source files using the Emacs found by \texttt{AM_PATH_LISPDIR}, if any was found.

Byte-compiled Emacs Lisp files are not portable among all versions of Emacs, so it makes sense to turn this off if you expect sites to have more than one version of Emacs installed. Furthermore, many packages don’t actually benefit from byte-compilation. Still, we recommend that you byte-compile your Emacs Lisp sources. It is probably better for sites with strange setups to cope for themselves than to make the installation less nice for everybody else.

There are two ways to avoid byte-compiling. Historically, we have recommended the following construct.

\begin{verbatim}
lisp_LISP = file1.el file2.el
ELCFILES =
\end{verbatim}

\texttt{ELCFILES} is an internal Automake variable that normally lists all .elc files that must be byte-compiled. Automake defines \texttt{ELCFILES} automatically from \texttt{lisp_LISP}. Emptying this variable explicitly prevents byte-compilation.

Since Automake 1.8, we now recommend using \texttt{lisp_DATA} instead:

\begin{verbatim}
lisp_DATA = file1.el file2.el
\end{verbatim}

Note that these two constructs are not equivalent. \_LISP will not install a file if Emacs is not installed, while \_DATA will always install its files.

\section*{10.2 Gettext}

If \texttt{AM_GNU_GETTEXT} is seen in \texttt{configure.ac}, then Automake turns on support for GNU gettext, a message catalog system for internationalization (see Section "Introduction" in \texttt{GNU gettext utilities}).

The \texttt{gettext} support in Automake requires the addition of one or two subdirectories to the package: \texttt{po} and possibly also \texttt{intl}. The latter is needed if \texttt{AM_GNU_GETTEXT} is not invoked with the 'external' argument, or if \texttt{AM_GNU_GETTEXT_INTL_SUBDIR} is used. Automake ensures that these directories exist and are mentioned in \texttt{SUBDIRS}.

\section*{10.3 Libtool}

Automake provides support for GNU Libtool (see Section "Introduction" in \texttt{The Libtool Manual}) with the \texttt{LTLIBRARIES} primary. See Section 8.3 [A Shared Library], page 58.

\section*{10.4 Java bytecode compilation (deprecated)}

Automake provides some minimal support for Java bytecode compilation with the \texttt{JAVA} primary (in addition to the support for compiling Java to native machine code; see Section 8.16 [Java Support with gcj], page 79). Note however that the interface and most features described here are deprecated; the next automake release will strive to provide a better and cleaner interface, which however won’t be backward-compatible; the present interface will probably be removed altogether in future automake releases (1.13 or later), so don’t use it in new code.

Any .java files listed in a \_JAVA variable will be compiled with JAVAC at build time. By default, .java files are not included in the distribution, you should use the dist_ prefix to distribute them.
Here is a typical setup for distributing .java files and installing the .class files resulting from their compilation.

```
javadir = $(datadir)/java
dist_java_JAVA = a.java b.java ...
```

Currently Automake enforces the restriction that only one _JAVA primary can be used in a given Makefile.am. The reason for this restriction is that, in general, it isn’t possible to know which .class files were generated from which .java files, so it would be impossible to know which files to install where. For instance, a .java file can define multiple classes; the resulting .class file names cannot be predicted without parsing the .java file.

There are a few variables that are used when compiling Java sources:

- **JAVAC**  
  The name of the Java compiler. This defaults to ‘javac’.

- **JAVACFLAGS**  
  The flags to pass to the compiler. This is considered to be a user variable (see Section 3.6 [User Variables], page 23).

- **AM_JAVACFLAGS**  
  More flags to pass to the Java compiler. This, and not JAVACFLAGS, should be used when it is necessary to put Java compiler flags into Makefile.am.

- **JAVAROOT**  
  The value of this variable is passed to the -d option to javac. It defaults to ‘$(top_builddir)’.

- **CLASSPATH_ENV**  
  This variable is a shell expression that is used to set the CLASSPATH environment variable on the javac command line. (In the future we will probably handle class path setting differently.)

### 10.5 Python

Automake provides support for Python compilation with the PYTHON primary. A typical setup is to call AM_PATH_PYTHON in configure.ac and use a line like the following in Makefile.am:

```
python_PYTHON = tree.py leave.py
```

Any files listed in a _PYTHON variable will be byte-compiled with py-compile at install time. py-compile actually creates both standard (.pyc) and optimized (.pyo) byte-compiled versions of the source files. Note that because byte-compilation occurs at install time, any files listed in noinst_PYTHON will not be compiled. Python source files are included in the distribution by default, prepend nodist_ (as in nodist_python_PYTHON) to omit them.

Automake ships with an Autoconf macro called AM_PATH_PYTHON that will determine some Python-related directory variables (see below). If you have called AM_PATH_PYTHON from configure.ac, then you may use the variables python_PYTHON or pkgpython_PYTHON to list Python source files in your Makefile.am, depending on where you want your files installed (see the definitions of pythondir and pkgpythondir below).

```
AM_PATH_PYTHON ([version], [action-if-found], [action-if-not-found])
```

[Macro]
Search for a Python interpreter on the system. This macro takes three optional arguments. The first argument, if present, is the minimum version of Python required for this package: `AM_PATH_PYTHON` will skip any Python interpreter that is older than version. If an interpreter is found and satisfies version, then `action-if-found` is run. Otherwise, `action-if-not-found` is run.

If `action-if-not-found` is not specified, as in the following example, the default is to abort `configure`.

```
AM_PATH_PYTHON([2.2])
```

This is fine when Python is an absolute requirement for the package. If Python \(\geq\) 2.5 was only optional to the package, `AM_PATH_PYTHON` could be called as follows.

```
AM_PATH_PYTHON([2.5],, [:])
```

If the `PYTHON` variable is set when `AM_PATH_PYTHON` is called, then that will be the only Python interpreter that is tried.

`AM_PATH_PYTHON` creates the following output variables based on the Python installation found during configuration.

- **PYTHON** The name of the Python executable, or `:` if no suitable interpreter could be found.
  
  Assuming `action-if-not-found` is used (otherwise `./configure` will abort if Python is absent), the value of `PYTHON` can be used to setup a conditional in order to disable the relevant part of a build as follows.
  
  ```
  AM_PATH_PYTHON(,, [:])
  AM_CONDITIONAL([HAVE_PYTHON], [test "$PYTHON" != :])
  ```

- **PYTHON_VERSION** The Python version number, in the form `major.minor` (e.g., `2.5`). This is currently the value of `sys.version[:3]`.

- **PYTHON_PREFIX** The string `${prefix}`. This term may be used in future work that needs the contents of Python’s `sys.prefix`, but general consensus is to always use the value from `configure`.

- **PYTHON_EXEC_PREFIX** The string `${exec_prefix}`. This term may be used in future work that needs the contents of Python’s `sys.exec_prefix`, but general consensus is to always use the value from `configure`.

- **PYTHON_PLATFORM** The canonical name used by Python to describe the operating system, as given by `sys.platform`. This value is sometimes needed when building Python extensions.

- **pythondir** The directory name for the `site-packages` subdirectory of the standard Python install tree.

- **pkgpythondir** This is the directory under `pythondir` that is named after the package. That is, it is `${pythondir}/${PACKAGE}`. It is provided as a convenience.
pyexecdir

This is the directory where Python extension modules (shared libraries) should be installed. An extension module written in C could be declared as follows to Automake:

```plaintext
pyexec_LTLIBRARIES = quaternion.la
quaternion_la_SOURCES = quaternion.c support.c support.h
quaternion_la_LDFLAGS = -avoid-version -module
```

pkgpyexecdir

This is a convenience variable that is defined as `$(pyexecdir)/$(PACKAGE)`.

All of these directory variables have values that start with either `${prefix}` or `${exec_prefix}` unexpanded. This works fine in Makefiles, but it makes these variables hard to use in configure. This is mandated by the GNU coding standards, so that the user can run `make prefix=/foo install`. The Autoconf manual has a section with more details on this topic (see Section “Installation Directory Variables” in The Autoconf Manual). See also Section 28.10 [Hard-Coded Install Paths], page 154.

11 Building documentation

Currently Automake provides support for Texinfo and man pages.

11.1 Texinfo

If the current directory contains Texinfo source, you must declare it with the TEXINFOS primary. Generally Texinfo files are converted into info, and thus the info_TEXINFOS variable is most commonly used here. Any Texinfo source file must end in the .texi, .txi, or .texinfo extension. We recommend .texi for new manuals.

Automake generates rules to build .info, .dvi, .ps, .pdf and .html files from your Texinfo sources. Following the GNU Coding Standards, only the .info files are built by `make all` and installed by `make install` (unless you use no-installinfo, see below). Furthermore, .info files are automatically distributed so that Texinfo is not a prerequisite for installing your package.

Other documentation formats can be built on request by `make dvi`, `make ps`, `make pdf` and `make html`, and they can be installed with `make install-dvi`, `make install-ps`, `make install-pdf` and `make install-html` explicitly. `make uninstall` will remove everything: the Texinfo documentation installed by default as well as all the above optional formats.

All of these targets can be extended using `-local` rules (see Section 24.1 [Extending], page 132).

If the .texi file @includes version.texi, then that file will be automatically generated. The file version.texi defines four Texinfo flag you can reference using @value{EDITION}, @value{VERSION}, @value{UPDATED}, and @value{UPDATED-MONTH}.

EDITION
VERSION

Both of these flags hold the version number of your program. They are kept separate for clarity.
For the version.texi support, Automake requires the mdate-sh script; this is supplied with Automake and automatically included when automake is invoked with the --add-missing option.

If you have multiple Texinfo files, and you want to use the version.texi feature, then you have to have a separate version file for each Texinfo file. Automake will treat any include in a Texinfo file that matches vers*.texi just as an automatically generated version file.

Sometimes an info file actually depends on more than one .texi file. For instance, in GNU Hello, hello.texi includes the file fdl.texi. You can tell Automake about these dependencies using the texi_TEXINFOS variable. Here is how GNU Hello does it:

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = fdl.texi
```

By default, Automake requires the file texinfo.tex to appear in the same directory as the Makefile.am file that lists the .texi files. If you used AC_CONFIG_AUX_DIR in configure.ac (see Section "Finding ‘configure’ Input" in The Autoconf Manual), then texinfo.tex is looked for there. In both cases, automake then supplies texinfo.tex if --add-missing is given, and takes care of its distribution. However, if you set the TEXINFO_TEX variable (see below), it overrides the location of the file and turns off its installation into the source as well as its distribution.

The option no-texinfo.tex can be used to eliminate the requirement for the file texinfo.tex. Use of the variable TEXINFO_TEX is preferable, however, because that allows the dvi, ps, and pdf targets to still work.

Automake generates an install-info rule; some people apparently use this. By default, info pages are installed by `make install`, so running make install-info is pointless. This can be prevented via the no-installinfo option. In this case, .info files are not installed by default, and user must request this explicitly using `make install-info`.

By default, make install-info and make install-info will try to run the install-info program (if available) to update (or create) the ${infodir}/dir index. If this is undesired, it can be prevented by exporting the AM_UPDATE_INFO_DIR variable to "no".

The following variables are used by the Texinfo build rules.

**MAKEINFO** The name of the program invoked to build .info files. This variable is defined by Automake. If the makeinfo program is found on the system then it will be used by default; otherwise missing will be used instead.

**MAKEINFOHTML** The command invoked to build .html files. Automake defines this to `$(MAKEINFO) --html`.

**MAKEINFOFLAGS** User flags passed to each invocation of `$(MAKEINFO)` and `$(MAKEINFOHTML)`. This user variable (see Section 3.6 [User Variables], page 23) is not expected to
be defined in any Makefile; it can be used by users to pass extra flags to suit their needs.

**AM_MAKEINFOFLAGS**

Maintainer flags passed to each makeinfo invocation. Unlike MAKEINFOFLAGS, these variables are meant to be defined by maintainers in Makefile.am.

\`\$(AM_MAKEINFOFLAGS)\` is passed to makeinfo when building .info files; and \`\$(AM_MAKEINFOHTMLFLAGS)\` is used when building .html files.

For instance, the following setting can be used to obtain one single .html file per manual, without node separators.

```
AM_MAKEINFOHTMLFLAGS = --no-headers --no-split
```

**TEXI2DVI**
The name of the command that converts a .texi file into a .dvi file. This defaults to `texi2dvi`, a script that ships with the Texinfo package.

**TEXI2PDF**
The name of the command that translates a .texi file into a .pdf file. This defaults to `\$(TEXI2DVI) --pdf --batch`.

**DVIPS**
The name of the command that builds a .ps file out of a .dvi file. This defaults to `dvips`.

**TEXINFO_TEX**
If your package has Texinfo files in many directories, you can use the variable TEXINFO_TEX to tell Automake where to find the canonical texinfo.tex for your package. The value of this variable should be the relative path from the current Makefile.am to texinfo.tex:

```
TEXINFO_TEX = ../doc/texinfo.tex
```

### 11.2 Man Pages

A package can also include man pages (but see the GNU standards on this matter, Section “Man Pages” in The GNU Coding Standards.) Man pages are declared using the MANS primary. Generally the man_MANS variable is used. Man pages are automatically installed in the correct subdirectory of mandir, based on the file extension.

File extensions such as .1c are handled by looking for the valid part of the extension and using that to determine the correct subdirectory of mandir. Valid section names are the digits ‘0’ through ‘9’, and the letters ‘1’ and ‘n’.

Sometimes developers prefer to name a man page something like foo.man in the source, and then rename it to have the correct suffix, for example foo.1, when installing the file. Automake also supports this mode. For a valid section named section, there is a corresponding directory named `mansectiondir`, and a corresponding _MANS variable. Files listed in such a variable are installed in the indicated section. If the file already has a valid suffix, then it is installed as-is; otherwise the file suffix is changed to match the section.

For instance, consider this example:

```
man1_MANS = rename.man thesame.1 alsothesame.1c
```
In this case, `rename.man` will be renamed to `rename.1` when installed, but the other files will keep their names.

By default, man pages are installed by `make install`. However, since the GNU project does not require man pages, many maintainers do not expend effort to keep the man pages up to date. In these cases, the `no-installman` option will prevent the man pages from being installed by default. The user can still explicitly install them via `make install-man`.

For fast installation, with many files it is preferable to use `mansection_MANS` over `man_MANS` as well as files that do not need to be renamed.

Man pages are not currently considered to be source, because it is not uncommon for man pages to be automatically generated. Therefore they are not automatically included in the distribution. However, this can be changed by use of the `dist_` prefix. For instance here is how to distribute and install the two man pages of GNU `cpio` (which includes both Texinfo documentation and man pages):

```
dist_man_MANS = cpio.1 mt.1
```

The `nobase_` prefix is meaningless for man pages and is disallowed.

Executables and manpages may be renamed upon installation (see Section 2.2.9 [Renaming], page 10). For manpages this can be avoided by use of the `notrans_` prefix. For instance, suppose an executable `foo` allowing to access a library function `foo` from the command line. The way to avoid renaming of the `foo.3` manpage is:

```
man_MANS = foo.1
notrans_man_MANS = foo.3
```

`notrans_` must be specified first when used in conjunction with either `dist_` or `nodist_` (see Section 14.2 [Fine-grained Distribution Control], page 98). For instance:

```
notrans_dist_man3_MANS = bar.3
```

# 12 What Gets Installed

Naturally, Automake handles the details of actually installing your program once it has been built. All files named by the various primaries are automatically installed in the appropriate places when the user runs `make install`.

## 12.1 Basics of Installation

A file named in a primary is installed by copying the built file into the appropriate directory. The base name of the file is used when installing.

```
bin_PROGRAMS = hello subdir/goodbye
```

In this example, both `hello` and `goodbye` will be installed in `$(bindir)`.

Sometimes it is useful to avoid the basename step at install time. For instance, you might have a number of header files in subdirectories of the source tree that are laid out precisely how you want to install them. In this situation you can use the `nobase_` prefix to suppress the base name step. For example:

```
nobase_include_HEADERS = stdio.h sys/types.h
```

will install `stdio.h` in `$(includedir)` and `types.h` in `$(includedir)/sys`.
Chapter 12: What Gets Installed

For most file types, Automake will install multiple files at once, while avoiding command line length issues (see Section 3.4 [Length Limitations], page 22). Since some `install` programs will not install the same file twice in one invocation, you may need to ensure that file lists are unique within one variable such as `nobase_include_HEADERS` above.

You should not rely on the order in which files listed in one variable are installed. Likewise, to cater for parallel make, you should not rely on any particular file installation order even among different file types (library dependencies are an exception here).

12.2 The Two Parts of Install

Automake generates separate `install-data` and `install-exec` rules, in case the installer is installing on multiple machines that share directory structure—these targets allow the machine-independent parts to be installed only once. `install-exec` installs platform-dependent files, and `install-data` installs platform-independent files. The `install` target depends on both of these targets. While Automake tries to automatically segregate objects into the correct category, the `Makefile.am` author is, in the end, responsible for making sure this is done correctly.

Variables using the standard directory prefixes `data`, `info`, `man`, `include`, `oldinclude`, `pkgdata`, or `pkginclude` are installed by `install-data`.

Variables using the standard directory prefixes `bin`, `sbin`, `libexec`, `sysconf`, `localstate`, `lib`, or `pkglib` are installed by `install-exec`.

For instance, `data_DATA` files are installed by `install-data`, while `bin_PROGRAMS` files are installed by `install-exec`.

Any variable using a user-defined directory prefix with `exec` in the name (e.g., `myexecbin_PROGRAMS`) is installed by `install-exec`. All other user-defined prefixes are installed by `install-data`.

12.3 Extending Installation

It is possible to extend this mechanism by defining an `install-exec-local` or `install-data-local` rule. If these rules exist, they will be run at `make install` time. These rules can do almost anything; care is required.

Automake also supports two install hooks, `install-exec-hook` and `install-data-hook`. These hooks are run after all other install rules of the appropriate type, exec or data, have completed. So, for instance, it is possible to perform post-installation modifications using an install hook. See Section 24.1 [Extending], page 132, for some examples.

12.4 Staged Installs

Automake generates support for the `DESTDIR` variable in all install rules. `DESTDIR` is used during the `make install` step to relocate install objects into a staging area. Each object and path is prefixed with the value of `DESTDIR` before being copied into the install area. Here is an example of typical `DESTDIR` usage:

```bash
mkdir /tmp/staging &&
make DESTDIR=/tmp/staging install
```

The `mkdir` command avoids a security problem if the attacker creates a symbolic link from `/tmp/staging` to a victim area; then `make` places install objects in a directory
tree built under `/tmp/staging`. If `/gnu/bin/foo` and `/gnu/share/aclocal/foo.m4` are to be installed, the above command would install `/tmp/staging/gnu/bin/foo` and `/tmp/staging/gnu/share/aclocal/foo.m4`.

This feature is commonly used to build install images and packages (see Section 2.2.10 [DESTDIR], page 10).

Support for DESTDIR is implemented by coding it directly into the install rules. If your Makefile.am uses a local install rule (e.g., `install-exec-local`) or an install hook, then you must write that code to respect DESTDIR.

See Section “Makefile Conventions” in The GNU Coding Standards, for another usage example.

### 12.5 Install Rules for the User

Automake also generates rules for targets `uninstall`, `installdirs`, and `install-strip`.

Automake supports `uninstall-local` and `uninstall-hook`. There is no notion of separate uninstalls for “exec” and “data”, as these features would not provide additional functionality.

Note that `uninstall` is not meant as a replacement for a real packaging tool.

### 13 What Gets Cleaned

The GNU Makefile Standards specify a number of different clean rules. See Section “Standard Targets for Users” in The GNU Coding Standards.

Generally the files that can be cleaned are determined automatically by Automake. Of course, Automake also recognizes some variables that can be defined to specify additional files to clean. These variables are `MOSTLYCLEANFILES`, `CLEANFILES`, `DISTCLEANFILES`, and `MAINTAINERCLEANFILES`.

When cleaning involves more than deleting some hard-coded list of files, it is also possible to supplement the cleaning rules with your own commands. Simply define a rule for any of the `mostlyclean-local`, `clean-local`, `distclean-local`, or `maintainer-clean-local` targets (see Section 24.1 [Extending], page 132). A common case is deleting a directory, for instance, a directory created by the test suite:

```
clean-local:
    -rm -rf testSubDir
```

Since `make` allows only one set of rules for a given target, a more extensible way of writing this is to use a separate target listed as a dependency:

