GNU Automake

For version 1.9.106, 18 September 2005

David MacKenzie
Tom Tromey
Alexandre Duret-Lutz

This manual is for GNU Automake (version 1.9.106, 18 September 2005), a program that creates GNU standards-compliant Makefiles from template files.

1 Introduction

Automake is a tool for automatically generating '`Makefile.in`'s 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`'s 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 maintainer).

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[2].

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

Mail suggestions and bug reports for Automake to `bug-automake@gnu.org`.

2 General ideas

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

2.1 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 verbatim into the generated file. This allows you to add arbitrary code into the generated '`Makefile.in`'. For instance, the Automake distribution includes a non-standard rule for the `cvs-dist` target, which the Automake maintainer uses to make distributions from his 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.

[1] These variables are also called *make macros* in Make terminology, however in this manual we reserve the term *macro* for Autoconf's macros.

[2] Older Autoconf versions used '`configure.in`'. Autoconf 2.50 and greater promotes '`configure.ac`' over '`configure.in`'. The rest of this documentation will refer to '`configure.ac`', but Automake also supports '`configure.in`' for backward compatibility.

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.

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_SUBST`ed 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

```
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:

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

2.2 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. 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 20 [Gnits], page 82, for more information on the precise implications of the strictness level.

Automake also has a special "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., the GCC tree). See Chapter 21 [Cygnus], page 83, for more information on this mode.

2.3 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 `pkglibdir`, `pkgincludedir`, and `pkgdatadir`; 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`:

```
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. 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'.

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

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 7.2 [A Library], page 34), 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', '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_', 'nodist_', and 'nobase_'. These prefixes are explained later (see Section 7.4 [Program and Library Variables], page 40).

2.4 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.

2.5 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 26.5 [Flag Variables Ordering], page 96, for more discussion about these variables and how they interact with per-target variables.

2.6 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.

`ansi2knr.c`
`ansi2knr.1`
: These two files are used by the automatic de-ANSI-fication support (see Section 7.14 [ANSI], page 55).

`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.

`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/cvs/?group=config` before making a release.

`config-ml.in`
: This file is not a program, it is a '`configure`' fragment used for multilib support (see Section 17.3 [Multilibs], page 81). This file is maintained in the GCC tree at `http://gcc.gnu.org/cvs.html`.

`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.

`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.

`mkinstalldirs`
: This script used to be a wrapper around '`mkdir -p`', which is not portable. Now we use 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 '`mkinstalldirs`' 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.

`symlink-tree`
: This program duplicates a tree of directories, using symbolic links instead of copying files. Such operation is performed when building multilibs (see Section 17.3 [Multilibs], page 81). This file is maintained in the GCC tree at `http://gcc.gnu.org/cvs.html`.

`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 downloaded from `http://www.gnu.org/software/texinfo/`.

`ylwrap`
: This program wraps `lex` and `yacc` and ensures that, for instance, multiple `yacc` instances can be invoked in a single directory in parallel.

3 Some example packages

3.1 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.in`'s 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`:

```
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:

```
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:

```
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!

3.2 A classic program

GNU hello is renowned for its classic simplicity and versatility. This section shows how Automake could be used with the GNU Hello package. The examples below are from the latest beta version of GNU Hello, but with all of the maintainer-only code stripped out, as well as all copyright comments.

Of course, GNU Hello is somewhat more featureful than your traditional two-liner. GNU Hello is internationalized, does option processing, and has a manual and a test suite.

Here is the ‘configure.ac’ from GNU Hello. **Please note:** The calls to AC_INIT and AM_INIT_AUTOMAKE in this example use a deprecated syntax. For the current approach, see the description of AM_INIT_AUTOMAKE in Section 5.6.1 [Public macros], page 19.

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(src/hello.c)
AM_INIT_AUTOMAKE(hello, 1.3.11)
AM_CONFIG_HEADER(config.h)

dnl Set of available languages.
ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.
AC_PROG_CC
AC_ISC_POSIX

dnl Checks for libraries.

dnl Checks for header files.
AC_STDC_HEADERS
AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.
AC_FUNC_ALLOCA
```

```
dnl Check for st_blksize in struct stat
AC_ST_BLKSIZE

dnl internationalization macros
AM_GNU_GETTEXT
AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \
           src/Makefile tests/Makefile tests/hello],
   [chmod +x tests/hello])
```

The 'AM_' macros are provided by Automake (or the Gettext library); the rest are standard Autoconf macros.

The top-level 'Makefile.am':

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

As you can see, all the work here is really done in subdirectories.

The 'po' and 'intl' directories are automatically generated using gettextize; they will not be discussed here.

In 'doc/Makefile.am' we see:

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

This is sufficient to build, install, and distribute the GNU Hello manual.

Here is 'tests/Makefile.am':

```
TESTS = hello
EXTRA_DIST = hello.in testdata
```

The script 'hello' is generated by configure, and is the only test case. 'make check' will run this test.

Last we have 'src/Makefile.am', where all the real work is done:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
hello_LDADD = $(INTLLIBS) $(ALLOCA)
localedir = $(datadir)/locale
INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"
```

3.3 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.

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

Note that there is no `true_SOURCES` definition. Automake will implicitly assume that there is a source file named '`true.c`', and define rules to compile '`true.o`' and link '`true`'. The '`true.o: 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
```

Also, these explicit rules do not work if the de-ANSI-fication feature is used (see Section 7.14 [ANSI], page 55). Supporting de-ANSI-fication requires a little more work:

```
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 techniques are 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. De-ANSI-fication will work automatically. In this instance, the names of the object files would be chosen by automake; they would be '`false-true.o`' and '`true-true.o`'. (The name of the object files rarely matters.)

4 Creating a ‘Makefile.in’

To create all the ‘`Makefile.in`’s 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 5 [configure], page 12) 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`’ runs `AC_CANONICAL_HOST`. Automake is distributed with several of these files (see Section 2.6 [Auxiliary Programs], page 5); 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 5.2 [Optional], page 13).

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

`-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.in`'s to follow Cygnus rules, instead of GNU or Gnits rules. For more information, see Chapter 21 [Cygnus], page 83.

`-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 2.2 [Strictness], page 2.

`--gnits`
: Set the global strictness to '`gnits`'. For more information, see Chapter 20 [Gnits], page 82.

`--gnu`
: Set the global strictness to '`gnu`'. For more information, see Chapter 20 [Gnits], page 82. 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 '`Makefile`'s; see Section 7.15 [Dependencies], page 55.

`--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.in`'s mentioned in '`configure.ac`'. This option causes it to only update those '`Makefile.in`'s 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 section “Top” in *The GNU Coding Standards*).
`obsolete`	obsolete features or constructions
`override`	user redefinitions of Automake rules or variables
`portability`	portability issues (e.g., use of `make` features that are known to be not portable)
`syntax`	weird syntax, unused variables, typos
`unsupported`	unsupported or incomplete features
`all`	all the warnings
`none`	turn off all the warnings
`error`	treat warnings as errors

A category can be turned off by prefixing its name with ‘`no-`’. For instance, ‘`-Wno-syntax`’ will hide the warnings about unused variables.

The categories output by default are ‘`syntax`’ and ‘`unsupported`’. Additionally, ‘`gnu`’ is enabled in ‘`--gnu`’ and ‘`--gnits`’ strictness.

‘`portability`’ warnings are currently disabled by default, but they will be enabled in ‘`--gnu`’ and ‘`--gnits`’ strictness in a future release.

The environment variable `WARNINGS` can contain a comma separated list of categories to enable. It will be taken into account before the command-line switches, this way ‘`-Wnone`’ will also ignore any warning category enabled by `WARNINGS`. This variable is also used by other tools like `autoconf`; unknown categories are ignored for this reason.

5 Scanning ‘configure.ac’

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.

5.1 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 5.6 [Macros], page 19).

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

`AC_CONFIG_FILES`
`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 Automake generated.

Files created by `AC_CONFIG_FILES` are removed by `make distclean`.

5.2 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_CONFIG_HEADERS`

Automake will generate rules to rebuild these headers. Older versions of Automake required the use of `AM_CONFIG_HEADER` (see Section 5.6 [Macros], page 19); this is no longer the case today.

`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'.

`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: 'config.guess', 'config.sub', 'depcomp', 'elisp-comp', 'compile', 'install-sh', 'ltmain.sh', 'mdate-sh', 'missing', 'mkinstalldirs', 'py-compile', 'texinfo.tex', and '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_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_LIBSOURCE`
`AC_LIBSOURCES`
`AC_LIBOBJ`
: Automake will automatically distribute any file listed in `AC_LIBSOURCE` or `AC_LIBSOURCES`.

  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_ALLOCA`, `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_F77`
: This is required if any Fortran 77 source is included. This macro is distributed with Autoconf version 2.13 and later. See section "Particular Program Checks" in *The Autoconf Manual*.

`AC_PROG_FC`
: This is the newer interface to Fortran source, replacing the older F77 interfaces. This macro is distributed with Autoconf version 2.50 and later. 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 7.11.4 [Mixing Fortran With C and C++], page 52). See Section 5.6 [Autoconf macros supplied with Automake], page 19.

`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_SUBST`
: The first argument is automatically defined as a variable in each generated '`Makefile.in`'. See section "Setting Output Variables" in *The Autoconf Manual*.

  If the Autoconf manual says that a macro calls `AC_SUBST` for *var*, or defines the output variable *var* then *var* will be defined in each '`Makefile.in`' generated by Automake. E.g. `AC_PATH_XTRA` defines `X_CFLAGS` and `X_LIBS`, so you can use these variables in any '`Makefile.am`' if `AC_PATH_XTRA` is called.

`AM_C_PROTOTYPES`
: This is required when using automatic de-ANSI-fication; see Section 7.14 [ANSI], page 55.

`AM_GNU_GETTEXT`
: This macro is required for packages that use GNU gettext (see Section 9.2 [gettext], page 63). It is distributed with gettext. If Automake sees this macro it ensures that the package meets some of gettext's requirements.

`AM_MAINTAINER_MODE`
: This macro adds a '`--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.in`'s. This macro defines the `MAINTAINER_MODE` conditional, which you can use in your own '`Makefile.am`'. See Section 26.2 [maintainer-mode], page 92.

`m4_include`
: Files included by '`configure.ac`' using this macro will be detected by Automake and automatically distributed. They will also appear as dependencies in '`Makefile`' rules.

  `m4_include` is seldom used by '`configure.ac`' authors, but can appear in '`aclocal.m4`' when `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 5.3 [Invoking aclocal], page 15).

5.3 Auto-generating aclocal.m4

Automake includes a number of Autoconf macros that can be used in your package (see Section 5.6 [Macros], page 19); some of them are actually required by Automake in certain situations. These macros must be defined in your ‘`aclocal.m4`’; otherwise they will not be seen by `autoconf`.

The `aclocal` program will automatically generate ‘`aclocal.m4`’ files based on the contents of ‘`configure.ac`’. This provides a convenient way to get Automake-provided macros, without having to search around. The `aclocal` mechanism allows other packages to supply their own macros (see Section 5.7 [Extending aclocal], page 22). You can also use it to maintain your own set of custom macros (see Section 5.8 [Local Macros], page 23).

At startup, `aclocal` scans all the ‘`.m4`’ files it can find, looking for macro definitions (see Section 5.5 [Macro search path], page 17). Then it scans ‘`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 ‘`aclocal.m4`’.

Putting the file that contains the macro definition into ‘`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 `aclocal`, use ‘`##`’ as the comment leader.

When a file selected by `aclocal` is located in a subdirectory specified as a relative search path with `aclocal`’s ‘`-I`’ argument, `aclocal` assumes the file belongs to the package and uses `m4_include` instead of copying it into ‘`aclocal.m4`’. This makes the package smaller, eases dependency tracking, and cause the file to be distributed automatically. (See Section 5.8 [Local Macros], page 23, 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 ‘pwd‘/reldir`’ instead of ‘`-I reldir`’ whenever some relative directory need to be considered outside the package.

The contents of ‘`acinclude.m4`’, if this file exists, are also automatically included in ‘`aclocal.m4`’. We recommend against using ‘`acinclude.m4`’ in new packages (see Section 5.8 [Local Macros], page 23).

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

5.4 aclocal options

`aclocal` accepts the following options:

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

`--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.

`--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.

`--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 can be used by a package to determine where to install a macro file.

`--verbose`
: Print the names of the files it examines.

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

5.5 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 automake 1.6.x, *APIVERSION* = `1.6`.

`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 5.4 [aclocal options], page 16).

As an example, suppose that `automake-1.6.2` was configured with ‘`--prefix=/usr/local`’. Then, the search path would be:

1. ‘`/usr/local/share/aclocal-1.6/`’
2. ‘`/usr/local/share/aclocal/`’

As explained in (see Section 5.4 [aclocal options], page 16), there are several options that can be used to change or extend this search path.

5.5.1 Modifying the macro search path: `--acdir`

The most erroneous option to modify the search path is `--acdir=`*`dir`*, which changes default directory and drops the *APIVERSION* directory. For example, if one specifies `--acdir=/opt/private/`, then the search path becomes:

1. ‘`/opt/private/`’

This option, `--acdir`, is intended for use by the internal automake test suite only; it is not ordinarily needed by end-users.

5.5.2 Modifying the macro search path: `-I` *`dir`*

Any extra directories specified using `-I` options (see Section 5.4 [aclocal options], page 16) 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*

5.5.3 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 directories, one per line, to be added to the search list. These directories are searched *after* all other directories.

For example, suppose '*acdir*/dirlist' contains the following:

```
/test1
/test2
```

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

If the '--acdir=*dir*' option is used, then aclocal will search for the 'dirlist' file in *dir*. In the '--acdir=/opt/private/' example above, aclocal would look for '/opt/private/dirlist'. Again, however, the '--acdir' option is intended for use by the internal automake test suite only; '--acdir' is not ordinarily needed by end-users.

'dirlist' is useful in the following situation: suppose that automake version 1.6.2 is installed with '--prefix=/usr' by the system vendor. Thus, the default search directories are

1. /usr/share/aclocal-1.6/
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-1.6/
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 Automake on your account and want `aclocal` to look for macros installed at other places on the system.

5.6 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`'.

5.6.1 Public macros

`AM_C_PROTOTYPES`
: Check to see if function prototypes are understood by the compiler. If so, define '`PROTOTYPES`' and set the output variables `U` and `ANSI2KNR` to the empty string. Otherwise, set `U` to '`_`' and `ANSI2KNR` to '`./ansi2knr`'. Automake uses these values to implement automatic de-ANSI-fication.

`AM_ENABLE_MULTILIB`
: This is used when a "multilib" library is being built. The first optional argument is the name of the '`Makefile`' being generated; it defaults to '`Makefile`'. The second option argument is used to find the top source directory; it defaults to the empty string (generally this should not be used unless you are familiar with the internals). See Section 17.3 [Multilibs], page 81.

`AM_INIT_AUTOMAKE([OPTIONS])`
`AM_INIT_AUTOMAKE(PACKAGE, VERSION, [NO-DEFINE])`
: Runs many macros required for proper operation of the generated Makefiles.

This macro has two forms, the first of which is preferred. In this form, `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 16 [Options], page 76).

The second, deprecated, form of `AM_INIT_AUTOMAKE` has 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:

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

you can modernize it as follows:

```
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, as in:

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

or by passing a third non-empty argument to the obsolete form.

`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 emacsen 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_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.

`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_WITH_DMALLOC`

Add support for the Dmalloc package. If the user runs `configure` with '`--with-dmalloc`', then define `WITH_DMALLOC` and add '`-ldmalloc`' to `LIBS`.

`AM_WITH_REGEX`

Adds '`--with-regex`' to the `configure` command line. If specified (the default), then the '`regex`' regular expression library is used, '`regex.o`' is put into `LIBOBJS`, and `WITH_REGEX` is defined. If '`--without-regex`' is given, then the `rx` regular expression library is used, and '`rx.o`' is put into `LIBOBJS`.

5.6.2 Obsolete macros

Although using some of the following macros was required in past releases, you should not used any of them in new code. Running `autoupdate` should adjust your '`configure.ac`' automatically (see section "Using `autoupdate` to Modernize '`configure.ac`'" in *The Autoconf Manual*).

`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 5.2 [Optional], page 13).

`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_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.

5.6.3 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`.

5.7 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 26.9 [Hard-Coded Install Paths], page 104, for arguments.

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 5.9 [Future of aclocal], page 24) will have to temporarily include all 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(2.57)
AC_DEFUN(AX_FOOBAR,
[AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

should be rewritten as

```
AC_DEFUN([AX_FOOBAR],
[AC_PREREQ(2.57)dnl
AC_REQUIRE([AX_SOMETHING])dnl
AX_FOO
AX_BAR
])
```

Wrapping the AC_PREREQ call inside the macro ensures that Autoconf 2.57 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).