```
clean-local: clean-local-check
.PHONY: clean-local-check
clean-local-check:
    -rm -rf testSubDir
```

As the GNU Standards aren’t always explicit as to which files should be removed by which rule, we’ve adopted a heuristic that we believe was first formulated by François Pinard:

- If `make` built it, and it is commonly something that one would want to rebuild (for instance, a `.o` file), then `mostlyclean` should delete it.
• Otherwise, if `make` built it, then `clean` should delete it.
• If `configure` built it, then `distclean` should delete it.
• If the maintainer built it (for instance, a `.info` file), then `maintainer-clean` should delete it. However `maintainer-clean` should not delete anything that needs to exist in order to run `./configure && make`.

We recommend that you follow this same set of heuristics in your `Makefile.am`.

14 What Goes in a Distribution

14.1 Basics of Distribution

The `dist` rule in the generated `Makefile.in` can be used to generate a gzipped `tar` file and other flavors of archive for distribution. The file is named based on the `PACKAGE` and `VERSION` variables defined by `AM_INIT_AUTOMAKE` (see Section 6.4 [Macros], page 44); more precisely the gzipped `tar` file is named `package-version.tar.gz`. You can use the `make` variable `GZIP_ENV` to control how gzip is run. The default setting is `--best`.

For the most part, the files to distribute are automatically found by Automake: all source files are automatically included in a distribution, as are all `Makefile.am` and `Makefile.in` files. Automake also has a built-in list of commonly used files that are automatically included if they are found in the current directory (either physically, or as the target of a `Makefile.am` rule); this list is printed by `automake --help`. Note that some files in this list are actually distributed only if other certain conditions hold (for example, the `config.h.top` and `config.h.bot` files are automatically distributed only if, e.g., `AC_CONFIG_HEADERS([config.h])` is used in `configure.ac`). Also, files that are read by `configure` (i.e. the source files corresponding to the files specified in various Autoconf macros such as `AC_CONFIG_FILES` and siblings) are automatically distributed. Files included in a `Makefile.am` (using `include`) or in `configure.ac` (using `m4_include`), and helper scripts installed with `automake --add-missing` are also distributed.

Still, sometimes there are files that must be distributed, but which are not covered in the automatic rules. These files should be listed in the `EXTRA_DIST` variable. You can mention files from subdirectories in `EXTRA_DIST`.

You can also mention a directory in `EXTRA_DIST`; in this case the entire directory will be recursively copied into the distribution. Please note that this will also copy *everything* in the directory, including, e.g., Subversion’s `.svn` private directories or CVS/RCS version control files. We recommend against using this feature.

If you define `SUBDIRS`, Automake will recursively include the subdirectories in the distribution. If `SUBDIRS` is defined conditionally (see Chapter 20 [Conditionals], page 124), Automake will normally include all directories that could possibly appear in `SUBDIRS` in the distribution. If you need to specify the set of directories conditionally, you can set the variable `DIST_SUBDIRS` to the exact list of subdirectories to include in the distribution (see Section 7.2 [Conditional Subdirectories], page 48).
14.2 Fine-grained Distribution Control

Sometimes you need tighter control over what does not go into the distribution; for instance, you might have source files that are generated and that you do not want to distribute. In this case Automake gives fine-grained control using the dist and nodist prefixes. Any primary or _SOURCES variable can be prefixed with dist_ to add the listed files to the distribution. Similarly, nodist_ can be used to omit the files from the distribution.

As an example, here is how you would cause some data to be distributed while leaving some source code out of the distribution:

- dist_data_DATA = distribute-this
- bin_PROGRAMS = foo
- nodist_foo_SOURCES = do-not-distribute.c

14.3 The dist Hook

Occasionally it is useful to be able to change the distribution before it is packaged up. If the dist-hook rule exists, it is run after the distribution directory is filled, but before the actual distribution archives are created. One way to use this is for removing unnecessary files that get recursively included by specifying a directory in EXTRA_DIST:

- EXTRA_DIST = doc
- dist-hook:
  - rm -rf $(distdir)/doc

Note that the dist-hook recipe shouldn’t assume that the regular files in the distribution directory are writable; this might not be the case if one is packaging from a read-only source tree, or when a make distcheck is being done. For similar reasons, the recipe shouldn’t assume that the subdirectories put into the distribution directory as effect of having them listed in EXTRA_DIST are writable. So, if the dist-hook recipe wants to modify the content of an existing file (or EXTRA_DIST subdirectory) in the distribution directory, it should explicitly to make it writable first:

- EXTRA_DIST = README doc
- dist-hook:
  - chmod u+w $(distdir)/README
  - echo "Distribution date: 'date'" >> README
  - rm -f $(distdir)/doc/HACKING

Two variables that come handy when writing dist-hook rules are ‘$(distdir)’ and ‘$(top_distdir)’.

‘$(distdir)’ points to the directory where the dist rule will copy files from the current directory before creating the tarball. If you are at the top-level directory, then ‘$(distdir) = $(PACKAGE)-$(VERSION)’. When used from subdirectory named foo/, then ‘$(distdir) = ../$(PACKAGE)-$(VERSION)/foo’. ‘$(distdir)’ can be a relative or absolute path, do not assume any form.

‘$(top_distdir)’ always points to the root directory of the distributed tree. At the top-level it’s equal to ‘$(distdir)’. In the foo/ subdirectory ‘$(top_distdir) = ../$(PACKAGE)-$(VERSION)’. $(top_distdir) too can be a relative or absolute path.
Note that when packages are nested using `AC_CONFIG_SUBDIRS` (see Section 7.4 [Sub-packages], page 52), then `$\texttt{distdir}$’ and `$\texttt{top_distdir}$’ are relative to the package where \texttt{make dist} was run, not to any sub-packages involved.

### 14.4 Checking the Distribution

Automake also generates a \texttt{distcheck} rule that can be of help to ensure that a given distribution will actually work. Simplifying a bit, we can say this rule first makes a distribution, and then, \textit{operating from it}, takes the following steps:

- tries to do a \texttt{VPATH} build (see Section 2.2.6 [VPATH Builds], page 6), with the \texttt{srcdir} and all its content made \texttt{read-only};
- runs the test suite (with \texttt{make check}) on this fresh build;
- installs the package in a temporary directory (with \texttt{make install}), and tries runs the test suite on the resulting installation (with \texttt{make installcheck});
- checks that the package can be correctly uninstalled (by \texttt{make uninstall}) and cleaned (by \texttt{make distclean});
- finally, makes another tarball to ensure the distribution is self-contained.

#### \texttt{DISTCHECK\_CONFIGURE\_FLAGS}

Building the package involves running \texttt{./configure}. If you need to supply additional flags to \texttt{configure}, define them in the \texttt{AM\_DISTCHECK\_CONFIGURE\_FLAGS} variable in your top-level Makefile.am. The user can still extend or override the flags provided there by defining the \texttt{DISTCHECK\_CONFIGURE\_FLAGS} variable, on the command line when invoking \texttt{make}.

Still, developers are encouraged to strive to make their code buildable without requiring any special configure option; thus, in general, you shouldn’t define \texttt{AM\_DISTCHECK\_CONFIGURE\_FLAGS}. However, there might be few scenarios in which the use of this variable is justified. GNU \texttt{m4} offers an example: GNU \texttt{m4} configures by default with its experimental and seldom used "changeword" feature disabled; so in its case it is useful to have \texttt{make distcheck} run configure with the \texttt{--with-changeword} option, to ensure that the code for changeword support still compiles correctly. GNU \texttt{m4} also employs the \texttt{AM\_DISTCHECK\_CONFIGURE\_FLAGS} variable to stress-test the use of \texttt{--program-prefix=g}, since at one point the \texttt{m4} build system had a bug where \texttt{make installcheck} was wrongly assuming it could blindly test "\texttt{m4}", rather than the just-installed "\texttt{gm4}".

#### \texttt{distcheck\_hook}

If the \texttt{distcheck\_hook} rule is defined in your top-level Makefile.am, then it will be invoked by \texttt{distcheck} after the new distribution has been unpacked, but before the unpacked copy is configured and built. Your \texttt{distcheck\_hook} can do almost anything, though as always caution is advised. Generally this hook is used to check for potential distribution errors not caught by the standard mechanism. Note that \texttt{distcheck\_hook} as well as \texttt{AM\_DISTCHECK\_CONFIGURE\_FLAGS} and \texttt{DISTCHECK\_CONFIGURE\_FLAGS} are not honored in a subpackage Makefile.am, but the flags from \texttt{AM\_DISTCHECK\_CONFIGURE\_FLAGS} and \texttt{DISTCHECK\_CONFIGURE\_FLAGS} are passed down to the \texttt{configure} script of the subpackage.
**distcleancheck**

Speaking of potential distribution errors, `distcheck` also ensures that the `distclean` rule actually removes all built files. This is done by running `make distcleancheck` at the end of the VPATH build. By default, `distcleancheck` will run `distclean` and then make sure the build tree has been emptied by running `${(distcleancheck_listfiles)}`. Usually this check will find generated files that you forgot to add to the `DISTCLEANFILES` variable (see Chapter 13 [Clean], page 96).

The `distcleancheck` behavior should be OK for most packages, otherwise you have the possibility to override the definition of either the `distcleancheck` rule, or the `${(distcleancheck_listfiles)}` variable. For instance, to disable `distcleancheck` completely, add the following rule to your top-level `Makefile.am`:

```
distcleancheck:
    @:
```

If you want `distcleancheck` to ignore built files that have not been cleaned because they are also part of the distribution, add the following definition instead:

```
distcleancheck_listfiles = \n    find . -type f -exec sh -c 'test -f $(srcdir)/$$1 || echo $$1' \n        sh '{ }';
```

The above definition is not the default because it’s usually an error if your Makefiles cause some distributed files to be rebuilt when the user build the package. (Think about the user missing the tool required to build the file; or if the required tool is built by your package, consider the cross-compilation case where it can’t be run.) There is an entry in the FAQ about this (see Section 28.5 [Errors with distclean], page 143), make sure you read it before playing with `distcleancheck_listfiles`.

**distuninstallcheck**

`distcheck` also checks that the `uninstall` rule works properly, both for ordinary and DESTDIR builds. It does this by invoking `make uninstall`, and then it checks the install tree to see if any files are left over. This check will make sure that you correctly coded your `uninstall`-related rules.

By default, the checking is done by the `distuninstallcheck` rule, and the list of files in the install tree is generated by `${(distuninstallcheck_listfiles)}` (this is a variable whose value is a shell command to run that prints the list of files to stdout).

Either of these can be overridden to modify the behavior of `distcheck`. For instance, to disable this check completely, you would write:

```
distuninstallcheck:
    @:
```

### 14.5 The Types of Distributions

Automake generates rules to provide archives of the project for distributions in various formats. Their targets are:

**dist-bzip2**

Generate a bzip2 tar archive of the distribution. bzip2 archives are frequently smaller than gzipped archives. By default, this rule makes `bzip2` use a com-
pression option of \texttt{-9}. To make it use a different one, set the \texttt{BZIP2} environment variable. For example, ‘\texttt{make dist-bzip2 BZIP2=-7}’.

\texttt{dist-gzip}

Generate a gzip tar archive of the distribution.

\texttt{dist-lzip}

Generate an ‘lzip’ tar archive of the distribution. lzip archives are frequently smaller than bzip2-compressed archives.

\texttt{dist-shar}

Generate a shar archive of the distribution.

\texttt{dist-xz}

Generate an ‘xz’ tar archive of the distribution. xz archives are frequently smaller than bzip2-compressed archives. By default, this rule makes ‘xz’ use a compression option of \texttt{-e}. To make it use a different one, set the \texttt{XZ_OPT} environment variable. For example, run this command to use the default compression ratio, but with a progress indicator: ‘\texttt{make dist-xz XZ_OPT=-7e}’.

\texttt{dist-zip}

Generate a zip archive of the distribution.

\texttt{dist-tarZ}

Generate a compressed tar archive of the distribution.

The rule \texttt{dist} (and its historical synonym \texttt{dist-all}) will create archives in all the enabled formats, Chapter 17 [Options], page 117. By default, only the \texttt{dist-gzip} target is hooked to \texttt{dist}.

15 Support for test suites

Automake can generate code to handle two kinds of test suites. One is based on integration with the \texttt{dejagnu} framework. The other (and most used) form is based on the use of generic test scripts, and its activation is triggered by the definition of the special \texttt{TESTS} variable. This second form allows for various degrees of sophistication and customization; in particular, it allows for concurrent execution of test scripts, use of established test protocols such as TAP, and definition of custom test drivers and test runners.

In either case, the testsuite is invoked via ‘\texttt{make check}’.

15.1 Generalities about Testing

The purpose of testing is to determine whether a program or system behaves as expected (e.g., known inputs produce the expected outputs, error conditions are correctly handled or reported, and older bugs do not resurface).

The minimal unit of testing is usually called \textit{test case}, or simply \textit{test}. How a test case is defined or delimited, and even what exactly \textit{constitutes} a test case, depends heavily on the testing paradigm and/or framework in use, so we won’t attempt any more precise definition. The set of the test cases for a given program or system constitutes its testsuite.

A test harness (also testsuite harness) is a program or software component that executes all (or part of) the defined test cases, analyzes their outcomes, and report or register these outcomes appropriately. Again, the details of how this is accomplished (and how the
developer and user can influence it or interface with it) varies wildly, and we’ll attempt no precise definition.

A test is said to pass when it can determine that the condition or behaviour it means to verify holds, and is said to fail when it can determine that such condition of behaviour does not hold.

Sometimes, tests can rely on non-portable tools or prerequisites, or simply make no sense on a given system (for example, a test checking a Windows-specific feature makes no sense on a GNU/Linux system). In this case, accordingly to the definition above, the tests can neither be considered passed nor failed; instead, they are skipped — i.e., they are not run, or their result is anyway ignored for what concerns the count of failures an successes. Skips are usually explicitly reported though, so that the user will be aware that not all of the testsuite has really run.

It’s not uncommon, especially during early development stages, that some tests fail for known reasons, and that the developer doesn’t want to tackle these failures immediately (this is especially true when the failing tests deal with corner cases). In this situation, the better policy is to declare that each of those failures is an expected failure (or xfail). In case a test that is expected to fail ends up passing instead, many testing environments will flag the result as a special kind of failure called unexpected pass (or xpass).

Many testing environments and frameworks distinguish between test failures and hard errors. As we’ve seen, a test failure happens when some invariant or expected behaviour of the software under test is not met. An hard error happens when e.g., the set-up of a test case scenario fails, or when some other unexpected or highly undesirable condition is encountered (for example, the program under test experiences a segmentation fault).

TODO: Links to other test harnesses (esp. those sharing our terminology)?

15.2 Simple Tests

15.2.1 Scripts-based Testsuites

If the special variable TESTS is defined, its value is taken to be a list of programs or scripts to run in order to do the testing. Under the appropriate circumstances, it’s possible for TESTS to list also data files to be passed to one or more test scripts defined by different means (the so-called “log compilers”, see Section 15.2.3 [Parallel Test Harness], page 105).

Test scripts can be executed serially or concurrently. Automake supports both these kinds of test execution, with the serial test harness being the default (for backward-compatibility reasons only, as its use is nowadays discouraged). The concurrent test harness relies on the concurrence capabilities (if any) offered by the underlying make implementation, and can thus only be as good as those are.

By default, only the exit statuses of the test scripts are considered when determining the testsuite outcome. But Automake allows also the use of more complex test protocols, either standard (see Section 15.4 [Using the TAP test protocol], page 112) or custom (see Section 15.3 [Custom Test Drivers], page 108). Note that you can enable such protocols only when the parallel harness is used: they won’t work with the serial test harness. In the rest of this section we are going to concentrate mostly on protocol-less tests, since we cover test protocols in a later section (again, see Section 15.3 [Custom Test Drivers], page 108).
When no test protocol is in use, an exit status of 0 from a test script will denote a success, an exit status of 77 a skipped test, an exit status of 99 an hard error, and any other exit status will denote a failure.

You may define the variable `XFAIL_TESTS` to a list of tests (usually a subset of `TESTS`) that are expected to fail; this will effectively reverse the result of those tests (with the provision that skips and hard errors remain untouched). You may also instruct the test suite harness to treat hard errors like simple failures, by defining the `DISABLE_HARD_ERRORS` make variable to a nonempty value.

Note however that, for tests based on more complex test protocols, the exact effects of `XFAIL_TESTS` and `DISABLE_HARD_ERRORS` might change, or they might even have no effect at all (for example, in tests using TAP, there is not way to disable hard errors, and the `DISABLE_HARD_ERRORS` variable has no effect on them).

The result of each test case run by the scripts in `TESTS` will be printed on standard output, along with the test name. For test protocols that allow more test cases per test script (such as TAP), a number, identifier and/or brief description specific for the single test case is expected to be printed in addition to the name of the test script. The possible results (whose meanings should be clear from the previous Section 15.1 [Generalities about Testing], page 101) are `PASS`, `FAIL`, `SKIP`, `XFAIL`, `XPASS` and `ERROR`. Here is an example of output from an hypothetical testsuite that uses both plain and TAP tests:

```
PASS: foo.sh
PASS: zardoz.tap 1 - Daemon started
PASS: zardoz.tap 2 - Daemon responding
SKIP: zardoz.tap 3 - Daemon uses /proc # SKIP /proc is not mounted
PASS: zardoz.tap 4 - Daemon stopped
SKIP: bar.sh
PASS: mu.tap 1
XFAIL: mu.tap 2 # TODO frobnication not yet implemented
```

A testsuite summary (expected to report at least the number of run, skipped and failed tests) will be printed at the end of the testsuite run.

If the Automake option `color-tests` is used (see Chapter 17 [Options], page 117) and standard output is connected to a capable terminal, then the test results and the summary are colored appropriately. The user can disable colored output by setting the `make` variable `AM_COLOR_TESTS=no`, or force colored output even without a connecting terminal with `AM_COLOR_TESTS=always`. It's also worth noting that some `make` implementations, when used in parallel mode, have slightly different semantics (see Section “Parallel make” in `The Autoconf Manual`), which can break the automatic detection of a connection to a capable terminal. If this is the case, you'll have to resort to the use of `AM_COLOR_TESTS=always` in order to have the testsuite output colorized.

Test programs that need data files should look for them in `srcdir` (which is both a make variable and an environment variable made available to the tests), so that they work when building in a separate directory (see Section “Build Directories” in `The Autoconf Manual`), and in particular for the `distcheck` rule (see Section 14.4 [Checking the Distribution], page 99).

The `AM_TESTS_ENVIRONMENT` and `TESTS_ENVIRONMENT` variables can be used to run initialization code and set environment variables for the test scripts. The former variable is
developer-reserved, and can be defined in the `Makefile.am`, while the latter is reserved for the user, which can employ it to extend or override the settings in the former; for this to work portably, however, the contents of a non-empty `AM_TESTS_ENVIRONMENT` must be terminated by a semicolon.

The `AM_TESTS_FD_REDIRECT` variable can be used to define file descriptor redirections for the test scripts. One might think that `AM_TESTS_ENVIRONMENT` could be used for this purpose, but experience has shown that doing so portably is practically impossible. The main hurdle is constituted by Korn shells, which usually set the close-on-exec flag on file descriptors opened with the `exec` builtin, thus rendering an idiom like `AM_TESTS_ENVIRONMENT = exec 9>&2;` ineffectual. This issue also affects some Bourne shells, such as the HP-UX’s `/bin/sh`.

```bash
AM_TESTS_ENVIRONMENT = \n   ## Some environment initializations are kept in a separate shell
   ## file ‘tests-env.sh’, which can make it easier to also run tests
   ## from the command line.
   . $(srcdir)/tests-env.sh; \n   ## On Solaris, prefer more POSIX-compliant versions of the standard
   ## tools by default.
   if test -d /usr/xpg4/bin; then \n      PATH=/usr/xpg4/bin:$$PATH; export PATH; \n   fi;
   ## With this, the test scripts will be able to print diagnostic
   ## messages to the original standard error stream, even if the test
   ## driver redirects the stderr of the test scripts to a log file
   ## before executing them.
   AM_TESTS_FD_REDIRECT = 9>&2
```

Note however that `AM_TESTS_ENVIRONMENT` is, for historical and implementation reasons, not supported by the serial harness (see Section 15.2.2 [Serial Test Harness], page 104).

Automake ensures that each file listed in `TESTS` is built before it is run; you can list both source and derived programs (or scripts) in `TESTS`; the generated rule will look both in `srcdir` and .. For instance, you might want to run a C program as a test. To do this you would list its name in `TESTS` and also in `check_PROGRAMS`, and then specify it as you would any other program.

Programs listed in `check_PROGRAMS` (and `check_LIBRARIES`, `check_LTLIBRARIES`...) are only built during `make check`, not during `make all`. You should list there any program needed by your tests that does not need to be built by `make all`. Note that `check_PROGRAMS` are not automatically added to `TESTS` because `check_PROGRAMS` usually lists programs used by the tests, not the tests themselves. Of course you can set `TESTS = $(check_PROGRAMS)` if all your programs are test cases.

### 15.2.2 Serial Test Harness

**NOTE:** This harness, while still being the default one, is obsolescent, and kept mostly for backward-compatibility reasons. The user is advised to use the parallel test harness instead (see Section 15.2.3 [Parallel Test Harness], page 105). Be warned that future Automake versions might switch to use that more modern and feature-rich harness by default.
The serial test harness is enabled by the Automake option `serial-tests`. It operates by simply running the tests serially, one at the time, without any I/O redirection. It’s up to the user to implement logging of tests’ output, if that’s required or desired.

For historical and implementation reasons, the `AM_TESTS_ENVIRONMENT` variable is not supported by this harness (it will be silently ignored if defined); only `TESTS_ENVIRONMENT` is, and it is to be considered a developer-reserved variable. This is done so that, when using the serial harness, `TESTS_ENVIRONMENT` can be defined to an invocation of an interpreter through which the tests are to be run. For instance, the following setup may be used to run tests with Perl:

```
TESTS_ENVIRONMENT = $(PERL) -Mstrict -w
TESTS = foo.pl bar.pl baz.pl
```

It’s important to note that the use of `TESTS_ENVIRONMENT` endorsed here would be invalid with the parallel harness. That harness provides a more elegant way to achieve the same effect, with the further benefit of freeing the `TESTS_ENVIRONMENT` variable for the user (see Section 15.2.3 [Parallel Test Harness], page 105).

Another, less serious limit of the serial harness is that it doesn’t really distinguish between simple failures and hard errors; this is due to historical reasons only, and might be fixed in future Automake versions.

### 15.2.3 Parallel Test Harness

The parallel (or concurrent) test harness is enabled by the Automake option `parallel-tests`. It features automatic collection of the test scripts output in `.log` files, concurrent execution of tests with `make -j`, specification of inter-test dependencies, lazy reruns of tests that have not completed in a prior run, and hard errors for exceptional failures.

This harness is still somewhat experimental and may undergo changes in order to satisfy additional portability requirements.

The parallel test harness operates by defining a set of `make` rules that run the test scripts listed in `TESTS`, and, for each such script, save its output in a corresponding `.log` file and its results (and other “metadata”, see Section 15.3.3 [API for Custom Test Drivers], page 109) in a corresponding `.trs` (as in `Test ReSults`) file. The `.log` file will contain all the output emitted by the test on its standard output and its standard error. The `.trs` file will contain, among the other things, the results of the test cases run by the script.

The parallel test harness will also create a summary log file, `TEST_SUITE_LOG`, which defaults to `test-suite.log` and requires a `.log` suffix. This file depends upon all the `.log` and `.trs` files created for the test scripts listed in `TESTS`.

As with the serial harness above, by default one status line is printed per completed test, and a short summary after the suite has completed. However, standard output and standard error of the test are redirected to a per-test log file, so that parallel execution does not produce intermingled output. The output from failed tests is collected in the `test-suite.log` file. If the variable ‘`VERBOSE`’ is set, this file is output after the summary. For best results, the tests should be verbose by default now.

Each couple of `.log` and `.trs` files is created when the corresponding test has completed. The set of log files is listed in the read-only variable `TEST_LOGS`, and defaults to `TESTS`, with the executable extension if any (see Section 8.20 [EXEEXT], page 81), as well as any suffix
listed in `TEST_EXTENSIONS` removed, and `.log` appended. Results are undefined if a test file name ends in several concatenated suffixes. `TEST_EXTENSIONS` defaults to `.test`; it can be overridden by the user, in which case any extension listed in it must be constituted by a dot, followed by a non-digit alphabetic character, followed by any number of alphabetic characters. For example, ‘.sh’, ‘.T’ and ‘.t1’ are valid extensions, while ‘.x-y’, ‘.6c’ and ‘.t.1’ are not.

For tests that match an extension `.ext` listed in `TEST_EXTENSIONS`, you can provide a custom “test runner” using the variable `ext_LOG_COMPILER` (note the upper-case extension) and pass options in `AM_ext_LOG_FLAGS` and allow the user to pass options in `ext_LOG_FLAGS`. It will cause all tests with this extension to be called with this runner. For all tests without a registered extension, the variables `LOG_COMPILER`, `AM_LOG_FLAGS`, and `LOG_FLAGS` may be used. For example,