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 last version of the macro and that the problem already hasn't 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 5.8 [Local Macros], page 23.

5.8 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 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

```
ACLOCAL_AMFLAGS = -I m4
```

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 ACLOCAL_AMFLAGS.

When 'aclocal -I m4' is run, it will build a '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 5.5 [Macro search path], page 17.

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 5.7 [Extending aclocal], page 22). 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.

5.9 The Future of `aclocal`

`aclocal` is expected to disappear. This feature really should not be offered by Automake. Automake should focus on generating '`Makefile`'s; dealing with M4 macros really is Autoconf's job. 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 5.8 [Local Macros], page 23, and take care of copying (and even updating) third-party macros from '`/usr/share/aclocal/`' into the local '`m4/`' directory.

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 itself 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, yum!), 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 5.7 [Extending aclocal], page 22).

6 Directories

For simple projects that distributes 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 instance one directory per program, per library or per module. The traditional approach is to build these subdirectory recursively: each directory contains its '`Makefile`' (generated from '`Makefile.am`'), and when `make` is run from the top level directory it enters each subdirectory in turn to build its contents.

6.1 Recursing subdirectories

In packages with subdirectories, 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 '`Makefile`'s (after configuration). This allows inclusion of libraries from packages that do not use Automake (such as `gettext`; see also Section 22.2 [Third-Party Makefiles], page 85).

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.

All `clean` rules are run in reverse order of build rules.

6.2 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 19 [Conditionals], page 81) 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 possibility, let's introduce `DIST_SUBDIRS`.

6.2.1 `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 6.1 [Subdirectories], page 25). 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` come 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 possibles 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.

6.2.2 Conditional 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.

```
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.

6.2.3 Conditional Subdirectories with `AC_SUBST`

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

```
...
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.

```
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`.

6.2.4 Non-configured Subdirectories

The semantic of `DIST_SUBDIRS` is 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 rule remove the generated 'Makefile's. 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 non-configured 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, non-configured 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 exist.

6.3 An Alternative Approach to Subdirectories

If you've ever read Peter Miller's excellent paper, Recursive Make Considered Harmful, the preceding sections on the use of subdirectories 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[3] 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
```

'`nobase_`' should be specified first when used in conjunction with either '`dist_`' or '`nodist_`' (see Chapter 13 [Dist], page 70). For instance:

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

Finally, note that a variable using the '`nobase_`' prefix can always be replaced by several variables, one for each destination directory (see Section 2.3 [Uniform], page 3). 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.

[3] We believe. This work is new and there are probably warts. See Chapter 1 [Introduction], page 1, for information on reporting bugs.

6.4 Nesting Packages

In the GNU Build System, packages can be nested to arbitrary depth. This means that a package can embedded other packages with their own 'configure', 'Makefile's, 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 'Makefile's 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 an 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
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 into search auxiliary script 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 these auxiliary scripts in the immediate parent and grand-parent directories. 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 22.2 [Third-Party Makefiles], page 85.

7 Building Programs and Libraries

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

7.1 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 7.2 [A Library], page 34) and libtool libraries (see Section 7.3 [A Shared Library], page 34).

7.1.1 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`, `pkglibdir`, 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 7.5 [Default _SOURCES], page 43).

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 7.8 [Yacc and Lex], page 47.

7.1.2 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 the global 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):

```
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 the *prog*_DEPENDENCIES variable. Each program depends on the contents of such a variable, but no further interpretation is done.

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.

7.1.3 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.

7.1.3.1 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:

```
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':

```
...
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.

7.1.3.2 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:

```
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 19 [Conditionals], page 81).

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 '`+=`'.

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

7.1.4 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.

7.1.4.1 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 7.16 [EXEEXT], page 56, 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}'])`'.

7.1.4.2 Conditional programs using Automake conditionals

You can also use Automake conditionals (see Chapter 19 [Conditionals], page 81) 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
```

7.2 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 7.3 [A Shared Library], page 34, 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 = ...
```

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 2.4 [Canonicalization], page 4), 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 8.4 [Sources], page 58).

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). `AR` and `ARFLAGS` default to `ar` and `cru` respectively; you can override these two variables my setting them in your '`Makefile.am`', by `AC_SUBST`ing them from your '`configure.ac`', or by defining a per-library `maude_AR` variable (see Section 7.4 [Program and Library Variables], page 40).

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

7.3 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.

7.3.1 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 '`Makefile`'s 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 7.3.4 [Conditional Libtool Sources], page 37).

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*).

7.3.2 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.

7.3.3 Building Libtool Libraries Conditionally

Like conditional programs (see Section 7.1.4 [Conditional Programs], page 33), there are two main ways to build conditional libraries: using Automake conditionals or using Autoconf AC_SUBSTitutions.

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' by 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 ...
```

7.3.4 Libtool Libraries with Conditional Sources

Conditional compilation of sources in a library can be achieved in the same way as conditional compilation of sources in a program (see Section 7.1.3 [Conditional Sources], page 32). 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 7.1.3 [Conditional Sources], page 32, 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
```

7.3.5 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 7.11.4.1 [How the Linker is Chosen], page 54.

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 7.3.4 [Conditional Libtool Sources], page 37), and the `nodist_` prefix means the listed sources are not to be distributed (see Section 7.4 [Program and Library Variables], page 40). 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`.

7.3.6 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 starts with `lib`. However, when building a dynamically loadable module you might wish to use a "nonstandard" name. Automake will not complain about such nonstandard name 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 7.5 [Default _SOURCES], page 43).

7.3.7 _LIBADD and _LDFLAGS

As shown in previous sections, the '`*library*_LIBADD`' variable should be used to list extra libtool objects ('`.lo`' files) or libtool libraries ('`.la`') to add to *library*.

The '`*library*_LDFLAGS`' variable is the place to list additional libtool flags, such as '`-version-info`', '`-static`', and a lot more. See section "Link mode" in *The Libtool Manual*.

7.3.8 `LTLIBOBJS` and `LTALLOCA`

Where an ordinary library might include '`$(LIBOBJS)`' or '`$(ALLOCA)`' (see Section 7.6 [LIBOBJS], page 44), a libtool library must use '`$(LTLIBOBJS)`' or '`$(LTALLOCA)`'. This is required because the object files that libtool operates on do not necessarily end in '`.o`'.

Nowadays, the computation of `LTLIBOBJS` from `LIBOBJS` is performed automatically by Autoconf (see section "`AC_LIBOBJ` vs. `LIBOBJS`" in *The Autoconf Manual*).

7.3.9 Common Issues Related to Libtool's Use

7.3.9.1 '`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` command (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.

7.3.9.2 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`'.

```
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 26.6 [renamed objects], page 98, this happens automatically when per-targets flags are used.

```
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.

7.4 Program and Library Variables

Associated with each program are 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 7.13 [Support for Other Languages], page 54). 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:

```
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 16 [Options], page 76).

`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.[4] 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 '`$(AR) $(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 templates that should go into a library. For instance, the SGI C++ compiler likes this variable set like so:

```
libmaude_a_AR = $(CXX) -ar -o
```

[4] There are other, more obscure reasons for this limitation as well.

`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 7.2 [A Library], page 34).

  In the case of libtool libraries, `maude_LIBADD` can also refer to other libtool libraries.

`maude_LDADD`
: Extra objects (‘`*.$(OBJDIR)`’) 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 7.1.2 [Linking], page 31).

  `_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:

```
maude_LDADD = $(X_PRE_LIBS) $(X_LIBS) $(X_EXTRA_LIBS)
```

`maude_LDFLAGS`
: This variable is used to pass extra flags to the link step of a program or a shared library.

`maude_DEPENDENCIES`
: 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 the ‘`_DEPENDENCIES`’ variable. Each program depends on the contents of such a variable, but no further interpretation is done.

  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 ‘`$(LIBOBJS)`’ and ‘`$(ALLOCA)`’; these are left because it is known that they will not cause an invalid value for ‘`_DEPENDENCIES`’ to be generated.

`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 as arguments. Note that the name of the underlying program is *not* passed to ‘`_LINK`’; typically one uses ‘`$@`’:

```
maude_LINK = $(CCLD) -magic -o $@
```

`maude_CCASFLAGS`
`maude_CFLAGS`
`maude_CPPFLAGS`
`maude_CXXFLAGS`
`maude_FFLAGS`
`maude_GCJFLAGS`
`maude_LFLAGS`
`maude_OBJCFLAGS`
`maude_RFLAGS`
`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`', '`_RFLAGS`', 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 26.6 [renamed objects], page 98.)

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 26.5 [Flag Variables Ordering], page 96, 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.

7.5 Default _SOURCES

_SOURCES variables are used to specify source files of programs (see Section 7.1 [A Program], page 30), libraries (see Section 7.2 [A Library], page 34), and Libtool libraries (see Section 7.3 [A Shared Library], page 34).

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 '.c'. (Defaulting to C is terrible but we are stuck with it for historical reasons.)

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 exist.)

Default sources are mainly useful in test suites, when building many tests programs each from a single source. For instance, in

```
check_PROGRAMS = test1 test2 test3
```

'test1', 'test2', and 'test3' will be built from 'test1.c', 'test2.c', and 'test3.c'.

Another case where is this convenient is building many Libtool modules ('moduleN.la'), each defined in its own file ('moduleN.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 3.3 [true], page 8; this happens when all the constituents of a target have already been compiled and need just 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.

```
bin_PROGRAMS = target
target_SOURCES =
target_LDADD = libmain.a libmisc.a
```

7.6 Special handling for LIBOBJS and ALLOCA

The '$(LIBOBJS)' and '$(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 AC_LIBOBJ, AC_REPLACE_FUNCS (see section "Generic Function Checks" in *The Autoconf Manual*), or AC_FUNC_ALLOCA (see section "Particular Function Checks" in *The Autoconf Manual*). Many other Autoconf macros call AC_LIBOBJ or AC_REPLACE_FUNCS to populate '$(LIBOBJS)'.

Using these variables is very similar to doing conditional compilation using AC_SUBST variables, as described in Section 7.1.3 [Conditional Sources], page 32. That is, when building a program, '$(LIBOBJS)' and '$(ALLOCA)' should be added to the associated '*_LDADD' variable, or to the '*_LIBADD' variable when building a library. However there is no need to list the corresponding sources in 'EXTRA_*_SOURCES' nor to define '*_DEPENDENCIES'. Automake automatically adds '$(LIBOBJS)' and '$(ALLOCA)' to the dependencies, and it will discover the list of corresponding source files automatically (by tracing the invocations of the AC_LIBSOURCE Autoconf macros).

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 LIBOBJS or ALLOCA.

```
# 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 does not yet use this information; it knows the source files are expected 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. 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 = ...
```

Please note it would be wrong to use the variables '`$(LIBOBJS)`' or '`$(ALLOCA)`' in '`src/Makefile.am`', because these variables contains unprefixed object names, and, for instance, '`malloc.$(OBJEXT)`' is not buildable in the '`src/`' directory. (Actually if you try using '`$(LIBOBJS)`' in '`src/`', Automake will require a copy of '`malloc.c`', '`memcmp.c`', '`strdup.c`', '`alloca.c`' in '`src/`' too.)

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*.

7.7 Variables used when building a program

Occasionally it is useful to know which '`Makefile`' variables Automake uses for compilations; 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 particular it generates '`-I$(srcdir)`', '`-I.`', 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.

  `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.

7.8 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++ file. Files with the extension '`.y`' will be turned into '`.c`' files; likewise, '`.yy`' will become '`.cc`'; '`.y++`', '`c++`'; and '`.yxx`', '`.cxx`'.

Likewise, lex source files can be used to generate C or C++; the extensions '`.l`', '`.ll`', '`.l++`', and '`.lxx`' 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 `YFLAGS` and `AM_YFLAGS`. The former is a user variable and the latter 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 8.4 [Sources], page 58) as follows.

```
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 5.6 [Macros], page 19) is recommended.

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

Automake makes it possible to include multiple `yacc` (or `lex`) source files in a single program. When there is more than one distinct `yacc` (or `lex`) source file in a directory, Automake uses a small program called `ylwrap` to run `yacc` (or `lex`) in a subdirectory. 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. The `ylwrap` program is distributed with Automake. It should appear in the directory specified by `AC_CONFIG_AUX_DIR`, or one of its default locations (see section "Finding 'configure' Input" in *The Autoconf Manual*).

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`:

```
#define yymaxdepth c_maxdepth
#define yyparse c_parse
#define yylex   c_lex
#define yyerror c_error
#define yylval  c_lval
#define yychar  c_char
#define yydebug c_debug
#define yypact  c_pact
#define yyr1    c_r1
#define yyr2    c_r2
#define yydef   c_def
#define yychk   c_chk
#define yypgo   c_pgo
#define yyact   c_act
#define yyexca  c_exca
#define yyerrflag c_errflag
#define yynerrs c_nerrs
#define yyps    c_ps
#define yypv    c_pv
#define yys     c_s
#define yy_yys  c_yys
#define yystate c_state
#define yytmp   c_tmp
#define yyv     c_v
#define yy_yyv  c_yyv
#define yyval   c_val
#define yylloc  c_lloc
#define yyreds  c_reds
#define yytoks  c_toks
#define yylhs   c_yylhs
#define yylen   c_yylen
#define yydefred c_yydefred
#define yydgoto c_yydgoto
#define yysindex c_yysindex
#define yyrindex c_yyrindex
```

```
#define yygindex c_yygindex
#define yytable  c_yytable
#define yycheck  c_yycheck
#define yyname   c_yyname
#define yyrule   c_yyrule
```

For each define, replace the 'c_' prefix with whatever you like. These defines work for `bison`, `byacc`, and traditional `yacc`s. 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.

7.9 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.

7.10 Assembly Support

Automake includes some support for assembly code.

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) are passed to the compilation.

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`' and '`.S`' are recognized by `automake` as being files containing assembly code.

7.11 Fortran Support

7.11.1 Overview

Automake includes full support for Fortran. There are *two* Fortran interfaces within automake, one specific to Fortran 77 (the so-called 'F77' interface) and one for all Fortran dialects (the 'FC' interface), which should include Fortran 77. The newer FC interface is preferred, and new code should use this in preference to the older F77 interface. The description below refers to the FC interface, but makes remarks about the F77 interface where these seem relevant; for full details, see the Autoconf manual.

Probably the only time you will have to use the F77 interface is when your project uses both Fortran 77 and Fortran 9x sources *and* the Fortran 77 code uses features which a Fortran 9x compiler will typically not support. If you are in this situation, you might have to be clever about using both interfaces; but in this case you quite probably have a variety of portability problems anyway.

Any package including Fortran 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 source file is seen:

`FC` The name of the Fortran compiler.

`FCFLAGS` Any flags to pass to the Fortran compiler.

`AM_FCFLAGS`
The maintainer's variant of `FCFLAGS`.

`RFLAGS` Any flags to pass to the Ratfor compiler.

`AM_RFLAGS`
The maintainer's variant of `RFLAGS`.

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

`FCLINK` The command used to actually link a pure Fortran program or shared library.

Automake still provides the older Fortran 77 support, using the '`AC_PROG_F77`' macro, and the substituted variables '`F77`', '`FFLAGS`', '`AM_FFLAGS`', '`F77COMPILE`' and '`FLINK`', instead of the analogous variables described above.

Automake can handle preprocessing Fortran 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 and other languages (see Section 7.11.4 [Mixing Fortran With C and C++], page 52).

Automake has in the past associated the file extension '`.f`' with the F77 interface. Though this is preserved at present for the sake of backward compatibility, it is rather inconsistent with the intention that the FC interface is the preferred one. If you wish to change this (and we think that you should), then you can do so by defining the environment variable '`AUTOMAKE_F_IS_FC`'. If this is set equal to '`1`', then the extension '`.f`' is associated with the FC interface; if it is zero or undefined, the extension is associated with the F77 interface; we hope to change the default (the unset case) to FC in future.

[5] See also the information on rules in section "Catalogue of Rules" in *The GNU Make Manual*.

7.11.2 Compiling Fortran Files

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

'`.f`' `$(FCCOMPILE) -c -o $ $<`

'`.r`' `$(RCOMPILE) -c -o $ $<`

The compilation commands in turn expand into

`$(FCCOMPILE)`
`$(FC) $(AM_FCFLAGS) $(FCFLAGS)`

`$(RCOMPILE)`
`$(FC) $(AM_FCFLAGS) $(FCFLAGS) $(AM_RFLAGS) $(RFLAGS)`

This is for the case where '`.f`' is associated with the FC interface; if '`.f`' is associated with the F77 interface, the commands are different but analogous.

The handling of preprocessable Fortran is rather more complicated – see the next section.

7.11.3 Preprocessing Fortran

It is possible to use Fortran which has cpp-style directives within it. Some Fortran compilers, such as `g77`, can process these directives internally, and so need no separate preprocessing stage; in other cases, the code must be compiled indirectly, with a preprocessor producing pure Fortran code which is only then passed to the compiler.

Because of the syntactical differences in the underlying languages, however it is not always possible to do this using the `cpp` program, and you may need help from a separate Fortran-specific preprocessor.[6]

Automake's support for preprocessed Fortran is tightly bound to the Autoconf `AC_PROG_FPP` macro (introduced in version XXX), which determines if the Fortran compiler can cope with preprocessor directives, and if not finds a preprocessor which can. It looks for files with a '`.F`' extension (the extension is configurable).

The `AC_PROG_FPP` macro takes as argument a list of features that you require a preprocessor to support (handling of '`-I`' and '`-D`' options, and so on), and does one of two things:

1. checks whether your Fortran compiler can handle the required set of preprocessor directives internally; and if not
2. checks whether it can find a separate preprocessor which can turn the preprocessable code into a form which the Fortran compiler *can* handle.

The first is referred to as 'direct' mode, and the second as 'indirect' mode. See the Autoconf documentation of the `AC_PROG_FPP` macro for more details, but note that the `@FPPDIRECT_TRUE@` and `_FALSE` variables referred to there are the implementation of an automake conditional, so that on the (hopefully rare) occasions when you have to handle something by hand in a '`Makefile.am`', you can handle both direct and indirect mode using a construction like:

[6] There is no formal standard for Fortran preprocessors, but Sun have produced a preprocessor `fpp`, which is available for download at `http://www.netlib.org/fortran/`; this comes with a free-ish but not quite open-source licence. The documentation within that distribution is, in effect, a useful specification of a Fortran preprocessor syntax.

```
if FPPDIRECT
## do things appropriate for direct mode
else !FPPDIRECT
## do things appropriate for indirect mode
endif !FPPDIRECT
```

Handling Ratfor is a type of preprocessing, but this is rather more straightforward. The file '`N.f`' is made automatically from '`N.r`' by a rule which runs just the preprocessor to convert a Ratfor source file into a strict Fortran 77 source file. The precise command used is as follows:

'`.r`' `$(FC) $(AM_FCFLAGS) $(FCFLAGS) $(AM_RFLAGS) $(RFLAGS)`

(or the corresponding alternatives for the F77 interface).

7.11.4 Mixing Fortran With C and C++

Automake currently provides *limited* support for creating programs and shared libraries that are a mixture of Fortran and C and/or C++. However, there are many other issues related to mixing Fortran with other languages that are *not* (currently) handled by Automake, but that are handled by other packages[7].

[7] For example, the cfortran package 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.

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 `FCLIBS` by the `AC_FC_LIBRARY_LDFLAGS` Autoconf macro supplied with newer versions of Autoconf (Autoconf version 2.13 and later; substitute `FLIBS` and `AC_F77_LIBRARY_LDFLAGS` for the F77 interface). See section "Fortran Compiler Characteristics" in *The Autoconf Manual*.

7.11.4.1 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. C++ (`CXXLINK`)
3. Fortran (`FCLINK`)
4. Fortran 77 (`F77LINK`)
5. Objective C (`OBJCLINK`)
6. 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 7.3.5 [Libtool Convenience Libraries], page 37, for a trick with `nodist_EXTRA_..._SOURCES`.

7.12 Java Support

Automake includes support for compiled Java, using `gcj`, the Java front end to the GNU Compiler Collection.

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.

7.13 Support for Other Languages

Automake currently only includes full support for C, C++ (see Section 7.9 [C++ Support], page 49), Fortran (see Section 7.11 [Fortran Support], page 49), and Java (see Section 7.12 [Java Support], page 54). 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 17.2 [Suffixes], page 80).

7.14 Automatic de-ANSI-fication

Although the GNU standards allow the use of ANSI C, this can have the effect of limiting portability of a package to some older compilers (notably the SunOS C compiler).

Automake allows you to work around this problem on such machines by *de-ANSI-fying* each source file before the actual compilation takes place.

If the 'Makefile.am' variable AUTOMAKE_OPTIONS (see Chapter 16 [Options], page 76) contains the option 'ansi2knr' then code to handle de-ANSI-fication is inserted into the generated 'Makefile.in'.

This causes each C source file in the directory to be treated as ANSI C. If an ANSI C compiler is available, it is used. If no ANSI C compiler is available, the ansi2knr program is used to convert the source files into K&R C, which is then compiled.

The ansi2knr program is simple-minded. It assumes the source code will be formatted in a particular way; see the ansi2knr man page for details.

Support for de-ANSI-fication requires the source files 'ansi2knr.c' and 'ansi2knr.1' to be in the same package as the ANSI C source; these files are distributed with Automake. Also, the package 'configure.ac' must call the macro AM_C_PROTOTYPES (see Section 5.6 [Macros], page 19).

Automake also handles finding the ansi2knr support files in some other directory in the current package. This is done by prepending the relative path to the appropriate directory to the ansi2knr option. For instance, suppose the package has ANSI C code in the 'src' and 'lib' subdirectories. The files 'ansi2knr.c' and 'ansi2knr.1' appear in 'lib'. Then this could appear in 'src/Makefile.am':

```
AUTOMAKE_OPTIONS = ../lib/ansi2knr
```

If no directory prefix is given, the files are assumed to be in the current directory.

Note that automatic de-ANSI-fication will not work when the package is being built for a different host architecture. That is because automake currently has no way to build ansi2knr for the build machine.

Using LIBOBJS with source de-ANSI-fication used to require hand-crafted code in 'configure' to append '$U' to basenames in LIBOBJS. This is no longer true today. Starting with version 2.54, Autoconf takes care of rewriting LIBOBJS and LTLIBOBJS. (see section "AC_LIBOBJ vs. LIBOBJS" in *The Autoconf Manual*)

7.15 Automatic dependency tracking

As a developer it is often painful to continually update the 'Makefile.in' whenever the include-file dependencies change in a project. Automake supplies a way to automatically track dependency changes.

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 27.2 [Dependency Tracking Evolution], page 117) 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`'.

7.16 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:

```
bin_PROGRAMS = liver
```

to this:

```
bin_PROGRAMS = liver$(EXEEXT)
```

The targets Automake generates are likewise given the '`$(EXEEXT)`' extension.

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.

With Autoconf 2.13 and earlier, you must explicitly use `AC_EXEEXT` to get this support. With Autoconf 2.50, `AC_EXEEXT` is run automatically if you configure a compiler (say, through `AC_PROG_CC`).

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.

Unfortunately, due to the change in Autoconf 2.50, this means you must always add this extension. However, this is a problem 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 16 [Options], page 76) 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.

8 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.

8.1 Executable Scripts

It is possible to define and install programs that are scripts. Such programs are listed using the SCRIPTS primary name. Automake doesn't define any dependencies for scripts; the 'Makefile.am' should include the appropriate rules.

Automake does not assume that scripts are derived objects; such objects must be deleted by hand (see Chapter 12 [Clean], page 70).

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)

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
```

Because—as we have just seen—scripts can be built, they are not distributed by default. Scripts that should be distributed can be specified using a dist_ prefix as in other primaries. For instance, the following 'Makefile.am' declares that 'my_script' should be distributed and installed in '$(sbindir)'.

```
dist_sbin_SCRIPTS = my_script
```

Script objects can be installed in bindir, sbindir, libexecdir, or pkgdatadir.

Scripts that need not being installed can be listed in noinst_SCRIPTS, and among them, those which are needed only by 'make check' should go in check_SCRIPTS.

8.2 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 2.3 [Uniform], page 3). For instance,

```
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,

```
nobase_include_HEADERS = foo.h bar/bar.h
```

will install the two files as '$(includedir)/foo.h' and '$(includedir)/bar/bar.h' (see Section 6.3 [Alternative], page 28).

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 one convenient library or program, we recommend listing it in the program's or library's _SOURCES variable (see Section 7.1.1 [Program Sources], page 30) 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 8.4 [Sources], page 58).

8.3 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 ...
```

8.4 Built sources

Because Automake's automatic dependency tracking works as a side-effect of compilation (see Section 7.15 [Dependencies], page 55) 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.

8.4.1 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 8.4 [Sources], page 58).

Here is a tentative '`Makefile.am`'.

```
# 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 7.15 [Dependencies], page 55). 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, not installed, and it is not even being built on-time. One may wonder what 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 17.1 [Tags], page 79). 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 8.4 [Sources], page 58).

```
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' get 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' \
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`'.

```
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`' work 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`'.

```
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`'.

```
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.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.

9 Other GNU Tools

Since Automake is primarily intended to generate 'Makefile.in's for use in GNU programs, it tries hard to interoperate with other GNU tools.

9.1 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 5.6 [Macros], page 19).

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 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.

```
lisp_LISP = file1.el file2.el
ELCFILES =
```

ELCFILES is an internal Automake variable that normally lists all '.elc' files that must be byte-compiled. Automake defines ELCFILES automatically from lisp_LISP. Emptying this variable explicitly prevents byte-compilation to occur.

Since Automake 1.8, we now recommend using lisp_DATA instead. As in

```
lisp_DATA = file1.el file2.el
```

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.

9.2 Gettext

If `AM_GNU_GETTEXT` is seen in '`configure.ac`', then Automake turns on support for GNU gettext, a message catalog system for internationalization (see section "GNU Gettext" in *GNU gettext utilities*).

The `gettext` support in Automake requires the addition of two subdirectories to the package, '`intl`' and '`po`'. Automake insures that these directories exist and are mentioned in `SUBDIRS`.

9.3 Libtool

Automake provides support for GNU Libtool (see section "Introduction" in *The Libtool Manual*) with the `LTLIBRARIES` primary. See Section 7.3 [A Shared Library], page 34.

9.4 Java

Automake provides some minimal support for Java compilation with the `JAVA` primary.

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 2.5 [User Variables], page 4).

`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 an `sh` 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.)

9.5 Python

Automake provides support for Python compilation with the `PYTHON` primary.

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 byte-compiled (‘`.pyo`’) 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.

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 following variables to list you Python source files in your variables: `python_PYTHON`, `pkgpython_PYTHON`, `pyexecdir_PYTHON`, `pkgpyexecdir_PYTHON`, depending where you want your files installed.

`AM_PATH_PYTHON ([`*`VERSION`*`], [`*`ACTION-IF-FOUND`*`], [`*`ACTION-IF-NOT-FOUND`*`])` [Macro]

Search 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, the default is to abort configure. This is fine when Python is an absolute requirement for the package. Therefore if Python >= 2.2 is only *optional* to the package, `AM_PATH_PYTHON` could be called as follows.

```
AM_PATH_PYTHON(2.2,, :)
```

`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., ‘`1.5`’). 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.

`pkgpyexecdir`
: This is a convenience variable that is defined as '`$(pyexecdir)/$(PACKAGE)`'.

All 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 26.9 [Hard-Coded Install Paths], page 104.

10 Building documentation

Currently Automake provides support for Texinfo and man pages.

10.1 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. The '`.info`' files are built by '`make all`' and installed by '`make install`' (unless you use '`no-installinfo`', see below). The other files can be built on request by '`make dvi`', '`make ps`', '`make pdf`' and '`make html`'.

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.

UPDATED
: This holds the date the primary '.texi' file was last modified.

UPDATED-MONTH
: This holds the name of the month in which the primary '.texi' file was last modified.

The 'version.texi' support requires the mdate-sh script; this script 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 'gpl.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 = gpl.texi
```

By default, Automake requires the file 'texinfo.tex' to appear in the same directory as the Texinfo source (this can be changed using the TEXINFO_TEX variable, see below). However, 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. Automake supplies 'texinfo.tex' if '--add-missing' is given.

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'. This can be prevented via the no-installinfo option.

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 2.5 [User Variables], page 4) 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`
`AM_MAKEINFOHTMLFLAGS`
: 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
```

  `AM_MAKEINFOHTMLFLAGS` defaults to '$(AM_MAKEINFOFLAGS)'. This means that defining `AM_MAKEINFOFLAGS` without defining `AM_MAKEINFOHTMLFLAGS` will impact builds of both '.info' and '.html' files.

`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 build 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
```

10.2 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 'l' 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 'man*SECTION*dir', 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`'.

Here is how the man pages are handled in GNU `cpio` (which includes both Texinfo documentation and man pages):

```
man_MANS = cpio.1 mt.1
EXTRA_DIST = $(man_MANS)
```

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.

The `nobase_` prefix is meaningless for man pages and is disallowed.

11 What Gets Installed

11.1 Basics of installation

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`'.

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`'.

11.2 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`.

11.3 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. Section 22.1 [Extending], page 84 gives some examples.

11.4 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:

```
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. For more information, see section "Makefile Conventions" in *The GNU Coding Standards*.

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`.

11.5 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.

12 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 22.1 [Extending], page 84). A common case is deleting a directory, for instance, a directory created by the test suite:

```
clean-local:
        -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`'.

13 What Goes in a Distribution

13.1 Basics of distribution

The `dist` rule in the generated '`Makefile.in`' can be used to generate a gzip'd `tar` file and other flavors of archive for distribution. The files is named based on the `PACKAGE` and `VERSION` variables defined by `AM_INIT_AUTOMAKE` (see Section 5.6 [Macros], page 19); more precisely the gzip'd `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`'s and '`Makefile.in`'s. 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`'. 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 'Makefile.am's (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 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 19 [Conditionals], page 81), 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 6.2 [Conditional Subdirectories], page 25).

13.2 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
```

13.3 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 tar (or shar) file is created. One way to use this is for distributing files in subdirectories for which a new 'Makefile.am' is overkill:

```
dist-hook:
        mkdir $(distdir)/random
        cp -p $(srcdir)/random/a1 $(srcdir)/random/a2 $(distdir)/random
```

Another way to 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 `find $(distdir)/doc -name CVS`
```

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 6.4 [Subpackages], page 29), then '$(distdir)' and '$(top_distdir)' are relative to the package where 'make dist' was run, not to any sub-packages involved.

13.4 Checking the distribution

Automake also generates a distcheck rule that can be of help to ensure that a given distribution will actually work. distcheck makes a distribution, then tries to do a VPATH build, run the test suite, and finally make another tarfile to ensure the distribution is self-contained.

Building the package involves running './configure'. If you need to supply additional flags to configure, define them in the DISTCHECK_CONFIGURE_FLAGS variable, either in your top-level 'Makefile.am', or on the command line when invoking make.

If the distcheck-hook rule is defined in your top-level 'Makefile.am', then it will be invoked by distcheck after the new distribution has been unpacked, but before the unpacked copy is configured and built. Your 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 distcheck-hook as well as DISTCHECK_CONFIGURE_FLAGS are not honored in a subpackage 'Makefile.am', but the DISTCHECK_CONFIGURE_FLAGS are passed down to the configure script of the subpackage.

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 12 [Clean], page 70).

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 = \
  find -type f -exec sh -c 'test -f $(srcdir)/{} || echo {}' ';'
```

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 a FAQ entry about this (see Section 26.4 [distcleancheck], page 94), make sure you read it before playing with `distcleancheck_listfiles`.