```plaintext
TESTS = foo.pl bar.py baz
TEST_EXTENSIONS = .pl .py
PL_LOG_COMPILER = $(PERL)
AM_PL_LOG_FLAGS = -w
PY_LOG_COMPILER = $(PYTHON)
AM_PY_LOG_FLAGS = -v
LOG_COMPILER = ./wrapper-script
AM_LOG_FLAGS = -d
```

will invoke `$(PERL) -w foo.pl`, `$(PYTHON) -v bar.py`, and `./wrapper-script -d baz` to produce `foo.log`, `bar.log`, and `baz.log`, respectively. The `foo.trs`, `bar.trs` and `baz.trs` files will be automatically produced as a side-effect.

It’s important to note that, differently from what we’ve seen for the serial test harness (see Section 15.2.3 [Parallel Test Harness], page 105), the `AM_TESTS_ENVIRONMENT` and `TESTS_ENVIRONMENT` variables cannot be use to define a custom test runner; the `LOG_COMPILER` and `LOG_FLAGS` (or their extension-specific counterparts) should be used instead:

```plaintext
## This is WRONG!
AM_TESTS_ENVIRONMENT = PERL5LIB='$(srcdir)/lib' $(PERL) -Mstrict -w
## Do this instead.
AM_TESTS_ENVIRONMENT = PERL5LIB='$(srcdir)/lib'; export PERL5LIB;
LOG_COMPILER = $(PERL)
AM_LOG_FLAGS = -Mstrict -w
```

By default, the test suite harness will run all tests, but there are several ways to limit the set of tests that are run:

- You can set the `TESTS` variable. For example, you can use a command like this to run only a subset of the tests:

  ```plaintext
  env TESTS="foo.test bar.test" make -e check
  ```

  Note however that the command above will unconditionally overwrite the `test-suite.log` file, thus clobbering the recorded results of any previous testsuite run. This might be undesirable for packages whose testsuite takes long time to execute. Luckily, this problem can easily be avoided by overriding also `TEST_SUITE_LOG` at runtime; for example,

  ```plaintext
  env TEST_SUITE_LOG=partial.log TESTS="..." make -e check
  ```
Chapter 15: Support for test suites

will write the result of the partial testsuite runs to the `partial.log`, without touching `test-suite.log`.

- You can set the `TEST_LOGS` variable. By default, this variable is computed at `make` run time from the value of `TESTS` as described above. For example, you can use the following:

  ```
  set x subset*.log; shift
  env TEST_LOGS="foo.log $*" make -e check
  ```

  The comments made above about `TEST_SUITE_LOG` overriding applies here too.

- By default, the test harness removes all old per-test `.log` and `.trs` files before it starts running tests to regenerate them. The variable `RECHECK_LOGS` contains the set of `.log` (and, by implication, `.trs`) files which are removed. `RECHECK_LOGS` defaults to `TEST_LOGS`, which means all tests need to be rechecked. By overriding this variable, you can choose which tests need to be reconsidered. For example, you can lazily rerun only those tests which are outdated, i.e., older than their prerequisite test files, by setting this variable to the empty value:

  ```
  env RECHECK_LOGS= make -e check
  ```

- You can ensure that all tests are rerun which have failed or passed unexpectedly, by running `make recheck` in the test directory. This convenience target will set `RECHECK_LOGS` appropriately before invoking the main test harness.

In order to guarantee an ordering between tests even with `make -j N`, dependencies between the corresponding `.log` files may be specified through usual `make` dependencies. For example, the following snippet lets the test named `foo-execute.test` depend upon completion of the test `foo-compile.test`:

```
TESTS = foo-compile.test foo-execute.test
foo-execute.log: foo-compile.log
```

Please note that this ordering ignores the results of required tests, thus the test `foo-execute.test` is run even if the test `foo-compile.test` failed or was skipped beforehand. Further, please note that specifying such dependencies currently works only for tests that end in one of the suffixes listed in `TEST_EXTENSIONS`.

Tests without such specified dependencies may be run concurrently with parallel `make -j N`, so be sure they are prepared for concurrent execution.

The combination of lazy test execution and correct dependencies between tests and their sources may be exploited for efficient unit testing during development. To further speed up the edit-compile-test cycle, it may even be useful to specify compiled programs in `EXTRA_PROGRAMS` instead of with `check_PROGRAMS`, as the former allows intertwined compilation and test execution (but note that `EXTRA_PROGRAMS` are not cleaned automatically, see Section 3.3 [Uniform], page 20).

The variables `TESTS` and `XFAIL_TESTS` may contain conditional parts as well as configure substitutions. In the latter case, however, certain restrictions apply: substituted test names must end with a nonempty test suffix like `.test`, so that one of the inference rules generated by `automake` can apply. For literal test names, `automake` can generate per-target rules to avoid this limitation.

Please note that it is currently not possible to use `$srcdir/` or `$top_srcdir/` in the `TESTS` variable. This technical limitation is necessary to avoid generating test logs in the
source tree and has the unfortunate consequence that it is not possible to specify distributed
tests that are themselves generated by means of explicit rules, in a way that is portable to
all make implementations (see Section “Make Target Lookup” in The Autoconf Manual, the
semantics of FreeBSD and OpenBSD make conflict with this). In case of doubt you may
want to require to use GNU make, or work around the issue with inference rules to generate
the tests.

15.3 Custom Test Drivers

15.3.1 Overview of Custom Test Drivers Support

Starting from Automake version 1.12, the parallel test harness allows the package authors
to use third-party custom test drivers, in case the default ones are inadequate for their
purposes, or do not support their testing protocol of choice.

A custom test driver is expected to properly run the test programs passed to it (including
the command-line arguments passed to those programs, if any), to analyze their execution
and outcome, to create the .log and .trs files associated to these test runs, and to display
the test results on the console. It is responsibility of the author of the test driver to ensure
that it implements all the above steps meaningfully and correctly; Automake isn’t and can’t
be of any help here. On the other hand, the Automake-provided code for testsuite summary
generation offers support for test drivers allowing several test results per test script, if they
take care to register such results properly (see Section 15.3.3.2 [Log files generation and test
results recording], page 110).

The exact details of how test scripts’ results are to be determined and analyzed is left
to the individual drivers. Some drivers might only consider the test script exit status (this
is done for example by the default test driver used by the parallel test harness, described
in the previous section). Other drivers might implement more complex and advanced test
protocols, which might require them to parse and interpreter the output emitted by the test
script they’re running (examples of such protocols are TAP and SubUnit).

It’s very important to note that, even when using custom test drivers, most of the
infrastructure described in the previous section about the parallel harness remains in place;
this includes:
- list of test scripts defined in TESTS, and overridable at runtime through the redefinition
  of TESTS or TEST_LOGS;
- concurrency through the use of make’s option -j;
- per-test .log and .trs files, and generation of a summary .log file from them;
- recheck target, RECHECK_LOGS variable, and lazy reruns of tests;
- inter-test dependencies;
- support for check_* variables (check_PROGRAMS, check_LIBRARIES, ...);
- use of VERBOSE environment variable to get verbose output on testsuite failures;
- definition and honoring of TESTS_ENVIRONMENT, AM_TESTS_ENVIRONMENT and
  AM_TESTS_FD_REDIRECT variables;
- definition of generic and extension-specific LOG_COMPILER and LOG_FLAGS variables.

On the other hand, the exact semantics of how (and if) color-tests, XFAIL_TESTS, and
hard errors are supported and handled is left to the individual test drivers.
15.3.2 Declaring Custom Test Drivers

Custom testsuite drivers are declared by defining the make variables LOG_DRIVER or ext_LOG_DRIVER (where ext must be declared in TEST_EXTENSIONS). They must be defined to programs or scripts that will be used to drive the execution, logging, and outcome report of the tests with corresponding extensions (or of those with no registered extension in the case of LOG_DRIVER). Clearly, multiple distinct test drivers can be declared in the same Makefile.am. Note moreover that the LOG_DRIVER variables are not a substitute for the LOG_COMPILER variables: the two sets of variables can, and often do, usefully and legitimately coexist.

The developer-reserved variable AM_LOG_DRIVER_FLAGS and the user-reserved variable LOG_DRIVER_FLAGS can be used to define flags that will be passed to each invocation of LOG_DRIVER, with the user-defined flags obviously taking precedence over the developer-reserved ones. Similarly, for each extension ext declared in TEST_EXTENSIONS, flags listed in AM_ext_LOG_DRIVER_FLAGS and ext_LOG_DRIVER_FLAGS will be passed to invocations of ext_LOG_DRIVER.

15.3.3 API for Custom Test Drivers

Note that the APIs described here are still highly experimental, and will very likely undergo tightenings and likely also extensive changes in the future, to accommodate for new features or to satisfy additional portability requirements.

The main characteristic of these APIs is that they are designed to share as much infrastructure, semantics, and implementation details as possible with the parallel test harness and its default driver.

15.3.3.1 Command-line arguments for test drivers

A custom driver can rely on various command-line options and arguments being passed to it automatically by the Automake’s parallel-tests harness. It is mandatory that it understands all of them (even if the exact interpretation of the associated semantics can legitimately change between a test driver and another, and even be a no-op in some drivers).

Here is the list of options:

--test-name=NAME
The name of the test, with VPATH prefix (if any) removed. This can have a suffix and a directory component (as in e.g., sub/foo.test), and is mostly meant to be used in console reports about testsuite advancements and results (see Section 15.3.3.3 [Testsuite progress output], page 111).

--log-file=PATH.log
The .log file the test driver must create (see [Basics of test metadata], page 105). If it has a directory component (as in e.g., sub/foo.log), the test harness will ensure that such directory exists before the test driver is called.

--trs-file=PATH.trs
The .trs file the test driver must create (see [Basics of test metadata], page 105). If it has a directory component (as in e.g., sub/foo.trs), the test harness will ensure that such directory exists before the test driver is called.
Chapter 15: Support for test suites

--color-tests={yes|no}
Whether the console output should be colorized or not (see [Simple tests and color-tests], page 103, to learn when this option gets activated and when it doesn’t).

--expect-failure={yes|no}
Whether the tested program is expected to fail.

--enable-hard-errors={yes|no}
Whether “hard errors” in the tested program should be treated differently from normal failures or not (the default should be yes). The exact meaning of “hard error” is highly dependent from the test protocols or conventions in use.

--
Explicitly terminate the list of options.

The first non-option argument passed to the test driver is the program to be run, and all the following ones are command-line options and arguments for this program.

Note that the exact semantics attached to the --color-tests, --expect-failure and --enable-hard-errors options are left up to the individual test drivers. Still, having a behaviour compatible or at least similar to that provided by the default parallel-tests driver is advised, as that would offer a better consistency and a more pleasant user experience.

15.3.3.2 Log files generation and test results recording

The test driver must correctly generate the files specified by the --log-file and --trs-file option (even when the tested program fails or crashes).

The .log file should ideally contain all the output produced by the tested program, plus optionally other information that might facilitate debugging or analysis of bug reports. Apart from that, its format is basically free.

The .trs file is used to register some metadata through the use of custom reStructured-Text fields. This metadata is expected to be employed in various ways by the parallel test harness; for example, to count the test results when printing the testsuite summary, or to decide which tests to re-run upon make recheck. Unrecognized metadata in a .trs file is currently ignored by the harness, but this might change in the future. The list of currently recognized metadata follows.

:test-result:
The test driver must use this field to register the results of each test case run by a test script file. Several :test-result: fields can be present in the same .trs file; this is done in order to support test protocols that allow a single test script to run more test cases.

The only recognized test results are currently PASS, XFAIL, SKIP, FAIL, XPASS and ERROR. These results, when declared with :test-result:, can be optionally followed by text holding the name and/or a brief description of the corresponding test; the parallel-tests harness will ignore such extra text when generating test-suite.log and preparing the testsuite summary.
Chapter 15: Support for test suites

:recheck:
If this field is present and defined to no, then the corresponding test script will not be run upon a make recheck. What happens when two or more :recheck: fields are present in the same .trs file is undefined behaviour.

:copy-in-global-log:
If this field is present and defined to no, then the content of the .log file will not be copied into the global test-suite.log. We allow to forsake such copying because, while it can be useful in debugging and analysis of bug report, it can also be just a waste of space in normal situations, e.g., when a test script is successful. What happens when two or more :copy-in-global-log: fields are present in the same .trs file is undefined behaviour.

:test-global-result:
This is used to declare the "global result" of the script. Currently, the value of this field is needed only to be reported (more or less verbatim) in the generated global log file $(TEST_SUITE_LOG), so it’s quite free-form. For example, a test script which run 10 test cases, 6 of which pass and 4 of which are skipped, could reasonably have a PASS/SKIP value for this field, while a test script which run 19 successful tests and one failed test could have an ALMOST PASSED value. What happens when two or more :test-global-result: fields are present in the same .trs file is undefined behaviour.

Let’s see a small example. Assume a .trs file contains the following lines:

```
:test-result: PASS server starts
:global-log-copy: no
:test-result: PASS HTTP/1.1 request
:test-result: FAIL HTTP/1.0 request
:recheck: yes
:test-result: SKIP HTTPS request (TLS library wasn’t available)
:test-result: PASS server stops
```

Then the corresponding test script will be re-run by make check, will contribute with five test results to the testsuite summary (three of these tests being successful, one failed, and one skipped), and the content of the corresponding .log file will not be copied in the global log file test-suite.log.

15.3.3.3 Testsuite progress output

A custom test driver also has the task of displaying, on the standard output, the test results as soon as they become available. Depending on the protocol in use, it can also display the reasons for failures and skips, and, more generally, any useful diagnostic output (but remember that each line on the screen is precious, so that cluttering the screen with overly verbose information is bad idea). The exact format of this progress output is left up to the test driver; in fact, a custom test driver might theoretically even decide not to do any such report, leaving it all to the testsuite summary (that would be a very lousy idea, of course, and serves only to illustrate the flexibility that is granted here).

Remember that consistency is good; so, if possible, try to be consistent with the output of the built-in Automake test drivers, providing a similar “look & feel”. In particular, the testsuite progress output should be colorized when the --color-tests is passed to the
driver. On the other end, if you are using a known and widespread test protocol with well-established implementations, being consistent with those implementations’ output might be a good idea too.

15.4 Using the TAP test protocol

15.4.1 Introduction to TAP

TAP, the Test Anything Protocol, is a simple text-based interface between testing modules or programs and a test harness. The tests (also called “TAP producers” in this context) write test results in a simple format on standard output; a test harness (also called “TAP consumer”) will parse and interpret these results, and properly present them to the user, and/or register them for later analysis. The exact details of how this is accomplished can vary among different test harnesses. The Automake parallel harness will present the results on the console in the usual fashion (see [Testsuite progress on console], page 103), and will use the .trs files (see [Basics of test metadata], page 105) to store the test results and related metadata. Apart from that, it will try to remain as much compatible as possible with pre-existing and widespread utilities, such as the prove utility (http://search.cpan.org/~andya/Test-Harness/bin/prove), at least for the simpler usages.

TAP started its life as part of the test harness for Perl, but today it has been (mostly) standardized, and has various independent implementations in different languages; among them, C, C++, Perl, Python, PHP, and Java. For a semi-official specification of the TAP protocol, please refer to the documentation of ‘Test::Harness::TAP’ (http://search.cpan.org/~petdance/Test-Harness/lib/Test/Harness/TAP.pod).

The most relevant real-world usages of TAP are obviously in the testsuites of perl and of many perl modules. Still, also other important third-party packages, such as git (http://git-scm.com/), use TAP in their testsuite.

15.4.2 Use TAP with the Automake test harness

Currently, the TAP driver that comes with Automake requires some by-hand steps on the developer’s part (this situation should hopefully be improved in future Automake versions). You’ll have to grab the tap-driver.sh script from the Automake distribution by hand, copy it in your source tree, add a call to AC_PROG_AWK in configure.ac to search for a proper awk program, and use the Automake support for third-party test drivers to instruct the harness to use the tap-driver.sh script and that awk program to run your TAP-producing tests. See the example below for clarification.

Apart from the options common to all the Automake test drivers (see Section 15.3.3.1 [Command-line arguments for test drivers], page 109), the tap-driver.sh supports the following options, whose names are chosen for enhanced compatibility with the prove utility.

`--ignore-exit`

Causes the test driver to ignore the exit status of the test scripts; by default, the driver will report an error if the script exits with a non-zero status. This option has effect also on non-zero exit statuses due to termination by a signal.
--comments
Instruct the test driver to display TAP diagnostic (i.e., lines beginning with the
`#` character) in the testsuite progress output too; by default, TAP diagnostic
is only copied to the `.log` file.

--no-comments
Revert the effects of --comments.

--merge
Instruct the test driver to merge the test scripts’ standard error into their
standard output. This is necessary if you want to ensure that diagnostics from
the test scripts are displayed in the correct order relative to test results; this
can be of great help in debugging (especially if your test scripts are shell scripts
run with shell tracing active). As a downside, this option might cause the test
harness to get confused if anything that appears on standard error looks like a
test result.

--no-merge
Revert the effects of --merge.

--diagnostic-string=STRING
Change the string that introduces TAP diagnostic from the default value of “#”
to STRING. This can be useful if your TAP-based test scripts produce verbose
output on which they have limited control (because, say, the output comes from
other tools invoked in the scripts), and it might contain text that gets spuriously
interpreted as TAP diagnostic: such an issue can be solved by redefining the
string that activates TAP diagnostic to a value you know won’t appear by
chance in the tests’ output. Note however that this feature is non-standard, as
the “official” TAP protocol does not allow for such a customization; so don’t
use it if you can avoid it.

Here is an example of how the TAP driver can be set up and used.

```
% cat configure.ac
AC_INIT([GNU Try Tap], [1.0], [bug-automake@gnu.org])
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign parallel-tests -Wall -Werror])
AC_CONFIG_FILES([Makefile])
AC_REQUIRE_AUX_FILE([tap-driver.sh])
AC_PROG_AWK
AC_OUTPUT

% cat Makefile.am
TEST_LOG_DRIVER = env AM_TAP_AWK='$(AWK)' $(SHELL) \
                   $(top_srcdir)/build-aux/tap-driver.sh
TESTS = foo.test bar.test baz.test
EXTRA_DIST = $(TESTS)

% cat foo.test
#!/bin/sh
echo 1..4 # Number of tests to be executed.
```
echo 'ok 1 - Swallows fly'
echo 'not ok 2 - Caterpillars fly # TODO metamorphosis in progress'
echo 'ok 3 - Pigs fly # SKIP not enough acid'
echo '# I just love word plays ...'
echo 'ok 4 - Flies fly too :-)'%

% cat bar.test
#!/bin/sh
echo 1..3
echo 'not ok 1 - Bummer, this test has failed.'
echo 'ok 2 - This passed though.'
echo 'Bail out! Ennui kicking in, sorry...'
echo 'ok 3 - This will not be seen.'%

% cat baz.test
#!/bin/sh
echo 1..1
echo ok 1
# Exit with error, even if all the tests have been successful.
exit 7%

% cp PREFIX/share/automake-APIVERSION/tap-driver.pl .
% autoreconf --vi & & ./configure & & make check
... PASS: foo.test 1 - Swallows fly
XFAIL: foo.test 2 - Caterpillars fly # TODO metamorphosis in progress
SKIP: foo.test 3 - Pigs fly # SKIP not enough acid
PASS: foo.test 4 - Flies fly too :-)
FAIL: bar.test 1 - Bummer, this test has failed.
PASS: bar.test 2 - This passed though.
ERROR: bar.test - Bail out! Ennui kicking in, sorry...
PASS: baz.test 1
ERROR: baz.test - exited with status 7
... Please report to bug-automake@gnu.org
...
% echo exit status: $?
exit status: 1
%
% env TEST_LOG_DRIVER_FLAGS='--comments --ignore-exit' \
    TESTS='foo.test baz.test' make -e check
... PASS: foo.test 1 - Swallows fly
XFAIL: foo.test 2 - Caterpillars fly # TODO metamorphosis in progress
SKIP: foo.test 3 - Pigs fly # SKIP not enough acid
# foo.test: I just love word plays...
PASS: foo.test 4 - Flies fly too :-(
PASS: baz.test 1
...
% echo exit status: $?
exit status: 0

15.4.3 Incompatibilities with other TAP parsers and drivers
For implementation or historical reasons, the TAP driver and harness as implemented by Automake have some minors incompatibilities with the mainstream versions, which you should be aware of.

- A `Bail out!` directive doesn’t stop the whole testsuite, but only the test script it occurs in. This doesn’t follow TAP specifications, but on the other hand it maximizes compatibility (and code sharing) with the “hard error” concept of the default `parallel-tests` driver.
- The `version` and `pragma` directives are not supported.
- The `--diagnostic-string` option of our driver allows to modify the string that introduces TAP diagnostic from the default value of “#”. The standard TAP protocol has currently no way to allow this, so if you use it your diagnostic will be lost to more compliant tools like `prove` and `Test::Harness`
- And there are probably some other small and yet undiscovered incompatibilities, especially in corner cases or with rare usages.

15.4.4 Links and external resources on TAP
Here are some links to more extensive official or third-party documentation and resources about the TAP protocol and related tools and libraries.

- ‘Test::Harness::TAP’ (http://search.cpan.org/~petdance/Test-Harness/lib/Test/Harness/TAP.pod), the (mostly) official documentation about the TAP format and protocol.
- `prove` (http://search.cpan.org/~andya/Test-Harness/bin/prove), the most famous command-line TAP test driver, included in the distribution of perl and ‘Test::Harness’ (http://search.cpan.org/~andya/Test-Harness/lib/Test/Harness.pm).
- ‘Test::Simple’ (http://search.cpan.org/~mschwern/Test-Simple/lib/Test/Simple.pm) and ‘Test::More’ (http://search.cpan.org/~mschwern/Test-Simple/lib/Test/More.pm), the standard perl testing libraries, which are based on TAP.
- C TAP Harness (http://www.eyrie.org/~eagle/software/c-tap-harness/), a C-based project implementing both a TAP producer and a TAP consumer.
- tap4j (http://www.tap4j.org/), a Java-based project implementing both a TAP producer and a TAP consumer.
15.5 DejaGnu Tests

If dejagnu (ftp://ftp.gnu.org/gnu/dejagnu/) appears in AUTOMAKE_OPTIONS, then a dejagnu-based test suite is assumed. The variable DEJATOOL is a list of names that are passed, one at a time, as the --tool argument to runtest invocations; it defaults to the name of the package.

The variable RUNTESTDEFAULTFLAGS holds the --tool and --srcdir flags that are passed to dejagnu by default; this can be overridden if necessary.

The variables EXPECT and RUNTEST can also be overridden to provide project-specific values. For instance, you will need to do this if you are testing a compiler toolchain, because the default values do not take into account host and target names.

The contents of the variable RUNTESTFLAGS are passed to the runtest invocation. This is considered a “user variable” (see Section 3.6 [User Variables], page 23). If you need to set runtest flags in Makefile.am, you can use AM_RUNTESTFLAGS instead.

Automake will generate rules to create a local site.exp file, defining various variables detected by configure. This file is automatically read by DejaGnu. It is OK for the user of a package to edit this file in order to tune the test suite. However this is not the place where the test suite author should define new variables: this should be done elsewhere in the real test suite code. Especially, site.exp should not be distributed.

Still, if the package author has legitimate reasons to extend site.exp at make time, he can do so by defining the variable EXTRA_DEJAGNU_SITE_CONFIG; the files listed there will be considered site.exp prerequisites, and their content will be appended to it (in the same order in which they appear in EXTRA_DEJAGNU_SITE_CONFIG). Note that files are not distributed by default.

For more information regarding DejaGnu test suites, see The DejaGnu Manual.

15.6 Install Tests

The installcheck target is available to the user as a way to run any tests after the package has been installed. You can add tests to this by writing an installcheck-local rule.

16 Rebuilding Makefiles

Automake generates rules to automatically rebuild Makefiles, configure, and other derived files like Makefile.in.

If you are using AM_MAINTAINER_MODE in configure.ac, then these automatic rebuilding rules are only enabled in maintainer mode.

Sometimes you need to run aclocal with an argument like -I to tell it where to find .m4 files. Since sometimes make will automatically run aclocal, you need a way to specify these arguments. You can do this by defining ACLOCAL_AMFLAGS; this holds arguments that are passed verbatim to aclocal. This variable is only useful in the top-level Makefile.am.

Sometimes it is convenient to supplement the rebuild rules for configure or config.status with additional dependencies. The variables CONFIGURE_DEPENDENCIES and CONFIG_STATUS_DEPENDENCIES can be used to list these extra dependencies. These variables should be defined in all Makefiles of the tree (because these two rebuild rules are
output in all them), so it is safer and easier to AC_SUBST them from configure.ac. For instance, the following statement will cause configure to be rerun each time version.sh is changed.

```
AC_SUBST([CONFIG_STATUS_DEPENDENCIES], ['$\$(top_srcdir)/version.sh'])
```

Note the ‘$(top_srcdir)/’ in the file name. Since this variable is to be used in all Makefiles, its value must be sensible at any level in the build hierarchy.

Beware not to mistake CONFIGURE_DEPENDENCIES for CONFIG_STATUS_DEPENDENCIES.

CONFIGURE_DEPENDENCIES adds dependencies to the configure rule, whose effect is to run autoconf. This variable should be seldom used, because automake already tracks m4_included files. However it can be useful when playing tricky games with m4_esyscmd or similar non-recommendable macros with side effects.

CONFIG_STATUS_DEPENDENCIES adds dependencies to the config.status rule, whose effect is to run configure. This variable should therefore carry any non-standard source that may be read as a side effect of running configure, like version.sh in the example above.

Speaking of version.sh scripts, we recommend against them today. They are mainly used when the version of a package is updated automatically by a script (e.g., in daily builds). Here is what some old-style configure.acs may look like:

```
AC_INIT
  . $srcdir/version.sh
AM_INIT_AUTOMAKE([name], $VERSION_NUMBER)
...
```

Here, version.sh is a shell fragment that sets VERSION_NUMBER. The problem with this example is that automake cannot track dependencies (listing version.sh in CONFIG_STATUS_DEPENDENCIES, and distributing this file is up to the user), and that it uses the obsolete form of AC_INIT and AM_INIT_AUTOMAKE. Upgrading to the new syntax is not straightforward, because shell variables are not allowed in AC_INIT’s arguments. We recommend that version.sh be replaced by an M4 file that is included by configure.ac:

```
m4.include([version.m4])
AC_INIT([name], VERSION_NUMBER)
AM_INIT_AUTOMAKE
...
```

Here version.m4 could contain something like ‘m4_define([VERSION_NUMBER], [1.2])’. The advantage of this second form is that automake will take care of the dependencies when defining the rebuild rule, and will also distribute the file automatically. An inconvenience is that autoconf will now be rerun each time the version number is bumped, when only configure had to be rerun in the previous setup.

17 Changing Automake’s Behavior
17.1 Options generalities

Various features of Automake can be controlled by options. Except where noted otherwise, options can be specified in one of several ways. Most options can be applied on a per-
Makefile basis when listed in a special Makefile variable named AUTOMAKE_OPTIONS. Some
of these options only make sense when specified in the toplevel Makefile.am file. Options
are applied globally to all processed Makefile files when listed in the first argument of AM_INIT_AUTOMAKE in configure.ac, and some options which require changes to the configure
script can only be specified there. These are annotated below.

As a general rule, options specified in AUTOMAKE_OPTIONS take precedence over those
specified in AM_INIT_AUTOMAKE, which in turn take precedence over those specified on the
command line.

Also, some care must be taken about the interactions among strictness level and warning
categories. As a general rule, strictness-implied warnings are overridden by those specified
by explicit options. For example, even if ‘portability’ warnings are disabled by default in
foreign strictness, an usage like this will end up enabling them:

AUTOMAKE_OPTIONS = -Wportability foreign

However, a strictness level specified in a higher-priority context will override all the ex-
plicit warnings specified in a lower-priority context. For example, if configure.ac contains:

AM_INIT_AUTOMAKE([-Wportability])

and Makefile.am contains:

AUTOMAKE_OPTIONS = foreign

then ‘portability’ warnings will be disabled in Makefile.am.

17.2 List of Automake options

gnits
gnu
foreign
cygnus

Set the strictness as appropriate. The gnits option also implies options
readme-alpha and check-news.

check-news

Cause ‘make dist’ to fail unless the current version number appears in the first
few lines of the NEWS file.

color-tests

Cause output of the serial and parallel test harnesses (see Section 15.2 [Simple
Tests], page 102) and of properly-written custom test drivers (see Section 15.3
[Custom Test Drivers], page 108) to be colorized on capable terminals.

dejagnu

Cause dejagnu-specific rules to be generated. See Section 15.5 [DejaGnu Tests],
page 116.

dist-bzip2

Hook dist-bzip2 to dist.
dist-lzip
Hook dist-lzip to dist.

dist-shar
Hook dist-shar to dist.

dist-zip
Hook dist-zip to dist.

dist-tarZ
Hook dist-tarZ to dist.

filename-length-max=99
Abort if file names longer than 99 characters are found during `make dist`. Such long file names are generally considered not to be portable in tarballs. See the `tar-v7` and `tar-ustar` options below. This option should be used in the top-level `Makefile.am` or as an argument of `AM_INIT_AUTOMAKE` in `configure.ac`, it will be ignored otherwise. It will also be ignored in sub-packages of nested packages (see Section 7.4 [Subpackages], page 52).

no-define
This option is meaningful only when passed as an argument to `AM_INIT_AUTOMAKE`. It will prevent the `PACKAGE` and `VERSION` variables from being `AC_DEFINE`d.

no-dependencies
This is similar to using `--ignore-deps` on the command line, but is useful for those situations where you don’t have the necessary bits to make automatic dependency tracking work (see Section 8.19 [Dependencies], page 80). In this case the effect is to effectively disable automatic dependency tracking.

no-dist
Don’t emit any code related to `dist` target. This is useful when a package has its own method for making distributions.

no-dist-gzip
Do not hook `dist-gzip` to `dist`.

no-exeext
If your `Makefile.am` defines a rule for target `foo`, it will override a rule for a target named `foo$(EXEEXT)`. This is necessary when `EXEEXT` is found to be empty. However, by default `automake` will generate an error for this use. The `no-exeext` option will disable this error. This is intended for use only where it is known in advance that the package will not be ported to Windows, or any other operating system using extensions on executables.

no-installinfo
The generated `Makefile.in` will not cause info pages to be built or installed by default. However, `info` and `install-info` targets will still be available. This option is disallowed at `gnu` strictness and above.

no-installman
The generated `Makefile.in` will not cause man pages to be installed by default. However, an `install-man` target will still be available for optional installation. This option is disallowed at `gnu` strictness and above.
nostdinc  This option can be used to disable the standard -I options that are ordinarily automatically provided by Automake.

no-texinfo.tex  
  Don’t require texinfo.tex, even if there are texinfo files in this directory.

parallel-tests  
  Enable test suite harness for TESTS that can run tests in parallel (see Section 15.2.3 [Parallel Test Harness], page 105, for more information).

serial-tests  
  Enable the older serial test suite harness for TESTS (see Section 15.2.2 [Serial Test Harness], page 104, for more information). This is still the default for the moment.

readme-alpha  
  If this release is an alpha release, and the file README-alpha exists, then it will be added to the distribution. If this option is given, version numbers are expected to follow one of two forms. The first form is ‘major.minor.alpha’, where each element is a number; the final period and number should be left off for non-alpha releases. The second form is ‘major.minormalpha’, where alpha is a letter; it should be omitted for non-alpha releases.

silent-rules  
  Enable less verbose build rules. This can be used to let build rules output status lines of the form:

        GEN output-file
        CC object-file

  instead of printing the command that will be executed to update output-file or to compile object-file. It can also silence libtool output.

  For more information about how to use, enable, or disable silent rules, see Section 21.3 [Automake silent-rules Option], page 127.

std-options  
  Make the installcheck rule check that installed scripts and programs support the --help and --version options. This also provides a basic check that the program’s run-time dependencies are satisfied after installation.

  In a few situations, programs (or scripts) have to be exempted from this test. For instance, false (from GNU coreutils) is never successful, even for --help or --version. You can list such programs in the variable AM_INSTALLCHECK_STD_OPTIONS_EXEMPT. Programs (not scripts) listed in this variable should be suffixed by ‘$(EXEEXT)’ for the sake of Windows or OS/2. For instance, suppose we build false as a program but true.sh as a script, and that neither of them support --help or --version:

        AUTOMAKE_OPTIONS = std-options
        bin_PROGRAMS = false ...
        bin_SCRIPTS = true.sh ...
        AM_INSTALLCHECK_STD_OPTIONS_EXEMPT = false$(EXEEXT) true.sh
subdir-objects
If this option is specified, then objects are placed into the subdirectory of the
build directory corresponding to the subdirectory of the source file. For in-
stance, if the source file is subdir/file.cxx, then the output file would be
subdir/file.o.
In order to use this option with C sources, you should add AM_PROG_CC_C_0 to
configure.ac.

\texttt{tar-v7} \hfill \texttt{tar-ustar} \hfill \texttt{tar-pax}

These three mutually exclusive options select the tar format to use when gen-
erating tarballs with \texttt{make dist}. (The tar file created is then compressed
according to the set of no-dist-gzip, dist-bzip2, dist-lzip, dist-xz and
dist-tarZ options in use.)
These options must be passed as arguments to \texttt{AM_INIT_AUTOMAKE} (see
Section 6.4 [Macros], page 44 because they can require additional configure
checks. Automake will complain if it sees such options in an AUTOMAKE_OPTIONS
variable.
\texttt{tar-v7} selects the old V7 tar format. This is the historical default. This anti-
quated format is understood by all tar implementations and supports file names
with up to 99 characters. When given longer file names some tar implementa-
tions will diagnose the problem while other will generate broken tarballs or use
non-portable extensions. Furthermore, the V7 format cannot store empty direc-
tories. When using this format, consider using the filename-length-max=99
option to catch file names too long.
\texttt{tar-ustar} selects the ustar format defined by POSIX 1003.1-1988. This format
is believed to be old enough to be portable. It fully supports empty directories.
It can store file names with up to 256 characters, provided that the file name
can be split at directory separator in two parts, first of them being at most 155
bytes long. So, in most cases the maximum file name length will be shorter
than 256 characters. However you may run against broken tar implementations
that incorrectly handle file names longer than 99 characters (please report them
to bug-automake@gnu.org so we can document this accurately).
\texttt{tar-pax} selects the new pax interchange format defined by POSIX 1003.1-2001.
It does not limit the length of file names. However, this format is very young
and should probably be restricted to packages that target only very modern
platforms. There are moves to change the pax format in an upward-compatible
way, so this option may refer to a more recent version in the future.
See Section “Controlling the Archive Format” in GNU Tar, for further discus-
sion about tar formats.
configure knows several ways to construct these formats. It will not abort if
it cannot find a tool up to the task (so that the package can still be built), but
\texttt{make dist} will fail.
version
A version number (e.g., ‘0.30’) can be specified. If Automake is not newer than
the version specified, creation of the Makefile.in will be suppressed.
-Wcategory or --warnings=category
These options behave exactly like their command-line counterpart (see Chapter 5 [automake Invocation], page 26). This allows you to enable or disable some warning categories on a per-file basis. You can also setup some warnings for your entire project; for instance, try ‘AM_INIT_AUTOMAKE([-Wall])’ in your configure.ac.

Unrecognized options are diagnosed by automake.

If you want an option to apply to all the files in the tree, you can use the AM_INIT_AUTOMAKE macro in configure.ac. See Section 6.4 [Macros], page 44.

18 Miscellaneous Rules

There are a few rules and variables that didn’t fit anywhere else.

18.1 Interfacing to etags

Automake will generate rules to generate TAGS files for use with GNU Emacs under some circumstances.

If any C, C++ or Fortran 77 source code or headers are present, then tags and TAGS rules will be generated for the directory. All files listed using the _SOURCES, _HEADERS, and _LISP primaries will be used to generate tags. Note that generated source files that are not distributed must be declared in variables like nodist_noinst_HEADERS or nodist_progsOURCES or they will be ignored.

A tags rule will be output at the topmost directory of a multi-directory package. When run from this topmost directory, ‘make tags’ will generate a TAGS file that includes by reference all TAGS files from subdirectories.

The tags rule will also be generated if the variable ETAGS_ARGS is defined. This variable is intended for use in directories that contain taggable source that etags does not understand. The user can use the ETAGSFLAGS to pass additional flags to etags; AM_ETAGSFLAGS is also available for use in Makefile.am.

Automake also generates a ctags rule that can be used to build vi-style tags files. The variable CTAGS is the name of the program to invoke (by default ctags); CTAGSFLAGS can be used by the user to pass additional flags, and AM_CTAGSFLAGS can be used by the Makefile.am.

Automake will also generate an ID rule that will run mkid on the source. This is only supported on a directory-by-directory basis.

Similarly, the cscope rule will create a list of all the source files in the tree and run cscope to build an inverted index database. The variable CSCOPE is the name of the program to
Chapter 19: Include

invoke (by default `cscope`); `CSCOPEFLAGS` and `CSCOPE_ARGS` can be used by the user to pass additional flags and file names respectively, while `AM_CSCOPEFLAGS` can be used by the `Makefile.am`. Note that, currently, the Automake-provided `cscope` support, when used in a VPATH build, might not work well with non-GNU make implementations (especially with make implementations performing Section “VPATH rewrites” in *The Autoconf Manual*).

Finally, Automake also emits rules to support the GNU Global Tags program (http://www.gnu.org/software/global/). The `GTAGS` rule runs Global Tags and puts the result in the top build directory. The variable `GTAGS_ARGS` holds arguments that are passed to `gtags`.

### 18.2 Handling new file extensions

It is sometimes useful to introduce a new implicit rule to handle a file type that Automake does not know about.

For instance, suppose you had a compiler that could compile `.foo` files to `.o` files. You would simply define a suffix rule for your language:

```
.foo.o:
    fucc -c -o $@ $<
```

Then you could directly use a `.foo` file in a `_SOURCES` variable and expect the correct results:

```
bin_PROGRAMS = doit
doit_SOURCES = doit.foo
```

This was the simpler and more common case. In other cases, you will have to help Automake to figure out which extensions you are defining your suffix rule for. This usually happens when your extension does not start with a dot. Then, all you have to do is to put a list of new suffixes in the `SUFFIXES` variable before you define your implicit rule.

For instance, the following definition prevents Automake from misinterpreting the `.'idlC.cpp:' rule as an attempt to transform `.idlC` files into `.cpp` files.

```
SUFFIXES = .idl C.cpp
.idlC.cpp:
    # whatever
```

As you may have noted, the `SUFFIXES` variable behaves like the `.SUFFIXES` special target of `make`. You should not touch `.SUFFIXES` yourself, but use `SUFFIXES` instead and let Automake generate the suffix list for `.SUFFIXES`. Any given `SUFFIXES` go at the start of the generated suffixes list, followed by Automake generated suffixes not already in the list.

### 19 Include

Automake supports an `include` directive that can be used to include other `Makefile` fragments when `automake` is run. Note that these fragments are read and interpreted by `automake`, not by `make`. As with conditionals, `make` has no idea that `include` is in use.

There are two forms of `include`:

```
include $(srcdir)/file
```

Include a fragment that is found relative to the current source directory.
include $(top_srcdir)/file

Include a fragment that is found relative to the top source directory.

Note that if a fragment is included inside a conditional, then the condition applies to the entire contents of that fragment.

Makefile fragments included this way are always distributed because they are needed to rebuild Makefile.in.

## 20 Conditionals

Automake supports a simple type of conditionals.

These conditionals are not the same as conditionals in GNU Make. Automake conditionals are checked at configure time by the configure script, and affect the translation from Makefile.in to Makefile. They are based on options passed to configure and on results that configure has discovered about the host system. GNU Make conditionals are checked at make time, and are based on variables passed to the make program or defined in the Makefile.

Automake conditionals will work with any make program.

### 20.1 Usage of Conditionals

Before using a conditional, you must define it by using AM_CONDITIONAL in the configure.ac file (see Section 6.4 [Macros], page 44).

```
AM_CONDITIONAL (conditional, condition) [Macro]
The conditional name, conditional, should be a simple string starting with a letter and containing only letters, digits, and underscores. It must be different from ‘TRUE’ and ‘FALSE’ that are reserved by Automake.
The shell condition (suitable for use in a shell if statement) is evaluated when configure is run. Note that you must arrange for every AM_CONDITIONAL to be invoked every time configure is run. If AM_CONDITIONAL is run conditionally (e.g., in a shell if statement), then the result will confuse automake.