`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:
        @:
```

13.5 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.

`dist-gzip`
: Generate a gzip tar archive of the distribution.

`dist-shar`
: Generate a shar archive of the distribution.

`dist-zip`
: Generate a zip archive of the distribution.

`dist-tarZ`
: Generate a compressed tar archive of the distribution.

The rule `dist` (and its historical synonym `dist-all`) will create archives in all the enabled formats, Chapter 16 [Options], page 76. By default, only the `dist-gzip` target is hooked to `dist`.

14 Support for test suites

Automake supports two forms of test suites.

14.1 Simple Tests

If the variable `TESTS` is defined, its value is taken to be a list of programs to run in order to do the testing. The programs can either be derived objects or source objects; the generated rule will look both in `srcdir` and '`.`'. Programs needing data files should look for them in `srcdir` (which is both an environment variable and a make variable) so they work when building in a separate directory (see section "Build Directories " in *The Autoconf Manual*), and in particular for the `distcheck` rule (see Chapter 13 [Dist], page 70).

The number of failures will be printed at the end of the run. If a given test program exits with a status of 77, then its result is ignored in the final count. This feature allows non-portable tests to be ignored in environments where they don't make sense.

The variable `TESTS_ENVIRONMENT` can be used to set environment variables for the test run; the environment variable `srcdir` is set in the rule. If all your test programs are scripts, you can also set `TESTS_ENVIRONMENT` to an invocation of the shell (e.g. '`$(SHELL) -x`'); this can be useful for debugging the tests.

You may define the variable `XFAIL_TESTS` to a list of tests (usually a subset of `TESTS`) that are expected to fail. This will reverse the result of those tests.

Automake ensures that each program listed in `TESTS` is built before any tests are run; you can list both source and derived programs in `TESTS`. 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.

14.2 DejaGnu Tests

If `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 2.5 [User Variables], page 4). 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.

For more information regarding DejaGnu test suites, see section "Top" in *The DejaGnu Manual*.

In either case, the testing is done via '`make check`'.

14.3 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.

15 Rebuilding Makefiles

Automake generates rules to automatically rebuild 'Makefile's, '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 variable should be defined in all 'Makefile's 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 'Makefile's, 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.ac's 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.

16 Changing Automake's Behavior

Various features of Automake can be controlled by options in the '`Makefile.am`'. Such options are applied on a per-'`Makefile`' basis when listed in a special '`Makefile`' variable named `AUTOMAKE_OPTIONS`. They are applied globally to all processed '`Makefiles`' when listed in the first argument of `AM_INIT_AUTOMAKE` in '`configure.ac`'. Currently understood options are:

'`gnits`'
'`gnu`'
'`foreign`'
'`cygnus`'
: Set the strictness as appropriate. The '`gnits`' option also implies options '`readme-alpha`' and '`check-news`'.

'`ansi2knr`'
'`*path*/ansi2knr`'
: Turn on automatic de-ANSI-fication. See Section 7.14 [ANSI], page 55. If preceded by a path, the generated '`Makefile.in`' will look in the specified directory to find the '`ansi2knr`' program. The path should be a relative path to another directory in the same distribution (Automake currently does not check this).

'`check-news`'
: Cause '`make dist`' to fail unless the current version number appears in the first few lines of the '`NEWS`' file.

'`dejagnu`' Cause `dejagnu`-specific rules to be generated. See Chapter 14 [Tests], page 73.

'`dist-bzip2`'
: Hook `dist-bzip2` 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.

'`no-define`'
: This options is meaningful only when passed as an argument to `AM_INIT_AUTOMAKE`. It will prevent the `PACKAGE` and `VERSION` variables to be `AC_DEFINE`d.

'`no-dependencies`'
: This is similar to using '`--include-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 7.15 [Dependencies], page 55). 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.

'`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.MINORALPHA*', where *ALPHA* is a letter; it should be omitted for non-alpha releases.

'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 sh-utils) 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 Win32 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 instance, if the source file is 'subdir/file.cxx', then the output file would be 'subdir/file.o'.

'tar-v7'
'tar-ustar'
'tar-pax'

These three mutually exclusive options select the tar format to use when generating tarballs with 'make dist'. (The tar file created is then compressed according to the set of 'no-dist-gzip', 'dist-bzip2' and 'dist-tarZ' options in use.)

These options must be passed as argument to AM_INIT_AUTOMAKE (see Section 5.6 [Macros], page 19) because they can require additional configure checks. Automake will complain if it sees such options in an AUTOMAKE_OPTIONS variable.

'tar-v7' selects the old V7 tar format. This is the historical default. This antiquated format is understood by all tar implementations and supports file names with up to 99 characters. When given longer file names some tar implementations will diagnose the problem while other will generate broken tarballs or use non-portable extensions. Furthermore, the V7 format cannot store empty directories. When using this format, consider using the 'filename-length-max=99' option to catch file names too long.

'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).

'`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 discussion 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 '`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.

'`-W`*`category`*' or '`--warnings=`*`category`*'
These options behave exactly like their command-line counterpart (see Chapter 4 [Invoking Automake], page 10). 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 5.6 [Macros], page 19.

17 Miscellaneous Rules

There are a few rules and variables that didn't fit anywhere else.

17.1 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_`*`prog`*`_SOURCES` 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`'.

Here is how Automake generates tags for its source, and for nodes in its Texinfo file:

```
ETAGS_ARGS = automake.in --lang=none \
 --regex='/^@node[ \t]+\([^,]+\)/\1/' automake.texi
```

If you add file names to `ETAGS_ARGS`, you will probably also want to define `TAGS_DEPENDENCIES`. The contents of this variable are added directly to the dependencies for the `tags` rule.

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.

Finally, Automake also emit rules to support the GNU Global Tags program. 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`.

17.2 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 an suffix rule for your language:

```
.foo.o:
        foocc -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 which extensions you are defining your suffix rule for. This usually happens when your extensions 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 to misinterpret '`.idlC.cpp:`' 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.

17.3 Support for Multilibs

Automake has support for an obscure feature called multilibs. A *multilib* is a library that is built for multiple different ABIs at a single time; each time the library is built with a different target flag combination. This is only useful when the library is intended to be cross-compiled, and it is almost exclusively used for compiler support libraries.

The multilib support is still experimental. Only use it if you are familiar with multilibs and can debug problems you might encounter.

18 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`'.

19 Conditionals

Automake supports a simple type of conditionals.

Before using a conditional, you must define it by using `AM_CONDITIONAL` in the '`configure.ac`' file (see Section 5.6 [Macros], page 19).

`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 7.1.4 [Conditional Programs], page 33).

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 by an `end`:

```
if DEBUG
DBG = debug
else !DEBUG
DBG =
endif !DEBUG
```

Unbalanced conditions are errors.

Note that conditionals in Automake 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 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.
- 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'.

The '--gnits' option does everything that '--gnu' does, and checks the following as well:

- 'make installcheck' will check to make sure that the '--help' and '--version' really print a usage message and a version string, respectively. This is the 'std-options' option (see Chapter 16 [Options], page 76).
- 'make dist' will check to make sure the 'NEWS' file has been updated to the current version.
- VERSION is checked to make sure its format complies with Gnits standards.
- If VERSION indicates that this is an alpha release, and the file 'README-alpha' appears in the topmost directory of a package, then it is included in the distribution. This is done in '--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 'README-alpha' should be included.
- The file 'THANKS' is required.

21 The effect of '--cygnus'

Some packages, notably GNU GCC and GNU gdb, 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 'Makefile.in' is to be constructed. Passing '--cygnus' to automake will cause any generated 'Makefile.in' to comply with Cygnus rules.

Here are the precise effects of '--cygnus':

- Info files are always created in the build directory, and not in the source directory.
- '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. This assumption is an artifact of how Cygnus packages are typically bundled.
- 'make dist' is not supported, and the rules for it are not generated. Cygnus-style trees use their own distribution mechanism.
- Certain tools will be searched for in the build tree as well as in the user's PATH. These tools are runtest, expect, makeinfo and texi2dvi.
- '--foreign' is implied.
- The options 'no-installinfo' and 'no-dependencies' are implied.
- The macros AM_MAINTAINER_MODE and AM_CYGWIN32 are required.
- The check target doesn't depend on all.

GNU maintainers are advised to use 'gnu' strictness in preference to the special Cygnus mode. Some day, perhaps, the differences between Cygnus trees and GNU trees will disappear (for instance, as GCC is made more standards compliant). At that time the special Cygnus mode will be removed.

22 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 'Makefile's.

22.1 Extending Automake Rules

With some minor exceptions (like `_PROGRAMS` variables being rewritten to append '`$(EXEEXT)`'), the contents of a '`Makefile.am`' is copied to '`Makefile.in`' verbatim.

These copying semantics means 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 have the last word. User defined variables in '`Makefile.am`' have priority over variables `AC_SUBST`ed from '`configure.ac`', and `AC_SUBST`ed variables have priority over `automake`-defined variables. As far 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 4 [Invoking Automake], page 10) comes handy to catch overridden definitions.

Note that Automake does not make any difference 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-exec`, `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 12 [Clean], page 70).

```
clean-local:
        -rm -rf testSubDir
```

Older version of this manual used to show how to use `install-data-local` to install a file to some hard-coded location, but you should avoid this. (see Section 26.9 [Hard-Coded Install Paths], page 104)

Some rule also have a way to run another rule, called a *hook*, after their work is done. The hook is named after the principal target, with '`-hook`' appended. The targets allowing hooks are `install-data`, `install-exec`, `uninstall`, `dist`, and `distcheck`.

For instance, here is how to create a hard link to an installed program:

```
install-exec-hook:
        ln $(DESTDIR)$(bindir)/program$(EXEEXT) \
           $(DESTDIR)$(bindir)/proglink$(EXEEXT)
```

Although cheaper and more portable than symbolic links, hard links will not work everywhere (for instance, OS/2 does not have `ln`). Ideally you should fall back to '`cp -p`' when `ln` does not work. An easy way, if symbolic links are acceptable to you, is to add `AC_PROG_LN_S` to '`configure.ac`' (see section "Particular Program Checks" in *The Autoconf Manual*) and use '`$(LN_S)`' in '`Makefile.am`'.

For instance, here is how you could install a versioned copy of a program using '`$(LN_S)`':

```
install-exec-hook:
        cd $(DESTDIR)$(bindir) && \
          mv -f prog$(EXEEXT) prog-$(VERSION)$(EXEEXT) && \
          $(LN_S) prog-$(VERSION)$(EXEEXT) prog$(EXEEXT)
```

Note that we rename the program so that a new version will erase the symbolic link, not the real binary. Also we `cd` into the destination directory in order to create relative links.

When writing `install-exec-hook` or `install-data-hook`, please bear in mind that the exec/data distinction is based on the installation directory, not on the primary used (see Chapter 11 [Install], page 68). So a `foo_SCRIPTS` will be installed by `install-data`, and a `barexec_SCRIPTS` will be installed by `install-exec`. You should define your hooks consequently.

22.2 Third-Party '`Makefile`'s

In most projects all '`Makefile`'s are generated by Automake. In some cases, however, projects need to embed subdirectories with handwritten '`Makefile`'s. 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 `SUBDIRS` or `DIST_SUBDIRS` provided each of these directories has a '`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 '`Makefile`'s support them.

`all`
: Compile the entire package. This is the default target in Automake-generated '`Makefile`'s, but it does not need to be the default in third-party '`Makefile`'s.

`distdir`
: Copy files to distribute into '`$(distdir)`', before a tarball is constructed. Of course this target is not required if the '`no-dist`' option (see Chapter 16 [Options], page 76) is used.

  The variables '`$(top_distdir)`' and '`$(distdir)`' (see Chapter 13 [Dist], page 70) 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 11 [Install], page 68).

`install-info`
: Install only the Texinfo documentation (see Section 10.1 [Texinfo], page 65).

`installdirs`
: Create install directories, but do not install any files.

`check`
`installcheck`
: Check the package (see Chapter 14 [Tests], page 73).

`mostlyclean`
`clean`
`distclean`
`maintainer-clean`
: Cleaning rules (see Chapter 12 [Clean], page 70).

`dvi`
`pdf`
`ps`
`info`
`html`
: Build the documentation in various formats (see Section 10.1 [Texinfo], page 65).

`tags`
`ctags`
: Build 'TAGS' and 'CTAGS' (see Section 17.1 [Tags], page 79).

If you have ever used Gettext in a project, this is a good example of how third-party 'Makefile's can be used with Automake. The 'Makefile's gettextize puts in the 'po/' and 'intl/' directories are handwritten 'Makefile's that implement all 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 6.2 [Conditional Subdirectories], page 25).

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
.PHONY: $(EMPTY_AUTOMAKE_TARGETS)
$(EMPTY_AUTOMAKE_TARGETS):
```

Another aspect of integrating third-party build systems is whether they support VPATH builds. 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 'Makefile's 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: '@scrdir@', '@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 include 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:

```
# 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 22.1 [Extending], page 84) 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':

```
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 sounds very sensible if you need VPATH builds but the subproject does not support them.

23 Distributing 'Makefile.in's

Automake places no restrictions on the distribution of the resulting 'Makefile.in's. 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.

24 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 sometime 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 a undocumented feature, contact `automake@gnu.org` and try to get it documented and exercised by the test-suite.

25 Upgrading a Package to a Newer Automake Version

Automake maintains three kind of files in a package.

- '`aclocal.m4`'
- '`Makefile.in`'s
- 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.in`'s 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 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 4 [Invoking Automake], page 10).

It is important to regenerate all 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.

26 Frequently Asked Questions about Automake

This chapter covers some questions that often come up on the mailing lists.

26.1 CVS and generated files

26.1.1 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 '`Makefile`'s 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.

26.1.2 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 alphabetical order, '`configure.ac`' will appear older 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`'.

26.1.3 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`'s 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 26.2 [maintainer-mode], page 92). `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 version 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 these rebuild rules by default. This is further discussed in Section 26.2 [maintainer-mode], page 92.
- 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`'. He then decide 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`' make the copy of '`Makefile.in`' out-of-date. Since CVS processes files alphabetically, when another developer '`cvs update`' his or her tree, '`Makefile.in`' will happen to be newer than '`Makefile.am`'. This other developer will not see '`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.

26.1.4 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*.

26.2 `missing` and `AM_MAINTAINER_MODE`

26.2.1 `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 CVS, this feature of `missing` allows user *with no maintainer tools* to build a package off CVS, bypassing any timestamp inconsistency implied by '`cvs update`'.

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.

26.2.2 `AM_MAINTAINER_MODE`

`AM_MAINTAINER_MODE` disables the so called "rebuild rules" by default. If you have `AM_MAINTAINER_MODE` in '`configure.ac`', and run '`./configure && make`', then `make` will *never* attempt to rebuilt '`configure`', '`Makefile.in`'s, 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.

If you run '`./configure --enable-maintainer-mode`', then these rebuild rules will be active.

People use `AM_MAINTAINER_MODE` either because they do want their users (or themselves) annoyed by timestamps lossage (see Section 26.1 [CVS], page 90), 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: change to sources 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 CVS, and because `missing` isn't enough if you have the wrong version of the tools.

26.3 Why doesn't Automake support wildcards?

Developers are lazy. They often would like to use wildcards in '`Makefile.am`'s, so they don't need to remember they have to update '`Makefile.am`'s 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 easy to distribute files by mistake. For instance, some code a developer is experimenting with (a test case, say) but 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 at many places, but forget to commit it. Another developer then checkout the incomplete project and is able to run 'make dist' successfully, even though a file is missing.
- Listing files, you control *exactly* what you distribute. If some file that should be distributed is missing from your tree, '`make dist`' will complain. Besides, you don't distribute more than what you listed.
- Finally it's really hard to '`forget`' adding a file to '`Makefile.am`', because if you don't add it, it doesn't get compiled nor installed, so you can't even test it.

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. Resulting '`Makefile.in`'s would be portable since they would list all files and not use '`$(wildcard ...)`'. However that means developers need to remember they must run `automake` each time they add, delete, or rename files.

Compared to editing 'Makefile.am', this is really little win. 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 add support for this syntax. Some people use scripts to generated file lists in 'Makefile.am' or in separate 'Makefile' fragments.

Even if you don't care about portability, and are tempted to use '$(wildcard ...)' anyway because you target only GNU Make, you should know there are many places where Automake need to know exactly which files should be processed. As Automake doesn't know how to expand '$(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.

26.4 Files left in build directory after distclean

This is a diagnostic you might encounter while running 'make distcheck'.

As explained in Chapter 13 [Dist], page 70, 'make distcheck' attempts to build and check your package for errors like this one.

'make distcheck' will perform a VPATH build of your package, 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 12 [Clean], page 70), 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 to force want 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:
./foo.1
```

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 installing help2man, however since this our 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 will all their dependencies.
- If a file is *intended* be rebuilt by users, there is no point in distributing it.

For desperate cases, it's always possible to disable this check by setting distcleancheck_listfiles as documented in Chapter 13 [Dist], page 70. 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.

26.5 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.

26.5.1 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`, `OBJCFLAGS`, `RFLAGS`, 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 2.5 [User Variables], page 4), `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 7.4 [Program and Library Variables], page 40).

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,

```
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 '`Makefile`'s.

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 -Woverride`' 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 what 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 '`Makefile`'s.

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`'.

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 '`prog1_CFLAGS = $(AM_CFLAGS) -no-flag`'. If all 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.

26.5.2 Other Variables

There are other variables in Automake that follow similar principles to allow user options. For instance, Texinfo rules (see Section 10.1 [Texinfo], page 65) use MAKEINFOFLAGS and AM_MAKEINFOFLAGS. Similarly, DejaGnu tests (see Chapter 14 [Tests], page 73) use RUNTESTDEFAULTFLAGS and AM_RUNTESTDEFAULTFLAGS. The tags and ctags rules (see Section 17.1 [Tags], page 79) use ETAGSFLAGS, AM_ETAGSFLAGS, CTAGSFLAGS, and AM_CTAGSFLAGS. Java rules (see Section 9.4 [Java], page 63) use JAVACFLAGS and AM_JAVACFLAGS. None of these rules do support per-target flags (yet).

To some extent, even AM_MAKEFLAGS (see Section 6.1 [Subdirectories], page 25) 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 Chapter 13 [Dist], page 70), ACLOCAL_AMFLAGS (see Chapter 15 [Rebuilding], page 75 and Section 5.8 [Local Macros], page 23), are two variables that are only useful to the maintainer and have no user counterpart.

ARFLAGS (see Section 7.2 [A Library], page 34) is usually defined by Automake and has neither AM_ nor per-target cousin.

Finally you should not think either that the existence of a per-target variable implies that of an AM_ variable or that of a user variable. For instance, the mumble_LDADD per-target variable overrides the global LDADD variable (which is not a user variable), and mumble_LIBADD exists only as a per-target variable. See Section 7.4 [Program and Library Variables], page 40.

26.6 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.