Conditionals typically depend upon options that the user provides to the configure script. Here is an example of how to write a conditional that is true if the user uses the --enable-debug option.
```

```
AC_ARG_ENABLE([debug],
    [ --enable-debug Turn on debugging],
    [case "${enableval}" in
        yes) debug=true ;;
        no) debug=false ;;
        *) AC_MSG_ERROR([bad value ${enableval} for --enable-debug]) ;;
    esac], [debug=false])

AM_CONDITIONAL([DEBUG], [test x$debug = xtrue])
```

Here is an example of how to use that conditional in Makefile.am:

```
if DEBUG
```
DBG = debug
else
DBG =
endif
noinst_PROGRAMS = $(DBG)

This trivial example could also be handled using EXTRA_PROGRAMS (see Section 8.1.4 [Conditional Programs], page 57).

You may only test a single variable in an if statement, possibly negated using ‘!’.

The else statement may be omitted. Conditionals may be nested to any depth. You
may specify an argument to else in which case it must be the negation of the condition used for
the current if. Similarly you may specify the condition that is closed on the endif line:

if DEBUG
DBG = debug
else !DEBUG
DBG =
endif !DEBUG

Unbalanced conditions are errors. The if, else, and endif statements should not be
indented, i.e., start on column one.

The else branch of the above two examples could be omitted, since assigning the empty
string to an otherwise undefined variable makes no difference.

In order to allow access to the condition registered by AM_CONDITIONAL inside
call configure.ac, and to allow conditional AC_CONFIG_FILES, AM_COND_IF may be used:

AM_COND_IF (conditional, [if-true], [if-false]) [Macro]
If conditional is fulfilled, execute if-true, otherwise execute if-false. If either branch
contains AC_CONFIG_FILES, it will cause automake to output the rules for the respective
files only for the given condition.

AM_COND_IF macros may be nested when m4 quotation is used properly (see Section “M4

Here is an example of how to define a conditional config file:

AM_CONDITIONAL([SHELL_WRAPPER], [test "x$with_wrapper" = xtrue])
AM_COND_IF([SHELL_WRAPPER],
[AC_CONFIG_FILES([wrapper:wrapper.in])])

20.2 Limits of Conditionals

Conditionals should enclose complete statements like variables or rules definitions. Automake
cannot deal with conditionals used inside a variable definition, for instance, and is
not even able to diagnose this situation. The following example would not work:

# This syntax is not understood by Automake
AM_CPPFLAGS = \
  -DFEATURE_A \
if WANT_DEBUG \
  -DDEBUG \
endif
However the intended definition of `AM_CPPFLAGS` can be achieved with

```bash
if WANT_DEBUG
    DEBUGFLAGS = -DDEBUG
endif
AM_CPPFLAGS = -DFEATURE_A $(DEBUGFLAGS) -DFEATURE_B
```

or

```bash
if WANT_DEBUG
    AM_CPPFLAGS += -DDEBUG
endif
AM_CPPFLAGS += -DFEATURE_B
```

More details and examples of conditionals are described alongside various Automake features in this manual (see Section 7.2 [Conditional Subdirectories], page 48, see Section 8.1.3 [Conditional Sources], page 55, see Section 8.1.4 [Conditional Programs], page 57, see Section 8.3.3 [Conditional Libtool Libraries], page 59, see Section 8.3.4 [Conditional Libtool Sources], page 60).

## 21 Silencing `make`

### 21.1 Make is verbose by default

Normally, when executing the set of rules associated with a target, `make` prints each rule before it is executed. This behaviour, while having been in place for a long time, and being even mandated by the POSIX standard, starkly violates the “silence is golden” UNIX principle:\footnote{See also \url{http://catb.org/~esr/writings/taoup/html/ch11s09.html}.}

> When a program has nothing interesting or surprising to say, it should say nothing. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss and bother. Silence is golden.

In fact, while such verbosity of `make` can theoretically be useful to track bugs and understand reasons of failures right away, it can also hide warning and error messages from `make-invoked` tools, drowning them in a flood of uninteresting and seldom useful messages, and thus allowing them to go easily undetected.

This problem can be very annoying, especially for developers, who usually know quite well what’s going on behind the scenes, and for whom the verbose output from `make` ends up being mostly noise that hampers the easy detection of potentially important warning messages.
21.2 Standard and generic ways to silence make

Here we describe some common idioms/tricks to obtain a quieter make output, with their relative advantages and drawbacks. In the next section (Section 21.3 [Automake silent-rules Option], page 127) we’ll see how Automake can help in this respect.

- **make -s**
  This simply causes make not to print any rule before executing it. The -s flag is mandated by POSIX, universally supported, and its purpose and function are easy to understand.
  But it also has its serious limitations too. First of all, it embodies an “all or nothing” strategy, i.e., either everything is silenced, or nothing is; this lack of granularity can sometimes be a fatal flaw. Moreover, when the -s flag is used, the make output might turn out to be too much terse; in case of errors, the user won’t be able to easily see what rule or command have caused them, or even, in case of tools with poor error reporting, what the errors were!

- **make >/dev/null || make**
  Apparently, this perfectly obeys the “silence is golden” rule: warnings from stderr are passed through, output reporting is done only in case of error, and in that case it should provide a verbose-enough report to allow an easy determination of the error location and causes.
  However, calling make two times in a row might hide errors (especially intermittent ones), or subtly change the expected semantic of the make calls — things which can clearly make debugging and error assessment very difficult.

- **make --no-print-directory**
  This is GNU make specific. When called with the --no-print-directory option, GNU make will disable printing of the working directory by invoked sub-makes (the well-known “Entering/Leaving directory...” messages). This helps to decrease the verbosity of the output, but experience has shown that it can also often render debugging considerably harder in projects using deeply-nested make recursion.
  As an aside, notice that the --no-print-directory option is automatically activated if the -s flag is used.

21.3 How Automake can help in silencing make

The tricks and idioms for silencing make described in the previous section can be useful from time to time, but we’ve seen that they all have their serious drawbacks and limitations. That’s why automake provides support for a more advanced and flexible way of obtaining quieter output from make: the silent-rules mode.

To give the gist of what silent-rules can do, here is a simple comparison between a typical make output (where silent rules are disabled) and one with silent rules enabled:

```
% cat Makefile.am
bin_PROGRAMS = foo
foo_SOURCES = main.c func.c
%
% cat main.c
int main (void) { return func (); } /* func used undeclared */
```
The make output is by default very verbose. This causes warnings from the compiler to be somewhat hidden, and not immediate to spot. Silent rules enabled: the output is minimal but informative. In particular, the warnings from the compiler stick out very clearly.

Also, in projects using libtool, the use of silent rules can automatically enable the libtool's --silent option:
Chapter 21: Silencing make

Let’s now see how the silent-rules mode interfaces with the package developer and the package user.

To enable the use of silent-rules in his package, a developer needs to do either of the following:

- Add the silent-rules option as argument to AM_INIT_AUTOMAKE.
- Call the AM_SILENT_RULES macro from within the configure.ac file.

It is not possible to instead specify silent-rules in a Makefile.am file.

If the developer has done either of the above, then the user of the package may influence the verbosity at configure run time as well as at make run time:

- Passing --enable-silent-rules to configure will cause build rules to be less verbose; the option --disable-silent-rules will cause normal verbose output.
- At make run time, the default chosen at configure time may be overridden: make V=1 will produce verbose output, make V=0 less verbose output.

Note that silent rules are disabled by default; the user must enable them explicitly at either configure run time or at make run time. We think that this is a good policy, since it provides the casual user with enough information to prepare a good bug report in case anything breaks.

Still, notwithstanding the rationales above, a developer who wants to make silent rules enabled by default in his own package can do so by adding a ‘yes’ argument to the AM_SILENT_RULES call in configure.ac. We advise against this approach, though.

Users who prefer to have silent rules enabled by default can edit their config.site file to make the variable enable_silent_rules default to ‘yes’. This should still allow disabling silent rules at configure time and at make time.

For portability to different make implementations, package authors are advised to not set the variable V inside the Makefile.am file, to allow the user to override the value for subdirectories as well.

The current implementation of this feature normally uses nested variable expansion ‘$((V))’, a Makefile feature that is not required by POSIX 2008 but is widely supported in practice. The silent-rules option thus turns off warnings about recursive variable expansion, which are in turn enabled by -Wportability (see Chapter 5 [automake Invocation], page 26). On the rare make implementations that do not support nested variable expansion, whether rules are silent is always determined at configure time, and cannot
be overridden at make time. Future versions of POSIX are likely to require nested variable expansion, so this minor limitation should go away with time.

To extend the silent mode to your own rules, you have few choices:

- You can use the predefined variable `AM_V_GEN` as a prefix to commands that should output a status line in silent mode, and `AM_V_at` as a prefix to commands that should not output anything in silent mode. When output is to be verbose, both of these variables will expand to the empty string.
- You can silence a recipe unconditionally with `@`, and then use the predefined variable `AM_V_P` to know whether make is being run in silent or verbose mode, adjust the verbose information your recipe displays accordingly:

```bash
generate-headers:
  ... [commands defining a shell variable '$headers'] ...; \
  if $(AM_V_P); then set -x; else echo " GEN [headers]"; fi; \n  rm -f $$headers && generate-header --flags $$headers
```
- You can add your own variables, so strings of your own choice are shown. The following snippet shows how you would define your own equivalent of `AM_V_GEN`:

```bash
pkg_verbose = $(pkg_verbose_@AM_V@)
pkg_verbose_ = $(pkg_verbose_@AM_DEFAULT_V@)
pkg_verbose_0 = @echo PKG-GEN $@;

foo: foo.in
  $(pkg_verbose)cp $(srcdir)/foo.in $@
```

As a final note, observe that, even when silent rules are enabled, the `--no-print-directory` option is still required with GNU `make` if the “Entering/Leaving directory ...” messages are to be disabled.

### 22 The effect of `--gnu` and `--gnits`

The `--gnu` option (or `gnu` in the `AUTOMAKE_OPTIONS` variable) causes `automake` to check the following:

- The files `INSTALL`, `NEWS`, `README`, `AUTHORS`, and `ChangeLog`, plus one of `COPYING.LIB`, `COPYING.LESSER` or `COPYING`, are required at the topmost directory of the package.

  If the `--add-missing` option is given, `automake` will add a generic version of the `INSTALL` file as well as the `COPYING` file containing the text of the current version of the GNU General Public License existing at the time of this Automake release (version 3 as this is written, [http://www.gnu.org/copyleft/gpl.html](http://www.gnu.org/copyleft/gpl.html)). However, an existing `COPYING` file will never be overwritten by `automake`.

- The options `no-installman` and `no-installinfo` are prohibited.

Note that this option will be extended in the future to do even more checking; it is advisable to be familiar with the precise requirements of the GNU standards. Also, `--gnu` can require certain non-standard GNU programs to exist for use by various maintainer-only rules; for instance, in the future `pathchk` might be required for `make dist`. 
Chapter 23: The effect of \texttt{--cygnus}

The \texttt{--gnits} option does everything that \texttt{--gnu} does, and checks the following as well:

- \texttt{make installcheck} will check to make sure that the \texttt{--help} and \texttt{--version} really print a usage message and a version string, respectively. This is the \texttt{std-options} option (see Chapter 17 [Options], page 117).
- \texttt{make dist} will check to make sure the \texttt{NEWS} file has been updated to the current version.
- \texttt{VERSION} is checked to make sure its format complies with Gnits standards.
- If \texttt{VERSION} indicates that this is an alpha release, and the file \texttt{README-alpha} appears in the topmost directory of a package, then it is included in the distribution. This is done in \texttt{--gnits} mode, and no other, because this mode is the only one where version number formats are constrained, and hence the only mode where Automake can automatically determine whether \texttt{README-alpha} should be included.
- The file \texttt{THANKS} is required.

23 The effect of \texttt{--cygnus}

The features described in this section are deprecated; you must not use any of them in new code, and should remove their use from older but still maintained code: they will be withdrawn the next major Automake release (1.13).

Some packages, notably GNU GCC and GNU gdb, used to have a build environment originally written at Cygnus Support (subsequently renamed Cygnus Solutions, and then later purchased by Red Hat). Packages with this ancestry are sometimes referred to as "Cygnus" trees.

A Cygnus tree has slightly different rules for how a \texttt{Makefile.in} is to be constructed. Passing \texttt{--cygnus} to \texttt{automake} will cause any generated \texttt{Makefile.in} to comply with Cygnus rules.

Here are the precise effects of \texttt{--cygnus}:

- The \texttt{foreign} strictness is implied.
- The options \texttt{no-installinfo}, \texttt{no-dependencies} and \texttt{no-dist} are implied (see Chapter 17 [Options], page 117).
- The macro \texttt{AM_MAINTAINER_MODE} is required.
- Info files are always created in the build directory, and not in the source directory. Packages that don’t use the \texttt{cygnus} option can emulate this effect by using the \texttt{no-installinfo} option and listing the generated info files in the \texttt{CLEANFILES} variable.
- \texttt{texinfo.tex} is not required if a Texinfo source file is specified. The assumption is that the file will be supplied, but in a place that Automake cannot find – it is an artifact of how Cygnus packages are typically bundled. This effect can be emulated in packages not using the \texttt{cygnus} option with a proper definition of the \texttt{TEXINFO_TEX} variable (see Section 11.1 [Texinfo], page 91).
- Certain tools will be searched for in the build tree as well as in the user’s \texttt{PATH}. These tools are \texttt{runtest}, \texttt{expect}, \texttt{makeinfo} and \texttt{texi2dvi}.
- The \texttt{check} target doesn’t depend on \texttt{all}. 

24 When Automake Isn’t Enough

In some situations, where Automake is not up to one task, one has to resort to handwritten rules or even handwritten Makefiles.

24.1 Extending Automake Rules

With some minor exceptions (for example _PROGRAMS variables, TESTS, or XFAIL_TESTS being rewritten to append ‘$(EXEEXT)’), the contents of a Makefile.am is copied to Makefile.in verbatim.

These copying semantics mean that many problems can be worked around by simply adding some make variables and rules to Makefile.am. Automake will ignore these additions.

Since a Makefile.in is built from data gathered from three different places (Makefile.am, configure.ac, and automake itself), it is possible to have conflicting definitions of rules or variables. When building Makefile.in the following priorities are respected by automake to ensure the user always has the last word:

- User defined variables in Makefile.am have priority over variables AC_SUBSTed from configure.ac, and AC_SUBSTed variables have priority over automake-defined variables.
- As far as rules are concerned, a user-defined rule overrides any automake-defined rule for the same target.

These overriding semantics make it possible to fine tune some default settings of Automake, or replace some of its rules. Overriding Automake rules is often inadvisable, particularly in the topmost directory of a package with subdirectories. The -Woverride option (see Chapter 5 [automake Invocation], page 26) comes in handy to catch overridden definitions.

Note that Automake does not make any distinction between rules with commands and rules that only specify dependencies. So it is not possible to append new dependencies to an automake-defined target without redefining the entire rule.

However, various useful targets have a ‘-local’ version you can specify in your Makefile.am. Automake will supplement the standard target with these user-supplied targets.

The targets that support a local version are all, info, dvi, ps, pdf, html, check, install-data, install-dvi, install-exec, install-html, install-info, install-pdf, install-ps, uninstall, installdirs, installcheck and the various clean targets (mostlyclean, clean, distclean, and maintainer-clean).

Note that there are no uninstall-exec-local or uninstall-data-local targets; just use uninstall-local. It doesn’t make sense to uninstall just data or just executables.

For instance, here is one way to erase a subdirectory during ‘make clean’ (see Chapter 13 [Clean], page 96).

    clean-local:
    -rm -rf testSubDir

You may be tempted to use install-data-local to install a file to some hard-coded location, but you should avoid this (see Section 28.10 [Hard-Coded Install Paths], page 154).
With the \texttt{-local} targets, there is no particular guarantee of execution order; typically, they are run early, but with parallel make, there is no way to be sure of that.

In contrast, some rules also have a way to run another rule, called a \textit{hook}; hooks are always executed after the main rule's work is done. The hook is named after the principal target, with \texttt{-hook} appended. The targets allowing hooks are \texttt{install-data}, \texttt{install-exec}, \texttt{uninstall}, \texttt{dist}, and \texttt{distcheck}.

For instance, here is how to create a hard link to an installed program:

\begin{verbatim}
install-exec-hook:
    ln $(DESTDIR)$(bindir)/program$(EXEEXT) \\
    $(DESTDIR)$(bindir)/progl ink$(EXEEXT)
\end{verbatim}

Although cheaper and more portable than symbolic links, hard links will not work everywhere (for instance, OS/2 does not have \texttt{ln}). Ideally you should fall back to \texttt{cp -p} when \texttt{ln} does not work. An easy way, if symbolic links are acceptable to you, is to add \texttt{AC_PROG_LN_S} to \texttt{configure.ac} (see Section “Particular Program Checks” in \textit{The Autoconf Manual}) and use \texttt{$(LN_S)$} in \texttt{Makefile.am}.

For instance, here is how you could install a versioned copy of a program using \texttt{$(LN_S)$}:

\begin{verbatim}
install-exec-hook:
    cd $(DESTDIR)$(bindir) && \\
    mv -f prog$(EXEEXT) prog-$(VERSION)$(EXEEXT) && \\
    $(LN_S) prog-$(VERSION)$(EXEEXT) prog$(EXEEXT)
\end{verbatim}

Note that we rename the program so that a new version will erase the symbolic link, not the real binary. Also we \texttt{cd} into the destination directory in order to create relative links.

When writing \texttt{install-exec-hook} or \texttt{install-data-hook}, please bear in mind that the \texttt{exec/data} distinction is based on the installation directory, not on the primary used (see Section 12.2 [The Two Parts of Install], page 95). So a \texttt{foo_SCRIPTS} will be installed by \texttt{install-data}, and a \texttt{baret ec_SCRIPTS} will be installed by \texttt{install-exec}. You should define your hooks consequently.

### 24.2 Third-Party Makefiles

In most projects all \texttt{Makefiles} are generated by Automake. In some cases, however, projects need to embed subdirectories with handwritten \texttt{Makefiles}. For instance, one subdirectory could be a third-party project with its own build system, not using Automake.

It is possible to list arbitrary directories in \texttt{SUBDIRS} or \texttt{DIST_SUBDIRS} provided each of these directories has a \texttt{Makefile} that recognizes all the following recursive targets.

When a user runs one of these targets, that target is run recursively in all subdirectories. This is why it is important that even third-party \texttt{Makefiles} support them.

\begin{verbatim}
all Compile the entire package. This is the default target in Automake-generated
Makefiles, but it does not need to be the default in third-party Makefiles.
distdir Copy files to distribute into \texttt{\$(distdir)}, before a tarball is constructed. Of
    course this target is not required if the \texttt{no-dist} option (see Chapter 17 [Op-
    tions], page 117) is used.
\end{verbatim}

The variables \texttt{$(top_distdir)$} and \texttt{$(distdir)$} (see Section 14.3 [The dist
Hook], page 98) will be passed from the outer package to the subpackage when
the distdir target is invoked. These two variables have been adjusted for the directory that is being recursed into, so they are ready to use.

install
install-data
install-exec
uninstall

Install or uninstall files (see Chapter 12 [Install], page 94).

install-dvi
install-html
install-info
install-ps
install-pdf

Install only some specific documentation format (see Section 11.1 [Texinfo], page 91).

installdirs
Creates install directories, but do not install any files.

check
installcheck

Check the package (see Chapter 15 [Tests], page 101).

mostlyclean
clean
distclean
maintainer-clean

Cleaning rules (see Chapter 13 [Clean], page 96).

dvi
pdf
ps
info
html
Build the documentation in various formats (see Section 11.1 [Texinfo], page 91).

tags
ctags

Build TAGS and CTAGS (see Section 18.1 [Tags], page 122).

If you have ever used Gettext in a project, this is a good example of how third-party
Makefiles can be used with Automake. The Makefiles gettextize puts in the po/ and
intl/ directories are handwritten Makefiles that implement all of these targets. That way
they can be added to SUBDIRS in Automake packages.

Directories that are only listed in DIST_SUBDIRS but not in SUBDIRS need only the
distclean, maintainer-clean, and distdir rules (see Section 7.2 [Conditional Subdirectories], page 48).

Usually, many of these rules are irrelevant to the third-party subproject, but they are
required for the whole package to work. It’s OK to have a rule that does nothing, so if
you are integrating a third-party project with no documentation or tag support, you could
simply augment its Makefile as follows:

EMPTY_AUTOMAKE_TARGETS = dvi pdf ps info html tags ctags
Another aspect of integrating third-party build systems is whether they support VPATH builds (see Section 2.2.6 [VPATH Builds], page 6). Obviously if the subpackage does not support VPATH builds the whole package will not support VPATH builds. This in turns means that `make distcheck` will not work, because it relies on VPATH builds. Some people can live without this (actually, many Automake users have never heard of `make distcheck`). Other people may prefer to revamp the existing Makefiles to support VPATH. Doing so does not necessarily require Automake, only Autoconf is needed (see Section “Build Directories” in The Autoconf Manual). The necessary substitutions: `@srcdir@`, `@top_srcdir@`, and `@top_builddir@` are defined by configure when it processes a Makefile (see Section “Preset Output Variables” in The Autoconf Manual), they are not computed by the Makefile like the aforementioned `$(distdir)` and `$(top_distdir)` variables.

It is sometimes inconvenient to modify a third-party Makefile to introduce the above required targets. For instance, one may want to keep the third-party sources untouched to ease upgrades to new versions.

Here are two other ideas. If GNU make is assumed, one possibility is to add to that subdirectory a GNUmakefile that defines the required targets and includes the third-party Makefile. For this to work in VPATH builds, GNUmakefile must lie in the build directory; the easiest way to do this is to write a GNUmakefile.in instead, and have it processed with AC_CONFIG_FILES from the outer package. For example if we assume Makefile defines all targets except the documentation targets, and that the check target is actually called test, we could write GNUmakefile (or GNUmakefile.in) like this:

```bash
# First, include the real Makefile
include Makefile

# Then, define the other targets needed by Automake Makefiles.
.PHONY: dvi pdf ps info html check
dvi pdf ps info html:
check: test
```

A similar idea that does not use include is to write a proxy Makefile that dispatches rules to the real Makefile, either with `$MAKE -f Makefile.real $(AM_MAKEFLAGS) target` (if it’s OK to rename the original Makefile) or with `cd subdir && $MAKE $(AM_MAKEFLAGS) target` (if it’s OK to store the subdirectory project one directory deeper). The good news is that this proxy Makefile can be generated with Automake. All we need are -local targets (see Section 24.1 [Extending], page 132) that perform the dispatch. Of course the other Automake features are available, so you could decide to let Automake perform distribution or installation. Here is a possible Makefile.am:

```bash
all-local:
  cd subdir && $MAKE $(AM_MAKEFLAGS) all
check-local:
  cd subdir && $MAKE $(AM_MAKEFLAGS) test
clean-local:
  cd subdir && $MAKE $(AM_MAKEFLAGS) clean
```

# Assuming the package knows how to install itself
install-data-local:
  cd subdir && $(MAKE) $(AM_MAKEFLAGS) install-data
install-exec-local:
  cd subdir && $(MAKE) $(AM_MAKEFLAGS) install-exec
uninstall-local:
  cd subdir && $(MAKE) $(AM_MAKEFLAGS) uninstall

# Distribute files from here.
EXTRA_DIST = subdir/Makefile subdir/program.c ...

Pushing this idea to the extreme, it is also possible to ignore the subproject build system and build everything from this proxy Makefile.am. This might sound very sensible if you need VPATH builds but the subproject does not support them.

25 Distributing Makefile.ins

Automake places no restrictions on the distribution of the resulting Makefile.ins. We still encourage software authors to distribute their work under terms like those of the GPL, but doing so is not required to use Automake.

Some of the files that can be automatically installed via the --add-missing switch do fall under the GPL. However, these also have a special exception allowing you to distribute them with your package, regardless of the licensing you choose.

26 Automake API Versioning

New Automake releases usually include bug fixes and new features. Unfortunately they may also introduce new bugs and incompatibilities. This makes four reasons why a package may require a particular Automake version.

Things get worse when maintaining a large tree of packages, each one requiring a different version of Automake. In the past, this meant that any developer (and sometimes users) had to install several versions of Automake in different places, and switch ‘$PATH’ appropriately for each package.

Starting with version 1.6, Automake installs versioned binaries. This means you can install several versions of Automake in the same ‘$prefix’, and can select an arbitrary Automake version by running automake-1.6 or automake-1.7 without juggling with ‘$PATH’. Furthermore, Makefile’s generated by Automake 1.6 will use automake-1.6 explicitly in their rebuild rules.

The number ‘1.6’ in automake-1.6 is Automake’s API version, not Automake’s version. If a bug fix release is made, for instance Automake 1.6.1, the API version will remain 1.6. This means that a package that works with Automake 1.6 should also work with 1.6.1; after all, this is what people expect from bug fix releases.

If your package relies on a feature or a bug fix introduced in a release, you can pass this version as an option to Automake to ensure older releases will not be used. For instance, use this in your configure.ac:

    AM_INIT_AUTOMAKE([1.6.1]) dnl Require Automake 1.6.1 or better.
or, in a particular `Makefile.am`:

    AUTOMAKE_OPTIONS = 1.6.1  # Require Automake 1.6.1 or better.

Automake will print an error message if its version is older than the requested version.

**What is in the API**

Automake’s programming interface is not easy to define. Basically it should include at least all **documented** variables and targets that a `Makefile.am` author can use, any behavior associated with them (e.g., the places where ‘-hook’-s are run), the command line interface of `automake` and `aclocal`, ...

**What is not in the API**

Every undocumented variable, target, or command line option, is not part of the API. You should avoid using them, as they could change from one version to the other (even in bug fix releases, if this helps to fix a bug).

If it turns out you need to use such an undocumented feature, contact `automake@gnu.org` and try to get it documented and exercised by the test-suite.

### 27 Upgrading a Package to a Newer Automake Version

Automake maintains three kind of files in a package.

- `aclocal.m4`
- `Makefile.am`
- auxiliary tools like `install-sh` or `py-compile`

`aclocal.m4` is generated by `aclocal` and contains some Automake-supplied M4 macros. Auxiliary tools are installed by `automake --add-missing` when needed. `Makefile.am` are built from `Makefile.am` by `automake`, and rely on the definitions of the M4 macros put in `aclocal.m4` as well as the behavior of the auxiliary tools installed.

Because all of these files are closely related, it is important to regenerate all of them when upgrading to a newer Automake release. The usual way to do that is

    aclocal # with any option needed (such a -I m4)
    autoconf
    automake --add-missing --force-missing

or more conveniently:

    autoreconf -vfi

The use of `--force-missing` ensures that auxiliary tools will be overridden by new versions (see Chapter 5 [automake Invocation], page 26).

It is important to regenerate all of these files each time Automake is upgraded, even between bug fixes releases. For instance, it is not unusual for a bug fix to involve changes to both the rules generated in `Makefile.in` and the supporting M4 macros copied to `aclocal.m4`. 
Presently `automake` is able to diagnose situations where `aclocal.m4` has been generated with another version of `aclocal`. However it never checks whether auxiliary scripts are up-to-date. In other words, `automake` will tell you when `aclocal` needs to be rerun, but it will never diagnose a missing `--force-missing`.

Before upgrading to a new major release, it is a good idea to read the file `NEWS`. This file lists all changes between releases: new features, obsolete constructs, known incompatibilities, and workarounds.

## 28 Frequently Asked Questions about Automake

This chapter covers some questions that often come up on the mailing lists.

### 28.1 CVS and generated files

**Background: distributed generated Files**

Packages made with Autoconf and Automake ship with some generated files like `configure` or `Makefile.in`. These files were generated on the developer's host and are distributed so that end-users do not have to install the maintainer tools required to rebuild them. Other generated files like Lex scanners, Yacc parsers, or Info documentation, are usually distributed on similar grounds.

Automake outputs rules in `Makefiles` to rebuild these files. For instance, `make` will run `autoconf` to rebuild `configure` whenever `configure.ac` is changed. This makes development safer by ensuring a `configure` is never out-of-date with respect to `configure.ac`.

As generated files shipped in packages are up-to-date, and because `tar` preserves timestamps, these rebuild rules are not triggered when a user unpacks and builds a package.

**Background: CVS and Timestamps**

Unless you use CVS keywords (in which case files must be updated at commit time), CVS preserves timestamp during `cvs commit` and `cvs import -d` operations.

When you check out a file using `cvs checkout` its timestamp is set to that of the revision that is being checked out.

However, during `cvs update`, files will have the date of the update, not the original timestamp of this revision. This is meant to make sure that `make` notices sources files have been updated.

This timestamp shift is troublesome when both sources and generated files are kept under CVS. Because CVS processes files in lexical order, `configure.ac` will appear newer than `configure` after a `cvs update` that updates both files, even if `configure` was newer than `configure.ac` when it was checked in. Calling `make` will then trigger a spurious rebuild of `configure`.

**Living with CVS in Autoconfiscated Projects**

There are basically two clans amongst maintainers: those who keep all distributed files under CVS, including generated files, and those who keep generated files out of CVS.
All Files in CVS

- The CVS repository contains all distributed files so you know exactly what is distributed, and you can checkout any prior version entirely.
- Maintainers can see how generated files evolve (for instance, you can see what happens to your `Makefile.in` when you upgrade Automake and make sure they look OK).
- Users do not need the autotools to build a checkout of the project, it works just like a released tarball.
- If users use `cvs update` to update their copy, instead of `cvs checkout` to fetch a fresh one, timestamps will be inaccurate. Some rebuild rules will be triggered and attempt to run developer tools such as `autoconf` or `automake`.

Actually, calls to such tools are all wrapped into a call to the `missing` script discussed later (see Section 28.2 [maintainer-mode], page 140). `missing` will take care of fixing the timestamps when these tools are not installed, so that the build can continue.

- In distributed development, developers are likely to have different versions of the maintainer tools installed. In this case rebuilds triggered by timestamp lossage will lead to spurious changes to generated files. There are several solutions to this:
  - All developers should use the same versions, so that the rebuilt files are identical to files in CVS. (This starts to be difficult when each project you work on uses different versions.)
  - Or people use a script to fix the timestamp after a checkout (the GCC folks have such a script).
  - Or `configure.ac` uses `AM_MAINTAINER_MODE`, which will disable all of these rebuild rules by default. This is further discussed in Section 28.2 [maintainer-mode], page 140.

- Although we focused on spurious rebuilds, the converse can also happen. CVS’s timestamp handling can also let you think an out-of-date file is up-to-date.

For instance, suppose a developer has modified `Makefile.am` and has rebuilt `Makefile.in`, and then decides to do a last-minute change to `Makefile.am` right before checking in both files (without rebuilding `Makefile.in` to account for the change).

This last change to `Makefile.am` makes the copy of `Makefile.in` out-of-date. Since CVS processes files alphabetically, when another developer `cvs update`’s his or her tree, `Makefile.in` will happen to be newer than `Makefile.am`. This other developer will not see that `Makefile.in` is out-of-date.

Generated Files out of CVS

One way to get CVS and `make` working peacefully is to never store generated files in CVS, i.e., do not CVS-control files that are `Makefile` targets (also called `derived` files).

This way developers are not annoyed by changes to generated files. It does not matter if they all have different versions (assuming they are compatible, of course). And finally, timestamps are not lost, changes to sources files can’t be missed as in the `Makefile.am/Makefile.in` example discussed earlier.
The drawback is that the CVS repository is not an exact copy of what is distributed and that users now need to install various development tools (maybe even specific versions) before they can build a checkout. But, after all, CVS’s job is versioning, not distribution.

Allowing developers to use different versions of their tools can also hide bugs during distributed development. Indeed, developers will be using (hence testing) their own generated files, instead of the generated files that will be released actually. The developer who prepares the tarball might be using a version of the tool that produces bogus output (for instance a non-portable C file), something other developers could have noticed if they weren’t using their own versions of this tool.

Third-party Files

Another class of files not discussed here (because they do not cause timestamp issues) are files that are shipped with a package, but maintained elsewhere. For instance, tools like gettextize and autopoint (from Gettext) or libtoolize (from Libtool), will install or update files in your package.

These files, whether they are kept under CVS or not, raise similar concerns about version mismatch between developers’ tools. The Gettext manual has a section about this, see Section “Integrating with CVS” in GNU gettext tools.

28.2 missing and AM_MAINTAINER_MODE

missing

The missing script is a wrapper around several maintainer tools, designed to warn users if a maintainer tool is required but missing. Typical maintainer tools are autoconf, automake, bison, etc. Because file generated by these tools are shipped with the other sources of a package, these tools shouldn’t be required during a user build and they are not checked for in configure.

However, if for some reason a rebuild rule is triggered and involves a missing tool, missing will notice it and warn the user. Besides the warning, when a tool is missing, missing will attempt to fix timestamps in a way that allows the build to continue. For instance, missing will touch configure if autoconf is not installed. When all distributed files are kept under version control, this feature of missing allows a user with no maintainer tools to build a package off its version control repository, bypassing any timestamp inconsistency (implied by e.g. ‘cvs update’ or ‘git clone’).

If the required tool is installed, missing will run it and won’t attempt to continue after failures. This is correct during development: developers love fixing failures. However, users with wrong versions of maintainer tools may get an error when the rebuild rule is spuriously triggered, halting the build. This failure to let the build continue is one of the arguments of the AM_MAINTAINER_MODE advocates.

AM_MAINTAINER_MODE

AM_MAINTAINER_MODE allows you to choose whether the so called "rebuild rules" should be enabled or disabled. With AM_MAINTAINER_MODE([enable]), they are enabled by default, otherwise they are disabled by default. In the latter case, if you have AM_MAINTAINER_MODE in configure.ac, and run './configure && make', then make will *never* attempt to
rebuild configure, Makefile.ins, Lex or Yacc outputs, etc. I.e., this disables build rules for files that are usually distributed and that users should normally not have to update.

The user can override the default setting by passing either ‘--enable-maintainer-mode’ or ‘--disable-maintainer-mode’ to configure.

People use AM_MAINTAINER_MODE either because they do not want their users (or themselves) annoyed by timestamps lossage (see Section 28.1 [CVS], page 138), or because they simply can’t stand the rebuild rules and prefer running maintainer tools explicitly.

AM_MAINTAINER_MODE also allows you to disable some custom build rules conditionally. Some developers use this feature to disable rules that need exotic tools that users may not have available.

Several years ago François Pinard pointed out several arguments against this AM_MAINTAINER_MODE macro. Most of them relate to insecurity. By removing dependencies you get non-dependable builds: changes to source files can have no effect on generated files and this can be very confusing when unnoticed. He adds that security shouldn’t be reserved to maintainers (what --enable-maintainer-mode suggests), on the contrary. If one user has to modify a Makefile.am, then either Makefile.in should be updated or a warning should be output (this is what Automake uses missing for) but the last thing you want is that nothing happens and the user doesn’t notice it (this is what happens when rebuild rules are disabled by AM_MAINTAINER_MODE).

Jim Meyering, the inventor of the AM_MAINTAINER_MODE macro was swayed by François’s arguments, and got rid of AM_MAINTAINER_MODE in all of his packages.

Still many people continue to use AM_MAINTAINER_MODE, because it helps them working on projects where all files are kept under version control, and because missing isn’t enough if you have the wrong version of the tools.

28.3 Why doesn’t Automake support wildcards?

Developers are lazy. They would often like to use wildcards in Makefile.am, so that they would not need to remember to update Makefile.am every time they add, delete, or rename a file.

There are several objections to this:

- When using CVS (or similar) developers need to remember they have to run ‘cvs add’ or ‘cvs rm’ anyway. Updating Makefile.am accordingly quickly becomes a reflex.

Conversely, if your application doesn’t compile because you forgot to add a file in Makefile.am, it will help you remember to ‘cvs add’ it.

- Using wildcards makes it easy to distribute files by mistake. For instance, some code a developer is experimenting with (a test case, say) that should not be part of the distribution.

- Using wildcards it’s easy to omit some files by mistake. For instance, one developer creates a new file, uses it in many places, but forgets to commit it. Another developer then checks out the incomplete project and is able to run ‘make dist’ successfully, even though a file is missing. By listing files, ‘make dist’ will complain.

- Wildcards are not portable to some non-GNU make implementations, e.g., NetBSD make will not expand globs such as ‘*’ in prerequisites of a target.
• Finally, it’s really hard to forget to add a file to Makefile.am: files that are not listed in Makefile.am are not compiled or installed, so you can’t even test them.

Still, these are philosophical objections, and as such you may disagree, or find enough value in wildcards to dismiss all of them. Before you start writing a patch against Automake to teach it about wildcards, let’s see the main technical issue: portability.

Although ‘$(wildcard ...)’ works with GNU make, it is not portable to other make implementations.

The only way Automake could support $(wildcard ...) is by expending $(wildcard ...) when automake is run. The resulting Makefile.ins would be portable since they would list all files and not use ‘$(wildcard ...)’. However that means developers would need to remember to run automake each time they add, delete, or rename files.

Compared to editing Makefile.am, this is a very small gain. Sure, it’s easier and faster to type ‘automake; make’ than to type ‘emacs Makefile.am; make’. But nobody bothered enough to write a patch to add support for this syntax. Some people use scripts to generate file lists in Makefile.am or in separate Makefile fragments.

Even if you don’t care about portability, and are tempted to use ‘$(wildcard ...)’ anyway because you target only GNU Make, you should know there are many places where Automake needs to know exactly which files should be processed. As Automake doesn’t know how to expand ‘$(wildcard ...)’, you cannot use it in these places. ‘$(wildcard ...)’ is a black box comparable to AC_SUBSTed variables as far Automake is concerned.

You can get warnings about ‘$(wildcard ...)’ constructs using the -Wportability flag.

28.4 Limitations on File Names

Automake attempts to support all kinds of file names, even those that contain unusual characters or are unusually long. However, some limitations are imposed by the underlying operating system and tools.

Most operating systems prohibit the use of the null byte in file names, and reserve ‘/’ as a directory separator. Also, they require that file names are properly encoded for the user’s locale. Automake is subject to these limits.

Portable packages should limit themselves to POSIX file names. These can contain ASCII letters and digits, ‘_’, ‘.’, and ‘-’. File names consist of components separated by ‘/’. File name components cannot begin with ‘-’.

Portable POSIX file names cannot contain components that exceed a 14-byte limit, but nowadays it’s normally safe to assume the more-generous XOPEN limit of 255 bytes. POSIX limits file names to 255 bytes (XOPEN allows 1023 bytes), but you may want to limit a source tarball to file names of 99 bytes to avoid interoperability problems with old versions of tar.

If you depart from these rules (e.g., by using non-ASCII characters in file names, or by using lengthy file names), your installers may have problems for reasons unrelated to Automake. However, if this does not concern you, you should know about the limitations imposed by Automake itself. These limitations are undesirable, but some of them seem to be inherent to underlying tools like Autoconf, Make, M4, and the shell. They fall into three categories: install directories, build directories, and file names.
Chapter 28: Frequently Asked Questions about Automake

The following characters:

```
newline " # $ ' '\n```

should not appear in the names of install directories. For example, the operand of `configure`'s --prefix option should not contain these characters.

Build directories suffer the same limitations as install directories, and in addition should not contain the following characters:

```
& @ \n```

For example, the full name of the directory containing the source files should not contain these characters.

Source and installation file names like `main.c` are limited even further: they should conform to the POSIX/XOPEN rules described above. In addition, if you plan to port to non-POSIX environments, you should avoid file names that differ only in case (e.g., `makefile` and `Makefile`). Nowadays it is no longer worth worrying about the 8.3 limits of DOS file systems.

### 28.5 Errors with distclean

This is a diagnostic you might encounter while running `make distcheck`.

As explained in Section 14.4 [Checking the Distribution], page 99, `make distcheck` attempts to build and check your package for errors like this one.

`make distcheck` will perform a VPATH build of your package (see Section 2.2.6 [VPATH Builds], page 6), and then call `make distclean`. Files left in the build directory after `make distclean` has run are listed after this error.

This diagnostic really covers two kinds of errors:

- files that are forgotten by distclean;
- distributed files that are erroneously rebuilt.

The former left-over files are not distributed, so the fix is to mark them for cleaning (see Chapter 13 [Clean], page 96), this is obvious and doesn’t deserve more explanations.

The latter bug is not always easy to understand and fix, so let’s proceed with an example. Suppose our package contains a program for which we want to build a man page using `help2man`. GNU `help2man` produces simple manual pages from the --help and --version output of other commands (see Section “Overview” in The Help2man Manual). Because we don’t want to force our users to install `help2man`, we decide to distribute the generated man page using the following setup.

```
# This Makefile.am is bogus.
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo$(EXEEXT)
    help2man --output=foo.1 ./foo$(EXEEXT)
```

This will effectively distribute the man page. However, `make distcheck` will fail with:

**ERROR**: files left in build directory after distclean:
Why was foo.1 rebuilt? Because although distributed, foo.1 depends on a non-distributed built file: foo$(EXEEXT). foo$(EXEEXT) is built by the user, so it will always appear to be newer than the distributed foo.1.

‘make distcheck’ caught an inconsistency in our package. Our intent was to distribute foo.1 so users do not need to install help2man, however since this rule causes this file to be always rebuilt, users do need help2man. Either we should ensure that foo.1 is not rebuilt by users, or there is no point in distributing foo.1.

More generally, the rule is that distributed files should never depend on non-distributed built files. If you distribute something generated, distribute its sources.

One way to fix the above example, while still distributing foo.1 is to not depend on foo$(EXEEXT). For instance, assuming foo --version and foo --help do not change unless foo.c or configure.ac change, we could write the following Makefile.am:

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo.c $(top_srcdir)/configure.ac
     $(MAKE) $(AM_MAKEFLAGS) foo$(EXEEXT)
     help2man --output=foo.1 ./foo$(EXEEXT)
```

This way, foo.1 will not get rebuilt every time foo$(EXEEXT) changes. The make call makes sure foo$(EXEEXT) is up-to-date before help2man. Another way to ensure this would be to use separate directories for binaries and man pages, and set SUBDIRS so that binaries are built before man pages.

We could also decide not to distribute foo.1. In this case it’s fine to have foo.1 dependent upon foo$(EXEEXT), since both will have to be rebuilt. However it would be impossible to build the package in a cross-compilation, because building foo.1 involves an execution of foo$(EXEEXT).

Another context where such errors are common is when distributed files are built by tools that are built by the package. The pattern is similar:

```
distributed-file: built-tools distributed-sources
    build-command
```

should be changed to

```
distributed-file: distributed-sources
    $(MAKE) $(AM_MAKEFLAGS) built-tools
    build-command
```
or you could choose not to distribute distributed-file, if cross-compilation does not matter.

The points made through these examples are worth a summary:

- Distributed files should never depend upon non-distributed built files.
- Distributed files should be distributed with all their dependencies.
- If a file is intended to be rebuilt by users, then there is no point in distributing it.
Chapter 28: Frequently Asked Questions about Automake

For desperate cases, it’s always possible to disable this check by setting `distcleancheck_listfiles` as documented in Section 14.4 [Checking the Distribution], page 99. Make sure you do understand the reason why ‘make distcheck’ complains before you do this. `distcleancheck_listfiles` is a way to hide errors, not to fix them. You can always do better.

28.6 Flag Variables Ordering

What is the difference between `AM_CFLAGS`, `CFLAGS`, and `mumble_CFLAGS`?

Why does `automake` output `CPPFLAGS` after `AM_CPPFLAGS` on compile lines? Shouldn’t it be the converse?

My `configure` adds some warning flags into `CXXFLAGS`. In one `Makefile.am` I would like to append a new flag, however if I put the flag into `AM_CXXFLAGS` it is prepended to the other flags, not appended.

Compile Flag Variables

This section attempts to answer all the above questions. We will mostly discuss `CPPFLAGS` in our examples, but actually the answer holds for all the compile flags used in Automake: `CCASFLAGS`, `CFLAGS`, `CPPFLAGS`, `CXXFLAGS`, `FCFLAGS`, `FFLAGS`, `GCJFLAGS`, `LDFLAGS`, `LFLAGS`, `LIBTOOLFLAGS`, `OBJCFLAGS`, `OBJCXXFLAGS`, `RFLAGS`, `UPCFLAGS`, and `YFLAGS`.

`CPPFLAGS`, `AM_CPPFLAGS`, and `mumble_CPPFLAGS` are three variables that can be used to pass flags to the C preprocessor (actually these variables are also used for other languages like C++ or preprocessed Fortran). `CPPFLAGS` is the user variable (see Section 3.6 [User Variables], page 23), `AM_CPPFLAGS` is the Automake variable, and `mumble_CPPFLAGS` is the variable specific to the mumble target (we call this a per-target variable, see Section 8.4 [Program and Library Variables], page 64).

Automake always uses two of these variables when compiling C sources files. When compiling an object file for the mumble target, the first variable will be `mumble_CPPFLAGS` if it is defined, or `AM_CPPFLAGS` otherwise. The second variable is always `CPPFLAGS`.

In the following example,

```bash
bin_PROGRAMS = foo bar
foo_SOURCES = xyz.c
bar_SOURCES = main.c
foo_CPPFLAGS = -DFOO
AM_CPPFLAGS = -DBAZ
```

`xyz.o` will be compiled with `${(foo_CPPFLAGS) $(CPPFLAGS)}`, (because `xyz.o` is part of the `foo` target), while `main.o` will be compiled with `${(AM_CPPFLAGS) $(CPPFLAGS)}` (because there is no per-target variable for target `bar`).

The difference between `mumble_CPPFLAGS` and `AM_CPPFLAGS` being clear enough, let’s focus on `CPPFLAGS`. `CPPFLAGS` is a user variable, i.e., a variable that users are entitled to modify in order to compile the package. This variable, like many others, is documented at the end of the output of ‘configure --help’.
For instance, someone who needs to add /home/my/usr/include to the C compiler’s search path would configure a package with

```
./configure CPPFLAGS='-I /home/my/usr/include'
```

and this flag would be propagated to the compile rules of all Makefiles.