```
bin_PROGRAMS = true false
true_SOURCES = generic.c
true_CPPFLAGS = -DEXIT_CODE=0
false_SOURCES = generic.c
false_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 are 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 7.4 [Program and Library Variables], page 40).

26.7 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 7.4 [Program and Library Variables], page 40 and Section 26.5 [Flag Variables Ordering], page 96). With this you can define compilation flags that apply to all files compiled for a target. For instance, in

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c foo.h bar.c bar.h main.c
foo_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 26.6 [renamed objects], page 98.) 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 require? 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
foo_SOURCES = bar.c bar.h main.c
foo_CFLAGS = -some -flags
foo_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 7.3.5 [Libtool Convenience Libraries], page 37).

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 26.6 [renamed objects], page 98). 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.

26.8 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 '`Makefile`'s.

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:

```
# 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:

```
# 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
```

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; \
        fi
```

However there are now two minor problems in this setup. One is related to the timestamp ordering of 'data.h', 'data.w', 'data.x', and 'data.c'. The other one is a race condition if a parallel make attempts to run multiple instances of the recover block at once.

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.

```
data.c: data.foo data.bar
        foo data.foo data.bar
data.h data.w data.x: data.c
        @if test -f $@; then \
          touch $@; \
        else \
## 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.

```
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 '`$(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; \
          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; exit $$?; \
          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
## Recover from the removal of $@
        @if test -f $@; then :; else \
          trap 'rm -rf elc-lock elc-stamp' 1 2 13 15; \
          if mkdir elc-lock 2>/dev/null; then \
## This code is being executed by the first process.
            rm -f elc-stamp; \
            $(MAKE) $(AM_MAKEFLAGS) elc-stamp; \
            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; do sleep 1; done; \
## Succeed if and only if the first process succeeded.
            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`.

26.9 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.

When a (non-root) user wants to install a package, he usually has no right to install anything in '`/usr`' or '`/usr/local`'. So he does something like '`./configure --prefix ~/usr`' to install package in his 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 a 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 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 shared library that is a Python extension module. If you ask Python where to install the library, it will answer something like this:

```
% python -c 'from distutils import sysconfig;
             print sysconfig.get_python_lib(1,0)'
/usr/lib/python2.3/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.

```
% python -c 'from distutils import sysconfig;
             print sysconfig.get_python_lib(1,0,"${exec_prefix}")'
${exec_prefix}/lib/python2.3/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 9.5 [Python], page 64).

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 of the installation directory that must be preserved. For instance, here is how `AM_PATH_LISPDIR` (see Section 9.1 [Emacs Lisp], page 62) computes '`$(lispdir)`':

```
$EMACS -batch -q -eval '(while load-path
  (princ (concat (car load-path) "\n"))
  (setq load-path (cdr load-path)))' >conftest.out
lispdir=`sed -n
  -e 's,/$,,'
  -e '/.*\/lib\/x*emacs\/site-lisp$/{
        s,.*/lib/\(x*emacs/site-lisp\)$,${libdir}/\1,;p;q;
      }'
  -e '/.*\/share\/x*emacs\/site-lisp$/{
        s,.*/share/\(x*emacs/site-lisp\),${datadir}/\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 expect two possible layouts, otherwise it's easy, and the benefit for non-root users are really worth the extra `sed` invocation.

27 History of Automake

This chapter presents various aspects of the history of Automake. The exhausted reader can safely skip it; this will be more of interest to nostalgic people, or to those curious to learn about the evolution of Automake.

27.1 Timeline

1994-09-19 First CVS commit.

If we can trust the CVS repository, David J. MacKenzie (djm) started working on Automake (or AutoMake, as it was spelt then) this Monday.

The first version of the `automake` script looks as follows.

```
#!/bin/sh

status=0

for makefile
do
  if test ! -f ${makefile}.am; then
    echo "automake: ${makefile}.am: No such honkin' file"
    status=1
    continue
  fi

  exec 4> ${makefile}.in

done
```

From this you can already see that Automake will be about reading '`*.am`' file and producing '`*.in`' files. You cannot see anything else, but if you also know that David is the one who created Autoconf two years before you can guess the rest.

Several commits follow, and by the end of the day Automake is reported to work for GNU fileutils and GNU m4.

The modus operandi is the one that is still used today: variables assignments in '`Makefile.am`' files trigger injections of precanned '`Makefile`' fragments into the generated '`Makefile.in`'. The use of '`Makefile`' fragments was inspired by the 4.4BSD `make` and include files, however Automake aims to be portable and to conform to the GNU standards for '`Makefile`' variables and targets.

At this point, the most recent release of Autoconf is version 1.11, and David is preparing to release Autoconf 2.0 in late October. As a matter of fact, he will barely touch Automake after September.

1994-11-05 David MacKenzie's last commit.

At this point Automake is a 200 line portable shell script, plus 332 lines of '`Makefile`' fragments. In the '`README`', David states his ambivalence between "portable shell" and "more appropriate language":

> I wrote it keeping in mind the possibility of it becoming an Autoconf macro, so it would run at configure-time. That would slow configuration down a bit, but allow users to modify the Makefile.am without needing to fetch the AutoMake package. And, the Makefile.in files wouldn't need to be distributed. But all of AutoMake would. So I might reimplement AutoMake in Perl, m4, or some other more appropriate language.

Automake is described as “an experimental Makefile generator”. There is no documentation. Adventurous users are referred to the examples and patches needed to use Automake with GNU m4 1.3, fileutils 3.9, time 1.6, and development versions of find and indent.

These examples seem to have been lost. However at the time of writing (10 years later in September, 2004) the FSF still distributes a package that uses this version of Automake: check out GNU termutils 2.0.

1995-11-12 Tom Tromey’s first commit.

After one year of inactivity, Tom Tromey takes over the package. Tom was working on GNU cpio back then, and doing this just for fun, having trouble finding a project to contribute to. So while hacking he wanted to bring the ‘`Makefile.in`’ up to GNU standards. This was hard, and one day he saw Automake on `ftp://alpha.gnu.org/`, grabbed it and tried it out.

Tom didn’t talk to djm about it until later, just to make sure he didn’t mind if he made a release. He did a bunch of early releases to the Gnits folks.

Gnits was (and still is) totally informal, just a few GNU friends who François Pinard knew, who were all interested in making a common infrastructure for GNU projects, and shared a similar outlook on how to do it. So they were able to make some progress. It came along with Autoconf and extensions thereof, and then Automake from David and Tom (who were both gnitsians). One of their ideas was to write a document paralleling the GNU standards, that was more strict in some ways and more detailed. They never finished the GNITS standards, but the ideas mostly made their way into Automake.

1995-11-23 Automake 0.20

Besides introducing automatic dependency tracking (see Section 27.2 [Dependency Tracking Evolution], page 117), this version also supplies a 9-page manual.

At this time `aclocal` and `AM_INIT_AUTOMAKE` did not exist, so many things had to be done by hand. For instance, here is what a configure.in (this is the former name of the ‘`configure.ac`’ we use today) must contain in order to use Automake 0.20:

```
PACKAGE=cpio
VERSION=2.3.911
AC_DEFINE_UNQUOTED(PACKAGE, "$PACKAGE")
AC_DEFINE_UNQUOTED(VERSION, "$VERSION")
AC_SUBST(PACKAGE)
AC_SUBST(VERSION)
AC_ARG_PROGRAM
AC_PROG_INSTALL
```

(Today all of the above is achieved by `AC_INIT` and `AM_INIT_AUTOMAKE`.)

Here is how programs are specified in ‘`Makefile.am`’:

```
PROGRAMS = hello
hello_SOURCES = hello.c
```

This looks pretty much like what we do today, except the `PROGRAMS` variable has no directory prefix specifying where ‘`hello`’ should be installed: all programs are installed in '$(bindir)'. LIBPROGRAMS can be used to specify programs that must be built but not installed (it is called noinst_PROGRAMS nowadays).

Programs can be built conditionally using AC_SUBSTitutions:

```
PROGRAMS = @progs@
AM_PROGRAMS = foo bar baz
```

(AM_PROGRAMS has since then been renamed to EXTRA_PROGRAMS.)

Similarly scripts, static libraries, and data can built and installed using the LIBRARIES, SCRIPTS, and DATA variables. However LIBRARIES were treated a bit specially in that Automake did automatically supply the 'lib' and '.a' prefixes. Therefore to build 'libcpio.a', one had to write

```
LIBRARIES = cpio
cpio_SOURCES = ...
```

Extra files to distribute must be listed in DIST_OTHER (the ancestor of EXTRA_DIST). Also extra directories that are to be distributed should appear in DIST_SUBDIRS, but the manual describes this as a temporary ugly hack (today extra directories should also be listed in EXTRA_DIST, and DIST_SUBDIRS is used for another purpose, see Section 6.2 [Conditional Subdirectories], page 25).

1995-11-26 Automake 0.21

In less time that it takes to cook a frozen pizza, Tom rewrites Automake using Perl. At this time Perl 5 is only one year old, and Perl 4.036 is in use at many sites. Supporting several Perl versions has been a source of problems through the whole history of Automake.

If you never used Perl 4, imagine Perl 5 without objects, without 'my' variables (only dynamically scoped 'local' variables), without function prototypes, with function calls that needs to be prefixed with '&', etc. Traces of this old style can still be found in today's automake.

1995-11-28 Automake 0.22
1995-11-29 Automake 0.23

Bug fixes.

1995-12-08 Automake 0.24
1995-12-10 Automake 0.25

Releases are raining. 0.24 introduces the uniform naming scheme we use today, i.e., bin_PROGRAMS instead of PROGRAMS, noinst_LIBRARIES instead of LIBLIBRARIES, etc. (However EXTRA_PROGRAMS does not exist yet, AM_PROGRAMS is still in use; and TEXINFOS and MANS still have no directory prefixes.) Adding support for prefixes like that was one of the major ideas in automake; it has lasted pretty well.

AutoMake is renamed to Automake (Tom seems to recall it was François Pinard's doing).

0.25 fixes a Perl 4 portability bug.

1995-12-18 Jim Meyering starts using Automake in GNU Textutils.
1995-12-31 François Pinard starts using Automake in GNU tar.
1996-01-03 Automake 0.26
1996-01-03 Automake 0.27

Of the many change and suggestions sent by François Pinard and included in 0.26, the most important is perhaps the advise that to ease customization a user rule or variable definition should always override an Automake rule or definition.

Gordon Matzigkeit and Jim Meyering are two other early contributors that have been sending fixes.

0.27 fixes yet another Perl 4 portability bug.

1996-01-13 Automake 0.28

Automake starts scanning '`configure.in`' for `LIBOBJS` support. This is an important step because until this version Automake did only know about the '`Makefile.am`'s it processed. '`configure.in`' was Autoconf's world and the link between Autoconf and Automake had to be done by the '`Makefile.am`' author. For instance, if '`config.h`' was generated by '`configure`', it was the package maintainer's responsibility to define the `CONFIG_HEADER` variable in each '`Makefile.am`'.

Succeeding releases will rely more and more on scanning '`configure.in`' to better automate the Autoconf integration.

0.28 also introduces the `AUTOMAKE_OPTIONS` variable and the '`--gnu`' and '`--gnits`' options, the latter being stricter.

1996-02-07 Automake 0.29

Thanks to '`configure.in`' scanning, `CONFIG_HEADER` is gone, and rebuild rules for '`configure`'-generated file are automatically output.

`TEXINFOS` and `MANS` converted to the uniform naming scheme.

1996-02-24 Automake 0.30

The test suite is born. It contains 9 tests. From now on test cases will be added pretty regularly (see Section 27.3 [Releases], page 121), and this proved to be really helpful later on.

`EXTRA_PROGRAMS` finally replaces `AM_PROGRAMS`.

All the third-party Autoconf macros, written mostly by François Pinard (and later Jim Meyering), are distributed in Automake's hand-written '`aclocal.m4`' file. Package maintainers are expected to extract the necessary macros from this file. (In previous version you had to copy and paste them from the manual...)

1996-03-11 Automake 0.31

The test suite in 0.30 was run via a long `check-local` rule. Upon Ulrich Drepper's suggestion, 0.31 makes it an Automake rule output whenever the `TESTS` variable is defined.

`DIST_OTHER` is renamed to `EXTRA_DIST`, and the `check_` prefix is introduced. The syntax is now the same as today.

1996-03-15 Gordon Matzigkeit starts writing libtool.

1996-04-27 Automake 0.32

`-hook` targets are introduced; an idea from Dieter Baron.

'`*.info`' files, which were output in the build directory are now built in the source directory, because they are distributed. It seems these files like to move back and forth as that will happen again in future versions.

1996-05-18 Automake 0.33

Gord Matzigkeit's main two contributions:

- very preliminary libtool support
- the distcheck rule

Although they were very basic at this point, these are probably among the top features for Automake today.

Jim Meyering also provides the infamous `jm_MAINTAINER_MODE`, since then renamed to `AM_MAINTAINER_MODE` and abandoned by its author (see Section 26.2 [maintainer-mode], page 92).

1996-05-28 Automake 1.0

After only six months of heavy development, the automake script is 3134 lines long, plus 973 lines of '`Makefile`' fragments. The package has 30 pages of documentation, and 38 test cases. '`aclocal.m4`' contains 4 macros.

From now on and until version 1.4, new releases will occur at a rate of about one a year. 1.1 did not exist, actually 1.1b to 1.1p have been the name of beta releases for 1.2. This is the first time Automake uses suffix letters to designate beta releases, an habit that lasts.

1996-10-10 Kevin Dalley packages Automake 1.0 for Debian GNU/Linux.

1996-11-26 David J. MacKenzie releases Autoconf 2.12.

Between June and October, the Autoconf development is almost staled. Roland McGrath has been working at the beginning of the year. David comes back in November to release 2.12, but he won't touch Autoconf anymore after this year, and Autoconf then really stagnates. The desolate Autoconf '`ChangeLog`' for 1997 lists only 7 commits.

1997-02-28 `automake@gnu.ai.mit.edu` list alive

The mailing list is announced as follows:

```
I've created the "automake" mailing list.  It is
"automake@gnu.ai.mit.edu".  Administrivia, as always, to
automake-request@gnu.ai.mit.edu.

The charter of this list is discussion of automake, autoconf, and
other configuration/portability tools (eg libtool).  It is expected
that discussion will range from pleas for help all the way up to
patches.

This list is archived on the FSF machines.  Offhand I don't know if
you can get the archive without an account there.