It is also not uncommon to override a user variable at make-time. Many installers do this with prefix, but this can be useful with compiler flags too. For instance, if, while debugging a C++ project, you need to disable optimization in one specific object file, you can run something like

```
rm file.o
make CXXFLAGS=-O0 file.o
make
```

The reason ‘$(CPPFLAGS)’ appears after ‘$(AM_CPPFLAGS)’ or ‘$(mumble_CPPFLAGS)’ in the compile command is that users should always have the last say. It probably makes more sense if you think about it while looking at the ‘CXXFLAGS=-O0’ above, which should supersede any other switch from AM_CXXFLAGS or mumble_CXXFLAGS (and this of course replaces the previous value of CXXFLAGS).

You should never redefine a user variable such as CPPFLAGS in Makefile.am. Use ‘automake --override’ to diagnose such mistakes. Even something like

```
CPPFLAGS = -DDATADIR="$(datadir)" @CPPFLAGS@
```

is erroneous. Although this preserves configure’s value of CPPFLAGS, the definition of DATADIR will disappear if a user attempts to override CPPFLAGS from the make command line.

```
AM_CPPFLAGS = -DDATADIR="$(datadir)"
```

is all that is needed here if no per-target flags are used.

You should not add options to these user variables within configure either, for the same reason. Occasionally you need to modify these variables to perform a test, but you should reset their values afterwards. In contrast, it is OK to modify the ‘AM_’ variables within configure if you AC_SUBST them, but it is rather rare that you need to do this, unless you really want to change the default definitions of the ‘AM_’ variables in all Makefiles.

What we recommend is that you define extra flags in separate variables. For instance, you may write an Autoconf macro that computes a set of warning options for the C compiler, and AC_SUBST them in WARNINGCFLAGS; you may also have an Autoconf macro that determines which compiler and which linker flags should be used to link with library libfoo, and AC_SUBST these in LIBFOOCFLAGS and LIBFOOLDFLAGS. Then, a Makefile.am could use these variables as follows:

```
AM_CFLAGS = $(WARNINGCFLAGS)
bin_PROGRAMS = prog1 prog2
prog1_SOURCES = ...
prog2_SOURCES = ...
prog2_CFLAGS = $(LIBFOOCFLAGS) $(AM_CFLAGS)
prog2_LDFLAGS = $(LIBFOOLDFLAGS)
```

In this example both programs will be compiled with the flags substituted into ‘$(WARNINGCFLAGS)’, and prog2 will additionally be compiled with the flags required to link with libfoo.
Chapter 28: Frequently Asked Questions about Automake

Note that listing AM_CFLAGS in a per-target CFLAGS variable is a common idiom to ensure that AM_CFLAGS applies to every target in a Makefile.in.

Using variables like this gives you full control over the ordering of the flags. For instance, if there is a flag in $(WARNINGCFLAGS) that you want to negate for a particular target, you can use something like 'progi_CFLAGS = $(AM_CFLAGS) -no-flag'. If all of these flags had been forcefully appended to CFLAGS, there would be no way to disable one flag. Yet another reason to leave user variables to users.

Finally, we have avoided naming the variable of the example LIBFOO_LDFLAGS (with an underscore) because that would cause Automake to think that this is actually a per-target variable (like mumble_LDFLAGS) for some non-declared LIBFOO target.

Other Variables

There are other variables in Automake that follow similar principles to allow user options. For instance, Texinfo rules (see Section 11.1 [Texinfo], page 91) use MAKEINFOFLAGS and AM_MAKEINFOFLAGS. Similarly, DejaGnu tests (see Section 15.5 [DejaGnu Tests], page 116) use RUNTESTDEFAULTFLAGS and AM_RUNTESTDEFAULTFLAGS. The tags and ctags rules (see Section 18.1 [Tags], page 122) use ETAGSFLAGS, AM_ETAGSFLAGS, CTAGSFLAGS, and AM_CTAGSFLAGS. Java rules (see Section 10.4 [Java], page 88) use JAVACFLAGS and AM_JAVACFLAGS. None of these rules support per-target flags (yet).

To some extent, even AM_MAKEFLAGS (see Section 7.1 [Subdirectories], page 48) obeys this naming scheme. The slight difference is that MAKEFLAGS is passed to sub-makes implicitly by make itself.

However you should not think that all variables ending with FLAGS follow this convention. For instance, DISTCHECK_CONFIGURE_FLAGS (see Section 14.4 [Checking the Distribution], page 99) and ACLOCAL_AMFLAGS (see Chapter 16 [Rebuilding], page 116, and Section 6.3.4 [Local Macros], page 41), are two variables that are only useful to the maintainer and have no user counterpart.

ARFLAGS (see Section 8.2 [A Library], page 57) is usually defined by Automake and has neither AM_ nor per-target cousin.

Finally you should not think that the existence of a per-target variable implies the existence of an AM_ variable or of a user variable. For instance, the mumble_LDADD per-target variable overrides the makefile-wide LDADD variable (which is not a user variable), and mumble_LIBADD exists only as a per-target variable. See Section 8.4 [Program and Library Variables], page 64.

28.7 Why are object files sometimes renamed?

This happens when per-target compilation flags are used. Object files need to be renamed just in case they would clash with object files compiled from the same sources, but with different flags. Consider the following example.

```bash
bin_PROGRAMS = true false
ture_SOURCES = generic.c
tue_CPPFLAGS = -DEXIT_CODE=0
alse_SOURCES = generic.c
alse_CPPFLAGS = -DEXIT_CODE=1
```
Obviously the two programs are built from the same source, but it would be bad if they shared the same object, because `generic.o` cannot be built with both `-DEXIT_CODE=0` and `-DEXIT_CODE=1`. Therefore `automake` outputs rules to build two different objects: `true-generic.o` and `false-generic.o`.

`automake` doesn’t actually look whether source files are shared to decide if it must rename objects. It will just rename all objects of a target as soon as it sees per-target compilation flags used.

It’s OK to share object files when per-target compilation flags are not used. For instance, `true` and `false` will both use `version.o` in the following example.

```
AM_CPPFLAGS = -DVERSION=1.0
bin_PROGRAMS = true false
true_SOURCES = true.c version.c
false_SOURCES = false.c version.c
```

Note that the renaming of objects is also affected by the `_SHORTNAME` variable (see Section 8.4 [Program and Library Variables], page 64).

### 28.8 Per-Object Flags Emulation

One of my source files needs to be compiled with different flags. How do I do?

Automake supports per-program and per-library compilation flags (see Section 8.4 [Program and Library Variables], page 64, and Section 28.6 [Flag Variables Ordering], page 145). With this you can define compilation flags that apply to all files compiled for a target. For instance, in

```
bin_PROGRAMS = foo
too_SOURCES = foo.c foo.h bar.c bar.h main.c
too_CFLAGS = -some -flags
```

`foo-foo.o`, `foo-bar.o`, and `foo-main.o` will all be compiled with `-some -flags`. (If you wonder about the names of these object files, see Section 28.7 [Renamed Objects], page 147.) Note that `foo_CFLAGS` gives the flags to use when compiling all the C sources of the program `foo`, it has nothing to do with `foo.c` or `foo-foo.o` specifically.

What if `foo.c` needs to be compiled into `foo.o` using some specific flags, that none of the other files requires? Obviously per-program flags are not directly applicable here. Something like per-object flags are expected, i.e., flags that would be used only when creating `foo-foo.o`. Automake does not support that, however this is easy to simulate using a library that contains only that object, and compiling this library with per-library flags.

```
bin_PROGRAMS = foo
too_SOURCES = bar.c bar.h main.c
too_CFLAGS = -some -flags
too_LDADD = libfoo.a
noinst_LIBRARIES = libfoo.a
libfoo_a_SOURCES = foo.c foo.h
libfoo_a_CFLAGS = -some -other -flags
```
Here `foo-bar.o` and `foo-main.o` will all be compiled with `'-some -flags'`, while `libfoo_a-foo.o` will be compiled using `'-some-other -flags'`. Eventually, all three objects will be linked to form `foo`.

This trick can also be achieved using Libtool convenience libraries, for instance `noinst_LTLIBRARIES = libfoo.la` (see Section 8.3.5 [Libtool Convenience Libraries], page 61).

Another tempting idea to implement per-object flags is to override the compile rules `automake` would output for these files. Automake will not define a rule for a target you have defined, so you could think about defining the `foo-foo.o: foo.c` rule yourself. We recommend against this, because this is error prone. For instance, if you add such a rule to the first example, it will break the day you decide to remove `foo_CFLAGS` (because `foo.c` will then be compiled as `foo.o` instead of `foo-foo.o`, see Section 28.7 [Renamed Objects], page 147). Also in order to support dependency tracking, the two `.o/.obj` extensions, and all the other flags variables involved in a compilation, you will end up modifying a copy of the rule previously output by `automake` for this file. If a new release of Automake generates a different rule, your copy will need to be updated by hand.

### 28.9 Handling Tools that Produce Many Outputs

This section describes a `make` idiom that can be used when a tool produces multiple output files. It is not specific to Automake and can be used in ordinary Makefiles.

Suppose we have a program called `foo` that will read one file called `data.foo` and produce two files named `data.c` and `data.h`. We want to write a `Makefile` rule that captures this one-to-two dependency.

The naive rule is incorrect:

```make
# This is incorrect.
data.c data.h: data.foo
  foo data.foo
```

What the above rule really says is that `data.c` and `data.h` each depend on `data.foo`, and can each be built by running `foo data.foo`. In other words it is equivalent to:

```make
# We do not want this.
data.c: data.foo
  foo data.foo
data.h: data.foo
  foo data.foo
```

which means that `foo` can be run twice. Usually it will not be run twice, because `make` implementations are smart enough to check for the existence of the second file after the first one has been built; they will therefore detect that it already exists. However there are a few situations where it can run twice anyway:

- The most worrying case is when running a parallel `make`. If `data.c` and `data.h` are built in parallel, two `foo data.foo` commands will run concurrently. This is harmful.
- Another case is when the dependency (here `data.foo`) is (or depends upon) a phony target.

A solution that works with parallel `make` but not with phony dependencies is the following:
data.c data.h: data.foo
  foo data.foo
data.h: data.c

The above rules are equivalent to

data.c: data.foo
  foo data.foo
data.h: data.foo data.c
  foo data.foo

therefore a parallel `make` will have to serialize the builds of `data.c` and `data.h`, and will detect that the second is no longer needed once the first is over.

Using this pattern is probably enough for most cases. However it does not scale easily to more output files (in this scheme all output files must be totally ordered by the dependency relation), so we will explore a more complicated solution.

Another idea is to write the following:

```
# There is still a problem with this one.
data.c: data.foo
  foo data.foo
data.h: data.c

## Recover from the removal of $@
@if test -f $@; then :; else 
  rm -f data.c; 
  $(MAKE) $(AM_MAKEFLAGS) data.c; 
fi
```

The idea is that `foo data.foo` is run only when `data.c` needs to be updated, but we further state that `data.h` depends upon `data.c`. That way, if `data.h` is required and `data.foo` is out of date, the dependency on `data.c` will trigger the build.

This is almost perfect, but suppose we have built `data.h` and `data.c`, and then we erase `data.h`. Then, running `make data.h` will not rebuild `data.h`. The above rules just state that `data.c` must be up-to-date with respect to `data.foo`, and this is already the case.

What we need is a rule that forces a rebuild when `data.h` is missing. Here it is:

```
data.c: data.foo
  foo data.foo
data.h: data.c
## Recover from the removal of $@
@if test -f $@; then :; else 
  rm -f data.c; 
  $(MAKE) $(AM_MAKEFLAGS) data.c; 
fi
```

The above scheme can be extended to handle more outputs and more inputs. One of the outputs is selected to serve as a witness to the successful completion of the command, it depends upon all inputs, and all other outputs depend upon it. For instance, if `foo` should additionally read `data.bar` and also produce `data.w` and `data.x`, we would write:

```
data.c: data.foo data.bar
  foo data.foo data.bar
data.h data.w data.x: data.c
## Recover from the removal of $@
@if test -f $@; then :; else 
  rm -f data.c; 
  $(MAKE) $(AM_MAKEFLAGS) data.c; 
```

However there are now three minor problems in this setup. One is related to the time-stamp ordering of data.h, data.w, data.x, and data.c. Another one is a race condition if a parallel make attempts to run multiple instances of the recover block at once. Finally, the recursive rule breaks `make -n` when run with GNU make (as well as some other make implementations), as it may remove data.h even when it should not (see Section “How the MAKE Variable Works” in The GNU Make Manual).

Let us deal with the first problem. foo outputs four files, but we do not know in which order these files are created. Suppose that data.h is created before data.c. Then we have a weird situation. The next time make is run, data.h will appear older than data.c, the second rule will be triggered, a shell will be started to execute the `if...fi` command, but actually it will just execute the then branch, that is: nothing. In other words, because the witness we selected is not the first file created by foo, make will start a shell to do nothing each time it is run.

A simple riposte is to fix the timestamps when this happens.

```bash
data.c: data.foo data.bar
data.h data.w data.x: data.c
  @if test -f $@; then \
    touch $@; \
  else \n
  ## Recover from the removal of $@
  rm -f data.c; \
  $(MAKE) $(AM_MAKEFLAGS) data.c; \
fi
```

Another solution is to use a different and dedicated file as witness, rather than using any of foo's outputs.

```bash
data.stamp: data.foo data.bar
  @rm -f data.tmp
  @touch data.tmp
  foo data.foo data.bar
  @mv -f data.tmp $@
data.c data.h data.w data.x: data.stamp
  ## Recover from the removal of $@
  @if test -f $@; then ;; else \
    rm -f data.stamp; \
    $(MAKE) $(AM_MAKEFLAGS) data.stamp; \
fi
```

data.tmp is created before foo is run, so it has a timestamp older than output files output by foo. It is then renamed to data.stamp after foo has run, because we do not want to update data.stamp if foo fails.

This solution still suffers from the second problem: the race condition in the recover rule. If, after a successful build, a user erases data.c and data.h, and runs `make -j`, then make may start both recover rules in parallel. If the two instances of the rule execute `$\text{(MAKE)}$`
$(AM_MAKEFLAGS) data.stamp' concurrently the build is likely to fail (for instance, the two
rules will create data.tmp, but only one can rename it).

Admittedly, such a weird situation does not arise during ordinary builds. It occurs only
when the build tree is mutilated. Here data.c and data.h have been explicitly removed
without also removing data.stamp and the other output files. make clean; make will always
recover from these situations even with parallel makes, so you may decide that the recover
rule is solely to help non-parallel make users and leave things as-is. Fixing this requires some
locking mechanism to ensure only one instance of the recover rule rebuilds data.stamp. One
could imagine something along the following lines.

```
data.c data.h data.w data.x: data.stamp
## Recover from the removal of $@
  @if test -f $@; then ;; else \
    trap 'rm -rf data.lock data.stamp' 1 2 13 15; \
  ## mkdir is a portable test-and-set
  if mkdir data.lock 2>/dev/null; then \
  ## This code is being executed by the first process.
    rm -f data.stamp; \
    $(MAKE) $(AM_MAKEFLAGS) data.stamp; \n    result=$$?; rm -rf data.lock; exit $$result; \n  else \
  ## This code is being executed by the follower processes.
  ## Wait until the first process is done.
    while test -d data.lock; do sleep 1; done; \
  ## Succeed if and only if the first process succeeded.
    test -f data.stamp; \n    fi; \
  fi
```

Using a dedicated witness, like data.stamp, is very handy when the list of output files
is not known beforehand. As an illustration, consider the following rules to compile many
*.el files into *.elc files in a single command. It does not matter how ELFILES is defined
(as long as it is not empty: empty targets are not accepted by POSIX).

```
ELFILES = one.el two.el three.el ...
ELCFILES = $(ELFILES:=c)
elc-stamp: $(ELFILES)
  @rm -f elc-temp
  @touch elc-temp
  $(elisp_comp) $(ELFILES)
  @mv -f elc-temp $@

$(ELCFILES): elc-stamp
  @if test -f $@; then ;; else \
  ## Recover from the removal of $@
    trap 'rm -rf elc-lock elc-stamp' 1 2 13 15; \n    if mkdir elc-lock 2>/dev/null; then \
    ## This code is being executed by the first process.
These solutions all still suffer from the third problem, namely that they break the promise that `make -n` should not cause any actual changes to the tree. For those solutions that do not create lock files, it is possible to split the recover rules into two separate recipe commands, one of which does all work but the recursion, and the other invokes the recursive `$MAKEFLAGS`. The solutions involving locking could act upon the contents of the `MAKEFLAGS` variable, but parsing that portably is not easy (see Section "The Make Macro MAKEFLAGS" in The Autoconf Manual). Here is an example:

```
ELFILES = one.el two.el three.el ...
ELCFILES = $(ELFILES:=c)

elc-stamp: $(ELFILES)
  @rm -f elc-temp
  @touch elc-temp
  $(elisp_comp) $(ELFILES)
  @mv -f elc-temp $@

$(ELCFILES): elc-stamp
  ## Recover from the removal of $@
  @dry=; for f in x $MAKEFLAGS; do 
    case $$f in 
      *=*|--*);; 
    *n*) dry=::; 
      esac; 
      done; 
  if test -f $@; then ;; else 
    $$dry trap 'rm -rf elc.lock elc-stamp' 1 2 13 15; 
    if $$dry mkdir elc.lock 2>/dev/null; then 
      $$dry rm elc-stamp; 
      $(MAKE) $(AM_MAKEFLAGS) elc-stamp; 
      $$dry rmdir elc-lock; 
    else 
      ## This code is being executed by the follower processes.
      ## Wait until the first process is done.
      while test -d elc-lock && test -z "$dry"; do 
    ```
## Succeed if and only if the first process succeeded.

```
sleep 1; \
  done; \
  ## Succeed if and only if the first process succeeded.
  $$\text{dry test -f elc-stamp; exit $$?; \\
  fi; \\
  fi }
```

For completeness it should be noted that GNU `make` is able to express rules with multiple output files using pattern rules (see Section “Pattern Rule Examples” in *The GNU Make Manual*). We do not discuss pattern rules here because they are not portable, but they can be convenient in packages that assume GNU `make`.

### 28.10 Installing to Hard-Coded Locations

My package needs to install some configuration file. I tried to use the following rule, but `make distcheck` fails. Why?

```
# Do not do this.
install-data-local:
  $(INSTALL_DATA) $(srcdir)/afile $(DESTDIR)/etc/afile
```

My package needs to populate the installation directory of another package at install-time. I can easily compute that installation directory in `configure`, but if I install files therein, `make distcheck` fails. How else should I do?

These two setups share their symptoms: `make distcheck` fails because they are installing files to hard-coded paths. In the later case the path is not really hard-coded in the package, but we can consider it to be hard-coded in the system (or in whichever tool that supplies the path). As long as the path does not use any of the standard directory variables (`$(prefix)`, `$(bindir)`, `$(datadir)`, etc.), the effect will be the same: user-installations are impossible.

As a (non-root) user who wants to install a package, you usually have no right to install anything in `/usr` or `/usr/local`. So you do something like `./configure --prefix ~/usr` to install a package in your own `~/usr` tree.

If a package attempts to install something to some hard-coded path (e.g., `/etc/afile`), regardless of this `--prefix` setting, then the installation will fail. `make distcheck` performs such a `--prefix` installation, hence it will fail too.

Now, there are some easy solutions.

The above `install-data-local` example for installing `/etc/afile` would be better replaced by

```
sysconf_DATA = afile
```

by default `sysconfdir` will be `$(prefix)/etc`, because this is what the GNU Standards require. When such a package is installed on an FHS compliant system, the installer will have to set `--sysconfdir=/etc`. As the maintainer of the package you should not be concerned by such site policies: use the appropriate standard directory variable to install your files so that the installer can easily redefine these variables to match their site conventions.
Installing files that should be used by another package is slightly more involved. Let’s take an example and assume you want to install a shared library that is a Python extension module. If you ask Python where to install the library, it will answer something like this:

```bash
% python -c 'from distutils import sysconfig;
print sysconfig.get_python_lib(1,0)'
/usr/lib/python2.5/site-packages
```

If you indeed use this absolute path to install your shared library, non-root users will not be able to install the package, hence distcheck fails.

Let’s do better. The `sysconfig.get_python_lib()` function actually accepts a third argument that will replace Python’s installation prefix.

```bash
% python -c 'from distutils import sysconfig;
print sysconfig.get_python_lib(1,0,"${exec_prefix}")

${exec_prefix}/lib/python2.5/site-packages
```

You can also use this new path. If you do

- root users can install your package with the same `--prefix` as Python (you get the behavior of the previous attempt)
- non-root users can install your package too, they will have the extension module in a place that is not searched by Python but they can work around this using environment variables (and if you installed scripts that use this shared library, it’s easy to tell Python were to look in the beginning of your script, so the script works in both cases).

The `AM_PATH_PYTHON` macro uses similar commands to define `$(pythondir)` and `$(pyexecdir)` (see Section 10.5 [Python], page 89).

Of course not all tools are as advanced as Python regarding that substitution of prefix. So another strategy is to figure the part of the installation directory that must be preserved. For instance, here is how `AM_PATH_LISPDIR` (see Section 10.1 [Emacs Lisp], page 87) computes `$(lispdir)`:

```bash
$EMACS -batch -q -eval '(while load-path
 (princ (concat (car load-path) "\n"))
 (setq load-path (cdr load-path)))' >conftest.out
lispdir='sed -n
 -e 's,,'
 -e '/.*\(/x*emacs/site-lisp$/{ s,\(/x*emacs/site-lisp\)$,${libdir}/\1,"p;q;
 }'
 -e '/.*\(/share/x*emacs/site-lisp$/{ s,\(/share/x*emacs/site-lisp\)$,${datarootdir}/\1,"p;q;
 }'
 conftest.out'
```

I.e., it just picks the first directory that looks like `*/lib/*emacs/site-lisp` or `*/share/*emacs/site-lisp` in the search path of emacs, and then substitutes `"$\{libdir\}` or `$\{datadir\}` appropriately.

The emacs case looks complicated because it processes a list and expects two possible layouts, otherwise it’s easy, and the benefits for non-root users are really worth the extra `sed` invocation.
28.11 Debugging Make Rules

The rules and dependency trees generated by `automake` can get rather complex, and leave the developer head-scratching when things don’t work as expected. Besides the debug options provided by the `make` command (see Section “Options Summary” in *The GNU Make Manual*), here’s a couple of further hints for debugging makefiles generated by `automake` effectively:

- If less verbose output has been enabled in the package with the ‘silent-rules’ option (see Chapter 17 [Options], page 117), you can use `make V=1` to see the commands being executed.

- `make -n` can help show what would be done without actually doing it. Note however, that this will *still execute* commands prefixed with ‘+’, and, when using GNU `make`, commands that contain the strings `$(MAKE)` or `${MAKE}` (see Section “Instead of Execution” in *The GNU Make Manual*). Typically, this is helpful to show what recursive rules would do, but it means that, in your own rules, you should not mix such recursion with actions that change any files. Furthermore, note that GNU `make` will update prerequisites for the `Makefile` file itself even with `-n` (see Section “Remaking Makefiles” in *The GNU Make Manual*).


- `echo 'print: ; @echo "$(VAR)"' | make -f Makefile -f - print` can be handy to examine the expanded value of variables. You may need to use a target other than ‘print’ if that is already used or a file with that name exists.

- [http://bashdb.sourceforge.net/remake/](http://bashdb.sourceforge.net/remake/) provides a modified GNU `make` command called `remake` that copes with complex GNU `make`-specific Makefiles and allows to trace execution, examine variables, and call rules interactively, much like a debugger.

28.12 Reporting Bugs

Most nontrivial software has bugs. Automake is no exception. Although we cannot promise we can or will fix a bug, and we might not even agree that it is a bug, we want to hear about problems you encounter. Often we agree they are bugs and want to fix them.

To make it possible for us to fix a bug, please report it. In order to do so effectively, it helps to know when and how to do it.

Before reporting a bug, it is a good idea to see if it is already known. You can look at the GNU Bug Tracker ([http://debbugs.gnu.org/](http://debbugs.gnu.org/)) and the bug-automake mailing list archives ([http://lists.gnu.org/archive/html/bug-automake/](http://lists.gnu.org/archive/html/bug-automake/)) for previous bug reports. We previously used a Gnats database ([http://sourceware.org/cgi-bin/gnatsweb.pl?database=automake](http://sourceware.org/cgi-bin/gnatsweb.pl?database=automake)) for bug tracking, so some bugs might have been reported there already. Please do not use it for new bug reports, however.

If the bug is not already known, it should be reported. It is very important to report bugs in a way that is useful and efficient. For this, please familiarize yourself with

---

8 Automake’s ‘dist’ and ‘distcheck’ rules had a bug in this regard in that they created directories even with `-n`, but this has been fixed in Automake 1.11.
How to Report Bugs Effectively (http://www.chiark.greenend.org.uk/~sgtatham/bugs.html) and How to Ask Questions the Smart Way (http://catb.org/~esr/faqs/smart-questions.html). This helps you and developers to save time which can then be spent on fixing more bugs and implementing more features.

For a bug report, a feature request or other suggestions, please send email to bug-automake@gnu.org. This will then open a new bug in the bug tracker (http://debbugs.gnu.org/automake). Be sure to include the versions of Autoconf and Automake that you use. Ideally, post a minimal Makefile.am and configure.ac that reproduces the problem you encounter. If you have encountered test suite failures, please attach the test-suite.log file.
Appendix A Copying This Manual

A.1 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000-2012 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.
2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all of these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:
Appendix A: Copying This Manual

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at
your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.
7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.
10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free
Documentation License from time to time. Such new versions will be similar in spirit
to the present version, but may differ in detail to address new problems or concerns.
See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document
specifies that a particular numbered version of this License “or any later version”
applies to it, you have the option of following the terms and conditions either of that
specified version or of any later version that has been published (not as a draft) by
the Free Software Foundation. If the Document does not specify a version number of
this License, you may choose any version ever published (not as a draft) by the Free
Software Foundation. If the Document specifies that a proxy can decide which future
versions of this License can be used, that proxy’s public statement of acceptance of a
version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide
Web server that publishes copyrightable works and also provides prominent facilities
for anybody to edit those works. A public wiki that anybody can edit is an example of
such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the
site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license pub-
lished by Creative Commons Corporation, a not-for-profit corporation with a principal
place of business in San Francisco, California, as well as future copyleft versions of that
license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part
of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works
that were first published under this License somewhere other than this MMC, and
subsequently incorporated in whole or in part into the MMC, (1) had no cover texts
or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under
CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is
eligible for relicensing.
ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) year your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the
“with...Texts.” line with this:

with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.

If you have Invariant Sections without Cover Texts, or some other combination of the
three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing
these examples in parallel under your choice of free software license, such as the GNU
General Public License, to permit their use in free software.
Appendix B Indices

B.1 Macro Index

A

AC_CANONICAL_BUILD ..................................................... 32
AC_CANONICAL_HOST ..................................................... 32
AC_CONFIG_AUX_DIR ...................................................... 32, 52
AC_CONFIG_FILES ......................................................... 30
AC_CONFIG_HEADERS ...................................................... 32
AC_CONFIG_LIBOBJ_DIR ................................................... 32, 70
AC_CONFIG_LINKS ........................................................... 32
AC_CONFIG_SUBDIRS ....................................................... 52
AC_DEFUN ................................................................. 40
AC_F77_LIBRARY_LDFLAGS .................................................. 33
AC_FC_SRCEXT .............................................................. 33
AC_INIT ................................................................. 33
AC_LIBOBJ ............................................................... 32, 63, 69
AC_LIBSOURCE ............................................................. 32, 69
AC_LIBSOURCES ............................................................ 32
AC_OUTPUT ............................................................... 30
AC_PREREQ .............................................................. 40
AC_PROG_CC_C_O .......................................................... 45
AC_PROG_CXX ............................................................. 33
AC_PROG_FC ............................................................... 33
AC_PROG_LEX .............................................................. 34, 46
AC_PROG_LIBTOOL ........................................................ 33
AC_PROG_OBJC ............................................................ 33
AC_PROG_OBJCXX ........................................................ 33
AC_PROG_RANLIB .......................................................... 33
AC_PROG_YACC ............................................................. 33
AC_REQUIRE_AUX_FILE .................................................... 34

AM_DEPENDENCIES ..................................................... 47
AM_MAKE ............................................................... 33
AM_MISSING_PROG ....................................................... 46
AM_OUTPUT_DEPENDENCY_COMMANDS .................................... 47
AM_PATH ............................................................... 33
AM_PHASE .............................................................. 33
AM_PYTHON .............................................................. 33
AM_SET ............................................................... 33
AM_SUBST .............................................................. 33
AM_VAR ............................................................... 33

B.2 Variable Index

_DATA ................................................................. 83
_HEADERS .............................................................. 83
_LIBRARIES ............................................................ 57
_LISP ................................................................. 87
_LOG_COMPILE .......................................................... 106
_LOG_COMPILER ........................................................ 106
_LOG_DRIVER .......................................................... 109
_LOG_DRIVER_FLAGS .................................................... 109
_LOG_FLAGS ............................................................ 106
_LTLIBRARIES .......................................................... 59
_MANS ............................................................... 93
_PRGRMS ............................................................. 20, 54
_PYTHON .............................................................. 89
_SCRIPTS .............................................................. 82
_SOURCES ............................................................ 54, 68
_TEXINFUS ............................................................ 91, 92
### Appendix B: Indices

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJC</td>
<td>TAGS_DEPENDENCIES</td>
</tr>
<tr>
<td>OBJCCOMPILE</td>
<td>target_triplet</td>
</tr>
<tr>
<td>OBJCFLAGS</td>
<td>TEST_EXTENSIONS</td>
</tr>
<tr>
<td>OBJCLINK</td>
<td>TEST_LOGS</td>
</tr>
<tr>
<td>OBJCXX</td>
<td>TEST_SUITE_LOG</td>
</tr>
<tr>
<td>OBJCXXCOMPILE</td>
<td>TESTS</td>
</tr>
<tr>
<td>OBJCXXFLAGS</td>
<td>TESTS_ENVIRONMENT</td>
</tr>
<tr>
<td>OBJCXXLINK</td>
<td>TEXI2DVI</td>
</tr>
<tr>
<td>oldinclude_HEADERS</td>
<td>TEXINFO_TEX</td>
</tr>
<tr>
<td></td>
<td>TEXINFOS</td>
</tr>
<tr>
<td></td>
<td>top_distdir</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACKAGE</td>
<td>UPC</td>
</tr>
<tr>
<td>pkgdata_DATA</td>
<td>UPC_COMPILE</td>
</tr>
<tr>
<td>pkgdata_SCRIPTS</td>
<td>UPCFLAGS</td>
</tr>
<tr>
<td>pkgdatadir</td>
<td>UPCLINK</td>
</tr>
<tr>
<td>pkginclude_HEADERS</td>
<td></td>
</tr>
<tr>
<td>pkginclude</td>
<td></td>
</tr>
<tr>
<td>pkglib_LIBRARIES</td>
<td></td>
</tr>
<tr>
<td>pkglib_LTLIBRARIES</td>
<td></td>
</tr>
<tr>
<td>pkglibdir</td>
<td></td>
</tr>
<tr>
<td>pkglibexec_PROGRAMS</td>
<td></td>
</tr>
<tr>
<td>pkglibexec_SCRIPTS</td>
<td></td>
</tr>
<tr>
<td>pkglibexecdir</td>
<td></td>
</tr>
<tr>
<td>pkgpythondir</td>
<td></td>
</tr>
<tr>
<td>PROGRAMS</td>
<td></td>
</tr>
<tr>
<td>pyexecdir</td>
<td></td>
</tr>
<tr>
<td>PYTHON</td>
<td></td>
</tr>
<tr>
<td>PYTHON_EXEC_PREFIX</td>
<td></td>
</tr>
<tr>
<td>PYTHON_PLATFORM</td>
<td></td>
</tr>
<tr>
<td>PYTHON_PREFIX</td>
<td></td>
</tr>
<tr>
<td>PYTHON_VERSION</td>
<td></td>
</tr>
<tr>
<td>pythondir</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECHECK_LOGS</td>
<td>V...</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>V...</td>
</tr>
<tr>
<td>RUNTEST</td>
<td>VALAC</td>
</tr>
<tr>
<td>RUNTESTDEFAULTFLAGS</td>
<td>VALAFLAGS</td>
</tr>
<tr>
<td>RUNTESTFLAGS</td>
<td>VERBOSE</td>
</tr>
<tr>
<td></td>
<td>VERSION</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbin_PROGRAMS</td>
<td>WARNINGS</td>
</tr>
<tr>
<td>sbin_SCRIPTS</td>
<td>WITH_DMALLOC</td>
</tr>
<tr>
<td>SCRIPTS</td>
<td></td>
</tr>
<tr>
<td>sharedstate_DATA</td>
<td></td>
</tr>
<tr>
<td>SOURCES</td>
<td></td>
</tr>
<tr>
<td>SUBDIRS</td>
<td></td>
</tr>
<tr>
<td>SUFFIXES</td>
<td></td>
</tr>
<tr>
<td>sysconf_DATA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFAIL_TESTS</td>
<td>YACC</td>
</tr>
<tr>
<td>XZ_OPT</td>
<td>YFLAGS</td>
</tr>
</tbody>
</table>
B.3  General Index
Appendix B: Indices

#
## (special Automake comment) .................................. 19
#serial syntax ................................................. 42

$
\$(LIBOBS)’ and empty libraries ............................... 70

+$
+= ............................................................. 18

--add-missing ................................................. 27
--automake-acdir ............................................. 36
--build=build ................................................ 9
--copy ........................................................ 27
--cygnus ...................................................... 27
--diff ........................................................ 36
--disable-dependency-tracking ......................... 12
--disable-maintainer-mode ............................... 34
--disable-silent-rules .................................... 129
--dry-run ..................................................... 36
--enable-debug, example .................................. 124
--enable-dependency-tracking ......................... 12
--enable-maintainer-mode ............................... 34
--enable-silent-rules .................................... 129
--force ......................................................... 36
--force-missing ............................................. 28
--foreign ....................................................... 28
--gnits ........................................................ 28
--gnits, complete description ......................... 130
--gnu ........................................................... 28
--gnu, complete description ............................ 130
--gnu, required files ...................................... 130
--help ......................................................... 28, 36
--help check .................................................. 120
--help=recursive ........................................... 12
--host=host .................................................... 9
--include-deps ............................................... 28
--install ......................................................... 36
--libdir ......................................................... 27
--no-force ....................................................... 28
--output ........................................................ 36
--output-dir ................................................... 28
--prefix ........................................................ 5
--print-ac-dir ............................................... 36
--print-libdir ............................................... 27
--program-prefix=prefix .................................... 10
--program-suffix=_suffix ................................ 10
--program-transform-name=program ..................... 10
--system-acdir .............................................. 36
--target=target ................................................ 9
--verbose ...................................................... 28, 37
--version ...................................................... 28, 37
--version check ............................................... 120
--warnings ..................................................... 28, 37

--with-dmalloc ............................................... 46
-a ............................................................ 27
-c ............................................................ 27
-f ............................................................ 28
-hook targets .................................................. 133
-i ............................................................ 28
-I .............................................................. 36
-l and LDADD ................................................... 55
-local targets ................................................ 132
-module, libtool ............................................. 62
-o .............................................................. 28
-v .............................................................. 28
-W ............................................................... 37
-Wall ........................................................... 16
-Werror ........................................................ 16

.

.1a suffix, defined .............................................. 58
.log files ....................................................... 105
.trs files ....................................................... 105

:

copy-in-global-log: ............................................ 111
recheck: ......................................................... 111
test-global-result: ............................................ 111
test-result: ....................................................... 110

-

_DATA primary, defined ........................................ 83
_DEPENDENCIES, defined ....................................... 55
_HEADERS primary, defined .................................. 83
_JAVA primary, defined ...................................... 88
_LDFLAGS, defined ............................................. 55
_LDFLAGS, libtool ............................................ 63
_LIBADD, libtool .............................................. 63
_LIBADD, libtool .............................................. 63
_LIBRARIES primary, defined ................................ 57
_LIBTOOLFLAGS, libtool ....................................... 63
_LISP primary, defined ...................................... 87
_LTLSOURCES primary, defined ................................ 59
_MANS primary, defined ....................................... 93
_PROGRAMS primary variable ................................... 20
_PYTHON primary, defined .................................... 89
_SCRIPTS primary, defined ..................................... 82
_SOURCES and header files ................................... 54
_SOURCES primary, defined ................................... 54
_SOURCES, default ............................................. 68
_SOURCES, empty ............................................... 69
_TEXINFOS primary, defined ................................... 91
CVS and third-party files .................................. 140
CVS and timestamps ........................................ 138
CXXFLAGS and AM_CXXFLAGS .............................. 145
cygnum ..................................................... 118
cygnum strictness ......................................... 131

D
DATA primary, defined ..................................... 83
debug build, example ...................................... 7
debugging rules ............................................ 156
default _SOURCES .......................................... 68
default source, Libtool modules example .............. 69
default verbosity for silent-rules ...................... 129
definitions, conflicts ................................... 132
dejaegnu ................................................... 116, 118
depcomp ..................................................... 81
dependencies and distributed files .................... 143
Dependency tracking ....................................... 11, 80
Dependency tracking, disabling ......................... 81
directory variables ....................................... 4
dirlist ....................................................... 38
Disabling dependency tracking ......................... 81
Disabling hard errors .................................... 103
dist ......................................................... 4, 97
dist-bzip2 .................................................. 101, 118
dist-gzip .................................................... 101
dist-hook ................................................... 98, 133
dist-lzzip .................................................... 101, 119
dist-shar ..................................................... 101, 119
dist-tar ....................................................... 101, 119
dist-xz ....................................................... 101, 119
dist-zip ....................................................... 101, 119
dist_and_nobase .......................................... 51
dist_and_notrans ......................................... 94
DIST_SUBDIRS, explained .............................. 49
distcheck .................................................... 15, 99
distcheck better than dist .............................. 11
distcheck example ....................................... 15
distcheck-hook ............................................ 99
distclean .................................................... 4, 132, 143
distclean, diagnostic .................................... 143
distclean-local .......................................... 96, 132
distcleancheck ........................................... 99
distdir ....................................................... 133
Distinction between errors and failures in testsuites ................................ 102
Distributions, preparation ............................. 11
distinstallcheck .......................................... 100
dmalloc, support for .................................... 46
dvi ........................................................... 91, 132
dvi-local ..................................................... 132
DVI output using Texinfo ................................. 91

E
E-mail, bug reports ........................................ 1
EDITION Texinfo flag ...................................... 91
e else ......................................................... 124
empty _SOURCES ........................................... 69
Empty libraries ............................................ 58
Empty libraries and ‘$(LIBOBS)’ ......................... 70
endif ....................................................... 124
Example conditional --enable-debug .................... 124
Example conditional AC_CONFIG_FILES ................. 125
Example Hello World ..................................... 13
Example of recursive operation ....................... 19
Example of shared libraries ............................ 59
Example, EXTRA_PROGRAMS ............................ 20
Example, false and true ................................ 25
Example, mixed language ............................... 78
Executable extension ..................................... 81
Exit status 77, special interpretation ............... 102
Exit status 99, special interpretation ............... 102
expected failure ........................................ 102
Expected test failure .................................... 103
expected test failure .................................... 102
Extending aclocal ........................................ 39
Extending list of installation directories .......... 21
Extension, executable .................................. 81
Extra files distributed with Automake ............... 27
EXTRA_, prepending .................................... 20
EXTRA_prog_SOURCES, defined ....................... 56
EXTRA_PROGRAMS, defined ............................ 20, 57

F
false Example ............................................. 25
FCFLAGS and AM_FCFLAGS ............................... 145
Features of the GNU Build System .................... 2
FFLAGS and AM_FFLAGS .................................. 145
file names, limitations on ............................... 142
filename-length-max=99 .................................. 119
Files distributed with Automake ....................... 27
First line of Makefile.am ............................... 19
Flag variables, ordering ................................ 145
Flag Variables, Ordering ............................... 145
FLIBS, defined .......................................... 77
foreign ..................................................... 16, 118
foreign strictness ....................................... 19
Fortran 77 support ...................................... 76
Fortran 77, mixing with C and C++ ................... 77
Fortran 77, Preprocessing ............................. 77
Fortran 9x support ....................................... 79
source tree, read-only .................................. 8
SOURCES primary, defined ............................ 54
Special Automake comment ............................ 19
Staged installation .................................... 10
std-options ........................................... 120
Strictness, command line .............................. 27
Strictness, defined ................................... 19
Strictness, foreign .................................... 19
Strictness, gnits ....................................... 19
Strictness,gnu ......................................... 19
su, before make install .............................. 3
subdir-objects ....................................... 121
Subdirectories, building conditionally ............ 48
Subdirectories, configured conditionally .......... 50
Subdirectories, not distributed ..................... 51
SUBDIRS and AC_SUBST .............................. 65
SUBDIRS and AM_CONDITIONAL ....................... 49
SUBDIRS, conditional ................................ 48
SUBDIRS, explained .................................. 48
Subpackages .......................................... 12, 52
suffix .la, defined .................................... 58
suffix .lo, defined .................................... 58
SUFFIXES, adding ................................... 123
Support for C++ ...................................... 74
Support for Fortran 77 ................................. 76
Support for Fortran 9x ................................ 79
Support for GNU Gettext ............................. 88
Support for Java with gcj ............................. 79
Support for Objective C .................................. 75
Support for Objective C++ ............................ 75
Support for Unified Parallel C .......................... 75
Support for Vala ....................................... 80

T
tags .................................................... 122
TAGS support ......................................... 122
tar formats ............................................ 121
tar-pax .................................................. 121
tar-ustar .............................................. 121
tar-v7 ................................................... 121
Target, install-info ................................... 92
Target, install-man .................................... 94
termios POSIX headers ................................ 47
test case .............................................. 101
Test case result, registering ......................... 110
test failure ............................................. 102
test harness ........................................... 101
test metadata .......................................... 105
test pass ................................................. 102
Test result, registering ................................ 110
test skip .................................................. 102
Test suites ............................................. 101
Tests, expected failure ................................ 103
testsuite harness ...................................... 101
Testsuite progress on console ......................... 103
Texinfo flag, EDITION ................................ 91
Texinfo flag, UPDATED ............................... 91
Texinfo flag, UPDATED-MONTH .......................... 91
Texinfo flag, VERSION ................................ 91
texinfo.tex ............................................ 92
TEXINFOS primary, defined .......................... 91
third-party files and CVS ............................ 140
Third-party packages, interfacing with .......... 133
timestamps and CVS .................................. 138
Transforming program names .......................... 10
trees, source vs. build ............................... 6
t true Example ....................................... 25

U
underquoted AC_DEFUN ................................ 40
unexpected pass ....................................... 102
unexpected test pass .................................. 102
Unified Parallel C support ........................... 75
Uniform naming scheme ................................ 20
uninstall .............................................. 4, 96, 132
uninstall-hook ........................................ 133
uninstall-local ......................................... 132
Unit tests .............................................. 107
Unpacking ............................................. 3
UPFLAGS and AM_UPCFLAGS ........................... 145
UPDATED Texinfo flag ................................ 91
UPDATED-MONTH Texinfo flag ....................... 91
Use Cases for the GNU Build System ............... 2
user variables ......................................... 23
Using aclocal ......................................... 29
ustar format ............................................ 121

V
v7 tar format .......................................... 121
Vala Support .......................................... 80
variables, conflicting .................................. 132
Variables, overriding .................................. 19
variables, reserved for the user .................... 23
version.m4, example .................................. 116
version.sh, example ................................... 116
VERSION Texinfo flag ................................ 91
VERSION, prevent definition ....................... 45
versioned binaries, installing ...................... 133
VPATH builds .......................................... 6

W
wildcards .............................................. 141
Windows ................................................. 81

X
xfail .................................................. 102
xpass .................................................. 102
<table>
<thead>
<tr>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>yacc, multiple parsers</td>
<td>zardoz example</td>
</tr>
<tr>
<td>YFLAGS and AM_YFLAGS</td>
<td></td>
</tr>
<tr>
<td>ylwrap</td>
<td></td>
</tr>
</tbody>
</table>

73                             25