This list is open to anybody who wants to join.  Tell all your
friends!
-- Tom Tromey
```

Before that people were discussing Automake privately, on the Gnits mailing list (which is not public either), and less frequently on `gnu.misc.discuss`.

`gnu.ai.mit.edu` is now `gnu.org`, in case you never noticed. The archives of the early years of the `automake@gnu.org` list have been lost, so today it is almost impossible to find traces of discussions that occurred before 1999. This has been annoying more than once, as such discussions can be useful to understand the rationale behind a piece of uncommented code that was introduced back then.

1997-06-22 Automake 1.2

Automake developments continues, and more and more new Autoconf macros are required. Distributing them in '`aclocal.m4`' and requiring people to browse this file to extract the relevant macros becomes uncomfortable. Ideally, some of them should be contributed to Autoconf so that they can be used directly, however Autoconf is currently inactive. Automake 1.2 consequently introduces `aclocal` (`aclocal` was actually started on 1996-07-28), a tool that automatically constructs an '`aclocal.m4`' file from a repository of third-party macros. Because Autoconf has stalled, Automake also becomes a kind of repository for such third-party macros, even macros completely unrelated to Automake (for instance macros that fix broken Autoconf macros).

The 1.2 release contains 20 macros, among which the `AM_INIT_AUTOMAKE` macro that simplifies the creation of '`configure.in`'.

Libtool is fully supported using `*_LTLIBRARIES`.

The missing script is introduced by François Pinard; it is meant to be a better solution than `AM_MAINTAINER_MODE` (see Section 26.2 [maintainer-mode], page 92).

Conditionals support was implemented by Ian Lance Taylor. At the time, Tom and Ian were working on an internal project at Cygnus. They were using ILU, which is pretty similar to CORBA. They wanted to integrate ILU into their build, which was all '`configure`'-based, and Ian thought that adding conditionals to `automake` was simpler than doing all the work in '`configure`' (which was the standard at the time). So this was actually funded by Cygnus.

This very useful but tricky feature will take a lot of time to stabilize. (At the time this text is written, there are still primaries that have not been updated to support conditional definitions in Automake 1.9.)

The `automake` script has almost doubled: 6089 lines of Perl, plus 1294 lines of '`Makefile`' fragments.

1997-07-08 Gordon Matzigkeit releases Libtool 1.0.

1998-04-05 Automake 1.3

This is a small advance compared to 1.2. It add support for assembly, and preliminary support for Java.

Perl 5.004_04 is out, but fixes to support Perl 4 are still regularly submitted whenever Automake breaks it.

1998-09-06 `sourceware.cygnus.com` is on-line.

Sourceware was setup by Jason Molenda to host open source projects.

1998-09-19 Automake CVS repository moved to `sourceware.cygnus.com`

1998-10-26 `sourceware.cygnus.com` announces it hosts Automake

Automake is now hosted on `sourceware.cygnus.com`. It has a publicly accessible CVS repository. This CVS repository is a copy of the one Tom was using on his machine, which in turn is based on a copy of the CVS repository of David MacKenzie. This is why we still have to full source history. (Automake is still on Sourceware today, but the host has been renamed to `sources.redhat.com`.)

The oldest file in the administrative directory of the CVS repository that was created on Sourceware is dated 1998-09-19, while the announcement that `automake` and `autoconf` had joined `sourceware` was made on 1998-10-26. They were among the first projects to be hosted there.

The heedful reader will have noticed Automake was exactly 4-year-old on 1998-09-19.

1999-01-05 Ben Elliston releases Autoconf 2.13.

1999-01-14 Automake 1.4

This release adds support for Fortran 77 and for the `include` statement. Also, '`+=`' assignments are introduced, but it is still quite easy to fool Automake when mixing this with conditionals.

These two releases, Automake 1.4 and Autoconf 2.13 makes a duo that will be used together for years.

`automake` is 7228 lines, plus 1591 lines of Makefile fragment, 20 macros (some 1.3 macros were finally contributed back to Autoconf), 197 test cases, and 51 pages of documentation.

1999-03-27 The `user-dep-branch` is created on the CVS repository.

This implements a new dependency tracking schemed that should be able to handle automatic dependency tracking using any compiler (not just gcc) and any make (not just GNU `make`). In addition, the new scheme should be more reliable than the old one, as dependencies are generated on the end user's machine. Alexandre Oliva creates depcomp for this purpose.

See Section 27.2 [Dependency Tracking Evolution], page 117, for more details about the evolution of automatic dependency tracking in Automake.

1999-11-21 The `user-dep-branch` is merged into the main trunk.

This was a huge problem since we also had patches going in on the trunk. The merge took a long time and was very painful.

2000-05-10

Since September 1999 and until 2003, Akim Demaille will be zealously revamping Autoconf.

> I think the next release should be called "3.0".
> Let's face it: you've basically rewritten autoconf.
> Every weekend there are 30 new patches.
> I don't see how we could call this "2.15" with a straight face.
> – Tom Tromey on `autoconf@gnu.org`

Actually Akim works like a submarine: he will pile up patches while he works off-line during the weekend, and flush them in batch when he resurfaces on Monday.

2001-01-24

On this Wednesday, Autoconf 2.49c, the last beta before Autoconf 2.50 is out, and Akim has to find something to do during his week-end :)

2001-01-28

Akim sends a batch of 14 patches to automake@gnu.org.

> Aiieeee! I was dreading the day that the Demaillator turned his sights on automake… and now it has arrived! – Tom Tromey

It's only the beginning: in two months he will send 192 patches. Then he would slow down so Tom can catch up and review all this. Initially Tom actually read all these patches, then he probably trustingly answered OK to most of them, and finally gave up and let Akim apply whatever he wanted. There was no way to keep up with that patch rate.

> Anyway the patch below won't apply since it predates Akim's sourcequake; I have yet to figure where the relevant passage has been moved :) – Alexandre Duret-Lutz

All these patches were sent to and discussed on automake@gnu.org, so subscribed users were literally drown in technical mails. Eventually, the automake-patches@gnu.org mailing list was created in May.

Year after year, Automake had drifted away from its initial design: construct '`Makefile.in`' by assembling various '`Makefile`' fragments. In 1.4, lots of '`Makefile`' rules are being emitted at various places in the `automake` script itself; this does not help ensuring a consistent treatment of these rules (for instance making sure that user-defined rules override Automake's own rules). One of Akim's goal was moving all these hard-coded rules to separate '`Makefile`' fragments, so the logic could be centralized in a '`Makefile`' fragment processor.

Another significant contribution of Akim is the interface with the "trace" feature of Autoconf. The way to scan '`configure.in`' at this time was to read the file and grep the various macro of interest to Automake. Doing so could break in many unexpected ways; automake could miss some definition (for instance '`AC_SUBST([$1], [$2])`' where the arguments are known only when M4 is run), or conversely it could detect some macro that was not expanded (because it is called conditionally). In the CVS version of Autoconf, Akim had implemented the '`--trace`' option, which provides accurate information about where macros are actually called and with what arguments. Akim will equip Automake with a second '`configure.in`' scanner that uses this '`--trace`' interface. Since it was not sensible to drop the Autoconf 2.13 compatibility yet, this experimental scanner was only used when an environment variable was set, the traditional grep-scanner being still the default.

2001-04-25 Gary V. Vaughan releases Libtool 1.4

It has been more than two years since Automake 1.4, CVS Automake has suffered lot's of heavy changes and still is not ready for release. Libtool 1.4 had to be distributed with a patch against Automake 1.4.

2001-05-08 Automake 1.4-p1
2001-05-24 Automake 1.4-p2

Gary V. Vaughan, the principal Libtool maintainer, makes a "patch release" of Automake:

> The main purpose of this release is to have a stable automake which is compatible with the latest stable libtool.

The release also contains obvious fixes for bugs in Automake 1.4, some of which were reported almost monthly.

2001-05-21 Akim Demaille releases Autoconf 2.50
2001-06-07 Automake 1.4-p3
2001-06-10 Automake 1.4-p4
2001-07-15 Automake 1.4-p5

Gary continues his patch-release series. These also add support for some new Autoconf 2.50 idioms. Essentially, Autoconf now advocates '`configure.ac`' over '`configure.in`', and it introduces a new syntax for `AC_OUTPUT`ing files.

2001-08-23 Automake 1.5

A major and long-awaited release, that comes more than two years after 1.4. It brings many changes, among which:

- The new dependency tracking scheme that uses `depcomp`. Aside from the improvement on the dependency tracking itself (see Section 27.2 [Dependency Tracking Evolution], page 117), this also streamlines the use of automake generated '`Makefile.in`'s as the '`Makefile.in`'s used during development are now the same as those used in distributions. Before that the '`Makefile.in`'s generated for maintainers required GNU `make` and GCC, they were different from the portable '`Makefile`' generated for distribution; this was causing some confusion.
- Support for per-target compilation flags.
- Support for reference to files in subdirectories in most '`Makefile.am`' variables.
- Introduction of the `dist_`, `nodist_`, and `nobase_` prefixes.
- Perl 4 support is finally dropped.

1.5 did broke several packages that worked with 1.4. Enough so that Linux distributions could not easily install the new Automake version without breaking many of the packages for which they had to run `automake`.

Some of these breakages were effectively bugs that would eventually be fixed in the next release. However, a lot of damage was caused by some changes made deliberately to render Automake stricter on some setup we did consider bogus. For instance, '`make distcheck`' was improved to check that '`make uninstall`' did remove all the files '`make install`' installed, that '`make distclean`' did not omit some file, and that a VPATH build would work even if the source directory was read-only. Similarly, Automake now rejects multiple definitions of the same variable (because that would mix very badly with conditionals), and '`+=`' assignments with no previous definition. Because these changes all occurred suddenly after 1.4 had been established for more that two years, it hurt users.

To make matter worse, meanwhile Autoconf (now at version 2.52) was facing similar troubles, for similar reasons.

2002-03-05 Automake 1.6

This release introduced versioned installation (see Chapter 24 [API versioning], page 88). This was mainly pushed by Havoc Pennington, taking the GNOME source tree as motive: due to incompatibilities between the autotools it's impossible for the GNOME packages to switch to Autoconf 2.53 and Automake 1.5 all at once, so they are currently stuck with Autoconf 2.13 and Automake 1.4.

The idea was to call this version '`automake-1.6`', call all its bug-fix versions identically, and switch to '`automake-1.7`' for the next release that adds new features or changes some rules. This scheme implies maintaining a bug-fix branch in addition to the development trunk, which means more work from the maintainer, but providing regular bug-fix releases proved to be really worthwhile.

Like 1.5, 1.6 also introduced a bunch of incompatibilities, meant or not. Perhaps the more annoying was the dependence on the newly released Autoconf 2.53. Autoconf seemed to have stabilized enough since its explosive 2.50 release, and included changes required to fix some bugs in Automake. In order to upgrade to Automake 1.6, people now had to upgrade Autoconf too; for some packages it was no picnic.

While versioned installation helped people to upgrade, it also unfortunately allowed people not to upgrade. At the time of writing, some Linux distributions are shipping packages for Automake 1.4, 1.5, 1.6, 1.7, 1.8, and 1.9. Most of these still install 1.4 by default. Some distribution also call 1.4 the "stable" version, and present "1.9" as the development version; this does not really makes sense since 1.9 is way more solid than 1.4. All this does not help the newcomer.

2002-04-11 Automake 1.6.1

1.6, and the upcoming 1.4-p6 release were the last release by Tom. This one and those following will be handled by Alexandre Duret-Lutz. Tom is still around, and will be there until about 1.7, but his interest into Automake is drifting away towards projects like `gcj`.

Alexandre has been using Automake since 2000, and started to contribute mostly on Akim's incitement (Akim and Alexandre have been working in the same room from 1999 to 2002). In 2001 and 2002 he had a lot of free time to enjoy hacking Automake.

2002-06-14 Automake 1.6.2
2002-07-28 Automake 1.6.3
2002-07-28 Automake 1.4-p6

Two releases on the same day. 1.6.3 is a bug-fix release.

Tom Tromey backported the versioned installation mechanism on the 1.4 branch, so that Automake 1.6.x and Automake 1.4-p6 could be installed side by side. Another request from the GNOME folks.

2002-09-25 Automake 1.7
This release switches to the new 'configure.ac' scanner Akim was experimenting in 1.5.

2002-10-16 Automake 1.7.1
2002-12-06 Automake 1.7.2
2003-02-20 Automake 1.7.3
2003-04-23 Automake 1.7.4
2003-05-18 Automake 1.7.5
2003-07-10 Automake 1.7.6
2003-09-07 Automake 1.7.7
2003-10-07 Automake 1.7.8
Many bug-fix releases. 1.7 lasted because the development version (upcoming 1.8) was suffering some major internal revamping.

2003-10-26 Automake on screen
Episode 49, 'Repercussions', in the third season of the 'Alias' TV show is first aired.

Marshall, one of the character, is working on a computer virus that he has to modify before it gets into the wrong hands or something like that. The screenshots you see do not show any program code, they show a 'Makefile.in' generated by automake...

2003-11-09 Automake 1.7.9
2003-12-10 Automake 1.8
The most striking update is probably that of aclocal.

aclocal now uses m4_include in the produced aclocal.m4 when the included macros are already distributed with the package (an idiom used in many packages), which reduces code duplication. Many people liked that, but in fact this change was really introduced to fix a bug in rebuild rules: 'Makefile.in' must be rebuilt whenever a dependency of 'configure' changes, but all the 'm4' files included in 'aclocal.m4' where unknown from automake. Now automake can just trace the m4_includes to discover the dependencies.

aclocal also starts using the '--trace' Autoconf option in order to discover used macros more accurately. This will turn out to be very tricky (later releases will improve this) as people had devised many ways to cope with the limitation of previous aclocal versions, notably using handwritten m4_includes: aclocal must make sure not to redefine a rule that is already included by such statement.

Automake also has seen its guts rewritten. Although this rewriting took a lot of efforts, it is only apparent to the users in that some constructions previously disallowed by the implementation now work nicely. Conditionals, Locations, Variable and Rule definitions, Options: these items on which Automake works have been rewritten as separate Perl modules, and documented.

2004-01-11 Automake 1.8.1

2004-01-12 Automake 1.8.2

2004-03-07 Automake 1.8.3

2004-04-25 Automake 1.8.4

2004-05-16 Automake 1.8.5

2004-07-28 Automake 1.9

This release tries to simplify the compilation rules it outputs to reduce the size of the Makefile. The complaint initially come from the libgcj developers. Their '`Makefile.in`' generated with Automake 1.4 and custom build rules (1.4 did not support compiled Java) is 250KB. The one generated by 1.8 was over 9MB! 1.9 gets it down to 1.2MB.

Aside from this it contains mainly minor changes and bug-fixes.

2004-08-11 Automake 1.9.1

2004-09-19 Automake 1.9.2

Automake has ten years. This chapter of the manual was initially written for this occasion.

27.2 Dependency Tracking in Automake

Over the years Automake has deployed three different dependency tracking methods. Each method, including the current one, has had flaws of various sorts. Here we lay out the different dependency tracking methods, their flaws, and their fixes. We conclude with recommendations for tool writers, and by indicating future directions for dependency tracking work in Automake.

27.2.1 First Take

Description

Our first attempt at automatic dependency tracking was based on the method recommended by GNU `make`. (see section "Generating Prerequisites Automatically" in *The GNU make Manual*)

This version worked by precomputing dependencies ahead of time. For each source file, it had a special '`.P`' file that held the dependencies. There was a rule to generate a '`.P`' file by invoking the compiler appropriately. All such '`.P`' files were included by the '`Makefile`', thus implicitly becoming dependencies of '`Makefile`'.

Bugs

This approach had several critical bugs.

- The code to generate the '`.P`' file relied on `gcc`. (A limitation, not technically a bug.)
- The dependency tracking mechanism itself relied on GNU `make`. (A limitation, not technically a bug.)
- Because each '`.P`' file was a dependency of '`Makefile`', this meant that dependency tracking was done eagerly by `make`. For instance, '`make clean`' would cause all the dependency files to be updated, and then immediately removed. This eagerness also

caused problems with some configurations; if a certain source file could not be compiled on a given architecture for some reason, dependency tracking would fail, aborting the entire build.

- As dependency tracking was done as a pre-pass, compile times were doubled–the compiler had to be run twice per source file.
- '`make dist`' re-ran `automake` to generate a '`Makefile`' that did not have automatic dependency tracking (and that was thus portable to any version of `make`). In order to do this portably, Automake had to scan the dependency files and remove any reference that was to a source file not in the distribution. This process was error-prone. Also, if '`make dist`' was run in an environment where some object file had a dependency on a source file that was only conditionally created, Automake would generate a '`Makefile`' that referred to a file that might not appear in the end user's build. A special, hacky mechanism was required to work around this.

Historical Note

The code generated by Automake is often inspired by the '`Makefile`' style of a particular author. In the case of the first implementation of dependency tracking, I believe the impetus and inspiration was Jim Meyering. (I could be mistaken. If you know otherwise feel free to correct me.)

27.2.2 Dependencies As Side Effects

Description

The next refinement of Automake's automatic dependency tracking scheme was to implement dependencies as side effects of the compilation. This was aimed at solving the most commonly reported problems with the first approach. In particular we were most concerned with eliminating the weird rebuilding effect associated with make clean.

In this approach, the '`.P`' files were included using the `-include` command, which let us create these files lazily. This avoided the '`make clean`' problem.

We only computed dependencies when a file was actually compiled. This avoided the performance penalty associated with scanning each file twice. It also let us avoid the other problems associated with the first, eager, implementation. For instance, dependencies would never be generated for a source file that was not compilable on a given architecture (because it in fact would never be compiled).

Bugs

- This approach also relied on the existence of `gcc` and GNU `make`. (A limitation, not technically a bug.)
- Dependency tracking was still done by the developer, so the problems from the first implementation relating to massaging of dependencies by '`make dist`' were still in effect.
- This implementation suffered from the "deleted header file" problem. Suppose a lazily-created '`.P`' file includes a dependency on a given header file, like this:

  ```
  maude.o: maude.c something.h
  ```

Now suppose that the developer removes ‘`something.h`’ and updates ‘`maude.c`’ so that this include is no longer needed. If he runs `make`, he will get an error because there is no way to create ‘`something.h`’.

We fixed this problem in a later release by further massaging the output of `gcc` to include a dummy dependency for each header file.

27.2.3 Dependencies for the User

Description

The bugs associated with ‘`make dist`’, over time, became a real problem. Packages using Automake were being built on a large number of platforms, and were becoming increasingly complex. Broken dependencies were distributed in “portable” ‘`Makefile.in`’s, leading to user complaints. Also, the requirement for `gcc` and GNU `make` was a constant source of bug reports. The next implementation of dependency tracking aimed to remove these problems.

We realized that the only truly reliable way to automatically track dependencies was to do it when the package itself was built. This meant discovering a method portable to any version of make and any compiler. Also, we wanted to preserve what we saw as the best point of the second implementation: dependency computation as a side effect of compilation.

In the end we found that most modern make implementations support some form of include directive. Also, we wrote a wrapper script that let us abstract away differences between dependency tracking methods for compilers. For instance, some compilers cannot generate dependencies as a side effect of compilation. In this case we simply have the script run the compiler twice. Currently our wrapper script (`depcomp`) knows about twelve different compilers (including a "compiler" that simply invokes `makedepend` and then the real compiler, which is assumed to be a standard Unix-like C compiler with no way to do dependency tracking).

Bugs

- Running a wrapper script for each compilation slows down the build.
- Many users don’t really care about precise dependencies.
- This implementation, like every other automatic dependency tracking scheme in common use today (indeed, every one we’ve ever heard of), suffers from the “duplicated new header” bug.

  This bug occurs because dependency tracking tools, such as the compiler, only generate dependencies on the successful opening of a file, and not on every probe.

  Suppose for instance that the compiler searches three directories for a given header, and that the header is found in the third directory. If the programmer erroneously adds a header file with the same name to the first directory, then a clean rebuild from scratch could fail (suppose the new header file is buggy), whereas an incremental rebuild will succeed.

  What has happened here is that people have a misunderstanding of what a dependency is. Tool writers think a dependency encodes information about which files were read by the compiler. However, a dependency must actually encode information about what the compiler tried to do.

This problem is not serious in practice. Programmers typically do not use the same name for a header file twice in a given project. (At least, not in C or C++. This problem may be more troublesome in Java.) This problem is easy to fix, by modifying dependency generators to record every probe, instead of every successful open.

- Since automake generates dependencies as a side effect of compilation, there is a bootstrapping problem when header files are generated by running a program. The problem is that, the first time the build is done, there is no way by default to know that the headers are required, so make might try to run a compilation for which the headers have not yet been built.

  This was also a problem in the previous dependency tracking implementation.

  The current fix is to use `BUILT_SOURCES` to list built headers (see Section 8.4 [Sources], page 58). This causes them to be built before any other other build rules are run. This is unsatisfactory as a general solution, however in practice it seems sufficient for most actual programs.

This code is used since Automake 1.5.

In GCC 3.0, we managed to convince the maintainers to add special command-line options to help Automake more efficiently do its job. We hoped this would let us avoid the use of a wrapper script when Automake's automatic dependency tracking was used with `gcc`.

Unfortunately, this code doesn't quite do what we want. In particular, it removes the dependency file if the compilation fails; we'd prefer that it instead only touch the file in any way if the compilation succeeds.

Nevertheless, since Automake 1.7, when a recent `gcc` is detected at `configure` time, we inline the dependency-generation code and do not use the `depcomp` wrapper script. This makes compilations faster for those using this compiler (probably our primary user base). The counterpart is that because we have to encode two compilation rules in '`Makefile`' (with or without `depcomp`), the produced '`Makefile`'s are larger.

27.2.4 Techniques for Computing Dependencies

There are actually several ways for a build tool like Automake to cause tools to generate dependencies.

`makedepend`
: This was a commonly-used method in the past. The idea is to run a special program over the source and have it generate dependency information. Traditional implementations of `makedepend` ere not completely precise; ordinarily they were conservative and discovered too many dependencies.

The tool
: An obvious way to generate dependencies is to simply write the tool so that it can generate the information needed by the build tool. This is also the most portable method. Many compilers have an option to generate dependencies. Unfortunately, not all tools provide such an option.

The file system
: It is possible to write a special file system that tracks opens, reads, writes, etc, and then feed this information back to the build tool. `clearmake` does this. This is a very powerful technique, as it doesn't require cooperation from the

tool. Unfortunately it is also very difficult to implement and also not practical in the general case.

`LD_PRELOAD`
: Rather than use the file system, one could write a special library to intercept `open` and other syscalls. This technique is also quite powerful, but unfortunately it is not portable enough for use in `automake`.

27.2.5 Recommendations for Tool Writers

We think that every compilation tool ought to be able to generate dependencies as a side effect of compilation. Furthermore, at least while `make`-based tools are nearly universally in use (at least in the free software community), the tool itself should generate dummy dependencies for header files, to avoid the deleted header file bug. Finally, the tool should generate a dependency for each probe, instead of each successful file open, in order to avoid the duplicated new header bug.

27.2.6 Future Directions for Automake's Dependency Tracking

Currently, only languages and compilers understood by Automake can have dependency tracking enabled. We would like to see if it is practical (and worthwhile) to let this support be extended by the user to languages unknown to Automake.

27.3 Release Statistics

The following table (inspired by '`perlhist(1)`') quantifies the evolution of Automake using these metrics:

Date, Rel	The date and version of the release.
am	The number of lines of the `automake` script.
acl	The number of lines of the `aclocal` script.
pm	The number of lines of the `Perl` supporting modules.
'`*.am`'	The number of lines of the '`Makefile`' fragments. The number in parenthesis is the number of files.
m4	The number of lines (and files) of Autoconf macros.
doc	The number of pages of the documentation (the Postscript version).
t	The number of test cases in the test suite.

Date	**Rel**	**am**	**acl**	**pm**	**'`*.am`'**	**m4**	**doc**	**t**
1994-09-19	CVS	141			299 (24)			
1994-11-05	CVS	208			332 (28)			
1995-11-23	0.20	533			458 (35)		9	
1995-11-26	0.21	613			480 (36)		11	
1995-11-28	0.22	1116			539 (38)		12	
1995-11-29	0.23	1240			541 (38)		12	
1995-12-08	0.24	1462			504 (33)		14	
1995-12-10	0.25	1513			511 (37)		15	

1996-01-03	0.26	1706			438 (36)		16	
1996-01-03	0.27	1706			438 (36)		16	
1996-01-13	0.28	1964			934 (33)		16	
1996-02-07	0.29	2299			936 (33)		17	
1996-02-24	0.30	2544			919 (32)	85 (1)	20	9
1996-03-11	0.31	2877			919 (32)	85 (1)	29	17
1996-04-27	0.32	3058			921 (31)	85 (1)	30	26
1996-05-18	0.33	3110			926 (31)	105 (1)	30	35
1996-05-28	1.0	3134			973 (32)	105 (1)	30	38
1997-06-22	1.2	6089	385		1294 (36)	592 (23)	37	126
1998-04-05	1.3	6415	422		1470 (39)	741 (26)	39	156
1999-01-14	1.4	7240	426		1591 (40)	734 (23)	51	197
2001-05-08	1.4-p1	7251	426		1591 (40)	734 (23)	51	197
2001-05-24	1.4-p2	7268	439		1591 (40)	734 (23)	49	197
2001-06-07	1.4-p3	7312	439		1591 (40)	734 (23)	49	197
2001-06-10	1.4-p4	7321	439		1591 (40)	734 (23)	49	198
2001-07-15	1.4-p5	7228	426		1596 (40)	734 (23)	51	198
2001-08-23	1.5	8016	475	600	2654 (39)	1166 (32)	63	327
2002-03-05	1.6	8465	475	1136	2732 (39)	1603 (31)	66	365
2002-04-11	1.6.1	8544	475	1136	2741 (39)	1603 (31)	66	372
2002-06-14	1.6.2	8575	475	1136	2800 (39)	1609 (31)	67	386
2002-07-28	1.6.3	8600	475	1153	2809 (39)	1609 (31)	67	391
2002-07-28	1.4-p6	7332	455		1596 (40)	735 (24)	49	197
2002-09-25	1.7	9189	471	1790	2965 (39)	1606 (33)	73	430
2002-10-16	1.7.1	9229	475	1790	2977 (39)	1606 (33)	73	437
2002-12-06	1.7.2	9334	475	1790	2988 (39)	1606 (33)	77	445
2003-02-20	1.7.3	9389	475	1790	3023 (39)	1651 (34)	84	448
2003-04-23	1.7.4	9429	475	1790	3031 (39)	1644 (34)	85	458
2003-05-18	1.7.5	9429	475	1790	3033 (39)	1645 (34)	85	459
2003-07-10	1.7.6	9442	475	1790	3033 (39)	1660 (34)	85	461
2003-09-07	1.7.7	9443	475	1790	3041 (39)	1660 (34)	90	467
2003-10-07	1.7.8	9444	475	1790	3041 (39)	1660 (34)	90	468
2003-11-09	1.7.9	9444	475	1790	3048 (39)	1660 (34)	90	468
2003-12-10	1.8	7171	585	7730	3236 (39)	1666 (36)	104	521
2004-01-11	1.8.1	7217	663	7726	3287 (39)	1686 (36)	104	525
2004-01-12	1.8.2	7217	663	7726	3288 (39)	1686 (36)	104	526
2004-03-07	1.8.3	7214	686	7735	3303 (39)	1695 (36)	111	530
2004-04-25	1.8.4	7214	686	7736	3310 (39)	1701 (36)	112	531
2004-05-16	1.8.5	7240	686	7736	3299 (39)	1701 (36)	112	533
2004-07-28	1.9	7508	715	7794	3352 (40)	1812 (37)	115	551
2004-08-11	1.9.1	7512	715	7794	3354 (40)	1812 (37)	115	552
2004-09-19	1.9.2	7512	715	7794	3354 (40)	1812 (37)	132	554

Appendix A Copying This Manual

A.1 GNU Free Documentation License

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.

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

A.1.1 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 1.2
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

_

_AM_DEPENDENCIES . . . 21

A

AC_CANONICAL_BUILD . . . 14
AC_CANONICAL_HOST . . . 14
AC_CANONICAL_TARGET . . . 14
AC_CONFIG_AUX_DIR . . . 13, 29
AC_CONFIG_FILES . . . 13
AC_CONFIG_HEADERS . . . 13
AC_CONFIG_LIBOBJ_DIR . . . 45
AC_CONFIG_LINKS . . . 13
AC_CONFIG_SUBDIRS . . . 29
AC_DEFUN . . . 22
AC_F77_LIBRARY_LDFLAGS . . . 14
AC_INIT . . . 19
AC_LIBOBJ . . . 14, 39, 44
AC_LIBSOURCE . . . 14, 44
AC_LIBSOURCES . . . 14
AC_OUTPUT . . . 13
AC_PREREQ . . . 22
AC_PROG_CC_C_O . . . 20
AC_PROG_CXX . . . 14
AC_PROG_F77 . . . 14
AC_PROG_FC . . . 14
AC_PROG_LEX . . . 15, 20
AC_PROG_LIBTOOL . . . 15
AC_PROG_RANLIB . . . 14
AC_PROG_YACC . . . 15
AC_SUBST . . . 15
AM_C_PROTOTYPES . . . 15, 19, 55
AM_CONDITIONAL . . . 81
AM_CONFIG_HEADER . . . 21
AM_DEP_TRACK . . . 21
AM_ENABLE_MULTILIB . . . 19
AM_GNU_GETTEXT . . . 15
AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL . . . 21
AM_INIT_AUTOMAKE . . . 12, 19
AM_MAINTAINER_MODE . . . 15, 75, 92
AM_MAKE_INCLUDE . . . 21
AM_OUTPUT_DEPENDENCY_COMMANDS . . . 21
AM_PATH_LISPDIR . . . 20
AM_PATH_PYTHON . . . 64
AM_PROG_AS . . . 20
AM_PROG_CC_C_O . . . 20
AM_PROG_GCJ . . . 20
AM_PROG_INSTALL_STRIP . . . 21
AM_PROG_LEX . . . 20
AM_SANITY_CHECK . . . 21
AM_SET_DEPDIR . . . 21
AM_SYS_POSIX_TERMIOS . . . 21
AM_WITH_DMALLOC . . . 20
AM_WITH_REGEX . . . 20

M

m4_include . . . 15, 70

B.2 Variable Index

_

_DATA . . . 58
_HEADERS . . . 57
_LIBRARIES . . . 34
_LISP . . . 62
_LTLIBRARIES . . . 35
_MANS . . . 67
_PROGRAMS . . . 3, 30
_PYTHON . . . 64
_SCRIPTS . . . 57
_SOURCES . . . 31, 44
_TEXINFOS . . . 65, 66

A

AC_PROG_FC . . . 14
ACLOCAL_AMFLAGS . . . 23, 75
ALLOCA . . . 39, 44
AM_CCASFLAGS . . . 49
AM_CFLAGS . . . 46
AM_CPPFLAGS . . . 46
AM_CXXFLAGS . . . 49
AM_ETAGSFLAGS . . . 80
AM_FCFLAGS . . . 50
AM_GCJFLAGS . . . 54
AM_INSTALLCHECK_STD_OPTIONS_EXEMPT . . . 78
AM_JAVACFLAGS . . . 63
AM_LDFLAGS . . . 31, 47
AM_LFLAGS . . . 47
AM_MAKEFLAGS . . . 25
AM_MAKEINFOFLAGS . . . 67
AM_MAKEINFOHTMLFLAGS . . . 67
AM_RFLAGS . . . 50
AM_RUNTESTFLAGS . . . 74
AM_YFLAGS . . . 47

ANSI2KNR 19
AUTOCONF 10
AUTOM4TE 16
AUTOMAKE_OPTIONS 19, 55, 56, 76

B

bin_PROGRAMS 30
bin_SCRIPTS 57
build_triplet 14
BUILT_SOURCES 58

C

CC 46
CCAS 20, 49
CCASFLAGS 20, 49
CFLAGS 46
check_ 4
check_LTLIBRARIES 37
check_PROGRAMS 30, 44
check_SCRIPTS 57
CLASSPATH_ENV 64
CLEANFILES 70
COMPILE 46
CONFIG_STATUS_DEPENDENCIES 75
CONFIGURE_DEPENDENCIES 75
CPPFLAGS 46
CXX 49
CXXCOMPILE 49
CXXFLAGS 49
CXXLINK 49, 54

D

DATA 4, 58
data_DATA 58
DEFS 46
DEJATOOL 74
DESTDIR 69
dist_ 28, 71
dist_lisp_LISP 62
dist_noinst_LISP 62
DIST_SUBDIRS 26, 71
DISTCHECK_CONFIGURE_FLAGS 72
distcleancheck_listfiles 72, 95
DISTCLEANFILES 70, 72
distdir 71, 85
distuninstallcheck_listfiles 72
DVIPS 67

E

EMACS 20
ETAGS_ARGS 80
ETAGSFLAGS 80
EXPECT 74
EXTRA_DIST 71
EXTRA_maude_SOURCES 41
EXTRA_PROGRAMS 33

F

F77LINK 54
FC 50
FCCOMPILE 50
FCFLAGS 50
FCLIBS 53
FCLINK 50, 54

G

GCJ 20
GCJFLAGS 20, 54
GCJLINK 54
GTAGS_ARGS 80
GZIP_ENV 70

H

HEADERS 4
host_triplet 14

I

include_HEADERS 57
INCLUDES 8, 46
info_TEXINFOS 65

J

JAVA 4
JAVAC 63
JAVACFLAGS 63
JAVAROOT 63

L

LDADD 31
LDFLAGS 46
LFLAGS 47
lib_LIBRARIES 34
lib_LTLIBRARIES 35
libexec_PROGRAMS 30
libexec_SCRIPTS 57
LIBOBJS 14, 39, 44
LIBRARIES 4
LIBS 46
LINK 47, 54
LISP 4
lisp_LISP 62
lispdir 20
localstate_DATA 58
LTALLOCA 39, 44
LTLIBOBJS 39, 44

M

MAINTAINERCLEANFILES 70
MAKE 25
MAKEINFO 66
MAKEINFOFLAGS 66
MAKEINFOHTML 66
man_MANS 67
MANS 4
maude_AR 41
maude_CCASFLAGS 43
maude_CFLAGS 43
maude_CPPFLAGS 43
maude_CXXFLAGS 43
maude_DEPENDENCIES 31, 42
maude_FFLAGS 43
maude_GCJFLAGS 43
maude_LDADD 31, 42
maude_LDFLAGS 31, 42
maude_LFLAGS 43
maude_LIBADD 34, 42
maude_LINK 42
maude_OBJCFLAGS 43
maude_RFLAGS 43
maude_SHORTNAME 43
maude_SOURCES 40
maude_YFLAGS 43
MOSTLYCLEANFILES 70

N

nobase_ 28
nodist_ 28, 71
noinst_ 4
noinst_HEADERS 57, 58
noinst_LIBRARIES 34
noinst_LISP 62
noinst_LTLIBRARIES 37
noinst_PROGRAMS 30
noinst_SCRIPTS 57

O

OBJCLINK 54
oldinclude_HEADERS 57

P

PACKAGE 70
pkgdata_DATA 58
pkgdata_SCRIPTS 57
pkgdatadir 3
pkginclude_HEADERS 57
pkgincludedir 3
pkglib_LIBRARIES 34
pkglib_LTLIBRARIES 35
pkglib_PROGRAMS 30
pkglibdir 3
pkgpyexecdir 65
pkgpythondir 65
PROGRAMS 3, 4
pyexecdir 65
PYTHON 4, 64
PYTHON_EXEC_PREFIX 65
PYTHON_PLATFORM 65
PYTHON_PREFIX 65
PYTHON_VERSION 64
pythondir 65

R

RFLAGS 50
RUNTEST 74
RUNTESTDEFAULTFLAGS 74
RUNTESTFLAGS 74

S

sbin_PROGRAMS 30
sbin_SCRIPTS 57
SCRIPTS 4, 57
sharedstate_DATA 58
SOURCES 31, 44
SUBDIRS 25, 71
SUFFIXES 80
sysconf_DATA 58

T

TAGS_DEPENDENCIES 80
target_triplet 14
TESTS 74
TESTS_ENVIRONMENT 74
TEXI2DVI 67
TEXI2PDF 67
TEXINFO_TEX 67
TEXINFOS 4, 66
top_distdir 71, 85

U

U 19

V

VERSION 70

W

WARNINGS 12
WITH_DMALLOC 20
WITH_REGEX 20

X

XFAIL_TESTS 74

Y

YACC . . . 15
YFLAGS . . . 47

B.3 General Index

#

## (special Automake comment) . . . 2

$

'$(LIBOBJS)' and empty libraries . . . 45

+

+= . . . 1

-

--acdir . . . 16
--add-missing . . . 10
--copy . . . 10
--cygnus . . . 11
'--enable-debug', example . . . 81
--enable-maintainer-mode . . . 15
--force . . . 16
--force-missing . . . 11
--foreign . . . 11
--gnits . . . 11
'--gnits', complete description . . . 83
--gnu . . . 11
'--gnu', complete description . . . 82
'--gnu', required files . . . 82
--help . . . 11, 16
'--help' check . . . 78
--include-deps . . . 11
--libdir . . . 10
--no-force . . . 11
--output . . . 17
--output-dir . . . 11
--print-ac-dir . . . 17
--verbose . . . 11, 17
--version . . . 11, 17
'--version' check . . . 78
--warnings . . . 11
--with-dmalloc . . . 20
--with-regex . . . 20
-a . . . 10
-c . . . 10
-f . . . 11
'-hook' targets . . . 85
-i . . . 11
-I . . . 16
'-local' targets . . . 84
'-module', libtool . . . 38
-o . . . 11
-v . . . 11
-W . . . 11

.

'.la' suffix, defined . . . 35

_

_DATA primary, defined . . . 58
_DEPENDENCIES, defined . . . 31
_HEADERS primary, defined . . . 57
_JAVA primary, defined . . . 63
_LDFLAGS, defined . . . 31
_LDFLAGS, libtool . . . 39
_LIBADD, libtool . . . 39
_LIBRARIES primary, defined . . . 34
_LISP primary, defined . . . 62
_LTLIBRARIES primary, defined . . . 35
_MANS primary, defined . . . 67
_PROGRAMS primary variable . . . 3
_PYTHON primary, defined . . . 64
_SCRIPTS primary, defined . . . 57
_SOURCES and header files . . . 31
_SOURCES primary, defined . . . 31
_SOURCES, default . . . 44
_SOURCES, empty . . . 44
_TEXINFOS primary, defined . . . 65

A

AC_SUBST and SUBDIRS . . . 26
'acinclude.m4', defined . . . 6
aclocal program, introduction . . . 6
aclocal search path . . . 17
aclocal's scheduled death . . . 24
aclocal, extending . . . 22
aclocal, Invoking . . . 15
aclocal, Options . . . 16
'aclocal.m4', preexisting . . . 6
Adding new SUFFIXES . . . 80
all . . . 84
all-local . . . 84
ALLOCA, and Libtool . . . 39
ALLOCA, example . . . 44
ALLOCA, special handling . . . 44
AM_CCASFLAGS and CCASFLAGS . . . 96
AM_CFLAGS and CFLAGS . . . 96
AM_CONDITIONAL and SUBDIRS . . . 26
AM_CPPFLAGS and CPPFLAGS . . . 96
AM_CXXFLAGS and CXXFLAGS . . . 96
AM_FCFLAGS and FCFLAGS . . . 96
AM_FFLAGS and FFLAGS . . . 96
AM_GCJFLAGS and GCJFLAGS . . . 96
AM_INIT_AUTOMAKE, example use . . . 6
AM_LDFLAGS and LDFLAGS . . . 96

AM_LFLAGS and LFLAGS . . . 96
AM_MAINTAINER_MODE, purpose . . . 92
AM_OBJCFLAGS and OBJCFLAGS . . . 96
AM_RFLAGS and RFLAGS . . . 96
AM_YFLAGS and YFLAGS . . . 96
ansi2knr . . . 55, 76
ansi2knr and LIBOBJS . . . 55
ansi2knr and LTLIBOBJS . . . 55
Append operator . . . 1
'autogen.sh' and autoreconf . . . 39
autom4te . . . 16
Automake constraints . . . 1
automake options . . . 10
Automake requirements . . . 1, 12
automake, invoking . . . 10
Automake, recursive operation . . . 2
Automatic dependency tracking . . . 55
Automatic linker selection . . . 54
autoreconf and libtoolize . . . 39
autoupdate . . . 21
Auxiliary programs . . . 5
Avoiding path stripping . . . 28

B

'bootstrap.sh' and autoreconf . . . 39
Bugs, reporting . . . 1
BUILT_SOURCES, defined . . . 58

C

C++ support . . . 49
canonicalizing Automake variables . . . 4
CCASFLAGS and AM_CCASFLAGS . . . 96
CFLAGS and AM_CFLAGS . . . 96
cfortran . . . 52
check . . . 73, 84
check-local . . . 84
check-news . . . 76
'check_' primary prefix, definition . . . 4
check_PROGRAMS example . . . 44
clean . . . 84
clean-local . . . 70, 84
Comment, special to Automake . . . 2
Compile Flag Variables . . . 96
Complete example . . . 6
Conditional example, '--enable-debug' . . . 81
conditional libtool libraries . . . 36
Conditional programs . . . 33
Conditional subdirectories . . . 25
Conditional SUBDIRS . . . 25
Conditionals . . . 81
'config.guess' . . . 10
'configure.ac', from GNU Hello . . . 7
'configure.ac', scanning . . . 12
conflicting definitions . . . 84
Constraints of Automake . . . 1
convenience libraries, libtool . . . 37
copying semantics . . . 84
cpio example . . . 3
CPPFLAGS and AM_CPPFLAGS . . . 96
CVS and generated files . . . 90
CVS and third-party files . . . 92
CVS and timestamps . . . 90
cvs-dist . . . 1
cvs-dist, non-standard example . . . 1
CXXFLAGS and AM_CXXFLAGS . . . 96
cygnus . . . 76
'cygnus' strictness . . . 83

D

DATA primary, defined . . . 58
de-ANSI-fication, defined . . . 55
default _SOURCES . . . 44
default source, Libtool modules example . . . 44
definitions, conflicts . . . 84
dejagnu . . . 74, 76
depcomp . . . 55
dependencies and distributed files . . . 94
Dependency tracking . . . 55
Dependency tracking, disabling . . . 56
'dirlist' . . . 18
Disabling dependency tracking . . . 56
dist . . . 70
dist-bzip2 . . . 73, 76
dist-gzip . . . 73
dist-hook . . . 71, 85
dist-shar . . . 73, 76
dist-tarZ . . . 73, 77
dist-zip . . . 73, 76
dist_ and nobase_ . . . 28
DIST_SUBDIRS, explained . . . 26
distcheck . . . 72
distcheck-hook . . . 72
distclean . . . 84, 94
distclean, diagnostic . . . 94
distclean-local . . . 70, 84
distcleancheck . . . 72, 94
distdir . . . 85
dmalloc, support for . . . 20
dvi . . . 84
DVI output using Texinfo . . . 65
dvi-local . . . 84

E

E-mail, bug reports . . . 1
EDITION Texinfo flag . . . 65
else . . . 82
empty _SOURCES . . . 44
Empty libraries . . . 34
Empty libraries and '$(LIBOBJS)' . . . 45
endif . . . 82
Example conditional '--enable-debug' . . . 81
Example of recursive operation . . . 2

Example of shared libraries 35
Example, `EXTRA_PROGRAMS` 3
Example, `false` and `true` 8
Example, GNU Hello 7
Example, handling Texinfo files 8
Example, regression test 8
Executable extension 56
Exit status 77, special interpretation 74
Expected test failure 74
Extending `aclocal` 22
Extending list of installation directories 3
Extension, executable 56
Extra files distributed with Automake 10
`EXTRA_`, prepending 3
`EXTRA_prog_SOURCES`, defined 32
`EXTRA_PROGRAMS`, defined 3, 33

F

`false` Example 8
`FCFLAGS` and `AM_FCFLAGS` 96
`FCLIBS`, defined 53
FDL, GNU Free Documentation License 123
`FFLAGS` and `AM_FFLAGS` 96
`filename-length-max=99` 77
Files distributed with Automake 10
First line of Makefile.am 2
Flag variables, ordering 96
Flag Variables, Ordering 96
`foreign` 76
‘`foreign`’ strictness 2
Fortran support 49
Fortran, mixing with C and C++ 52
Fortran, Preprocessing 51

G

`GCJFLAGS` and `AM_GCJFLAGS` 96
generated files and CVS 90
generated files, distributed 90
Gettext support 63
`gnits` 76
‘`gnits`’ strictness 2
`gnu` 76
GNU Gettext support 63
GNU Hello, ‘`configure.ac`’ 7
GNU Hello, example 7
GNU make extensions 1
GNU Makefile standards 1
‘`gnu`’ strictness 2
‘`GNUmakefile`’ including ‘`Makefile`’ 87

H

Header files in `_SOURCES` 31
`HEADERS` primary, defined 57
`HEADERS`, installation directories 57
Hello example 7
Hello, ‘`configure.ac`’ 7
hook targets 85
HP-UX 10, `lex` problems 20
`html` 84
HTML output using Texinfo 65
`html-local` 84

I

`id` 80
`if` 82
`include` 70, 81
`include`, distribution 70
`INCLUDES`, example usage 8
Including ‘`Makefile`’ fragment 81
`info` 77, 84
`info-local` 84
`install` 68, 84
Install hook 69
Install, two parts of 68
`install-data` 68
`install-data-hook` 85
`install-data-local` 69, 84
`install-exec` 68, 84
`install-exec-hook` 85
`install-exec-local` 69, 84
`install-info` 66, 77
`install-info` target 66
`install-man` 68, 77
`install-man` target 68
`install-strip` 69
Installation directories, extending list 3
Installation support 68
`installcheck` 84
`installcheck-local` 84
`installdirs` 69, 84
`installdirs-local` 84
Installing headers 57
Installing scripts 57
installing versioned binaries 85
Interfacing with third-party packages 85
Invoking `aclocal` 15
Invoking `automake` 10

J

`JAVA` primary, defined 63
`JAVA` restrictions 63
Java support 54

L

`LDFLAGS` and `AM_LDFLAGS` 96
`lex` problems with HP-UX 10 20
`lex`, multiple lexers 47
`LFLAGS` and `AM_LFLAGS` 96
‘`libltdl`’, introduction 35
`LIBOBJS` and `ansi2knr` 55

LIBOBJS, and Libtool 39
LIBOBJS, example 44
LIBOBJS, special handling 44
LIBRARIES primary, defined 34
libtool convenience libraries 37
libtool libraries, conditional 36
libtool library, definition 35
libtool modules 38
Libtool modules, default source example 44
libtool, introduction 35
libtoolize and autoreconf 39
libtoolize, no longer run by automake 39
Linking Fortran with C and C++ 52
LISP primary, defined 62
LN_S example 85
local targets 84
LTALLOCA, special handling 39
LTLIBOBJS and ansi2knr 55
LTLIBOBJS, special handling 39
LTLIBRARIES primary, defined 35
'ltmain.sh' not found 39

M

m4_include, distribution 70
Macro search path 17
Macros Automake recognizes 13
maintainer-clean-local 70
make check 73
'make clean' support 70
'make dist' 70
'make distcheck' 72
'make distclean', diagnostic 94
'make distcleancheck' 72
'make distuninstallcheck' 72
'make install' support 68
'make installcheck', testing '--help' and '--version' 78
Make rules, overriding 2
Make targets, overriding 2
'Makefile' fragment, including 81
Makefile.am, first line 2
MANS primary, defined 67
many outputs, rules with 100
'mdate-sh' 65
missing, purpose 92
Mixing Fortran with C and C++ 52
Mixing Fortran with C and/or C++ 52
modules, libtool 38
mostlyclean 84
mostlyclean-local 70, 84
Multiple 'configure.ac' files 10
Multiple lex lexers 47
multiple outputs, rules with 100
Multiple yacc parsers 47

N

Nesting packages 29
no-define 20, 77
no-dependencies 56, 77
no-dist 77
no-dist-gzip 77
no-exeext 77
no-installinfo 66, 77
no-installman 68, 77
no-texinfo.tex 66, 77
nobase_ and dist_ or nodist_ 28
nobase_ prefix 28
nodist_ and nobase_ 28
'noinst_' primary prefix, definition 4
noinstall-info option 66
'noinstall-man' option 68
Non-GNU packages 2
Non-standard targets 1
nostdinc 77

O

OBJCFLAGS and AM_OBJCFLAGS 96
Objects in subdirectory 41
obsolete macros 21
Option, '--warnings=*category*' 79
Option, '-W*category*' 79
Option, 'ansi2knr' 76
Option, 'check-news' 76
Option, 'cygnus' 76
Option, 'dejagnu' 76
Option, 'dist-bzip2' 76
Option, 'dist-shar' 76
Option, 'dist-tarZ' 77
Option, 'dist-zip' 76
Option, 'filename-length-max=99' 77
Option, 'foreign' 76
Option, 'gnits' 76
Option, 'gnu' 76
Option, 'no-define' 77
Option, 'no-dependencies' 77
Option, 'no-dist' 77
Option, 'no-dist-gzip' 77
Option, 'no-exeext' 77
Option, 'no-installinfo' 77
Option, 'no-installman' 77
Option, 'no-texinfo.tex' 77
Option, noinstall-info 66
Option, 'noinstall-man' 68
Option, 'nostdinc' 77
Option, 'readme-alpha' 77
Option, 'tar-pax' 78
Option, 'tar-ustar' 78
Option, 'tar-v7' 78
Option, *version* 79
Option, warnings 79
Options, aclocal 16
Options, automake 10

Options, 'std-options' . . . 78
Options, 'subdir-objects' . . . 78
Ordering flag variables . . . 96
Overriding make rules . . . 2
Overriding make targets . . . 2
Overriding make variables . . . 2
overriding rules . . . 84
overriding semantics . . . 84

P

PACKAGE, directory . . . 3
PACKAGE, prevent definition . . . 20
Path stripping, avoiding . . . 28
pax format . . . 78
pdf . . . 84
PDF output using Texinfo . . . 65
pdf-local . . . 84
Per-object flags, emulated . . . 99
per-target compilation flags, defined . . . 43
pkgdatadir, defined . . . 3
pkgincludedir, defined . . . 3
pkglibdir, defined . . . 3
POSIX termios headers . . . 21
Preprocessing Fortran . . . 51
Primary variable, DATA . . . 58
Primary variable, defined . . . 3
Primary variable, HEADERS . . . 57
Primary variable, JAVA . . . 63
Primary variable, LIBRARIES . . . 34
Primary variable, LISP . . . 62
Primary variable, LTLIBRARIES . . . 35
Primary variable, MANS . . . 67
Primary variable, PROGRAMS . . . 3
Primary variable, PYTHON . . . 64
Primary variable, SCRIPTS . . . 57
Primary variable, SOURCES . . . 31
Primary variable, TEXINFOS . . . 65
prog_LDADD, defined . . . 31
PROGRAMS primary variable . . . 3
Programs, auxiliary . . . 5
PROGRAMS, bindir . . . 30
Programs, conditional . . . 33
Proxy 'Makefile' for third-party packages . . . 87
ps . . . 84
PS output using Texinfo . . . 65
ps-local . . . 84
PYTHON primary, defined . . . 64

R

Ratfor programs . . . 51
readme-alpha . . . 77
'README-alpha' . . . 83
rebuild rules . . . 75, 90
Recognized macros by Automake . . . 13
Recursive operation of Automake . . . 2
recursive targets and third-party 'Makefile's . . . 85
regex package . . . 20
Regression test example . . . 8
Reporting bugs . . . 1
Requirements of Automake . . . 12
Requirements, Automake . . . 1
Restrictions for JAVA . . . 63
RFLAGS and AM_RFLAGS . . . 96
rules with multiple outputs . . . 100
rules, conflicting . . . 84
rules, overriding . . . 84
rx package . . . 20

S

Scanning 'configure.ac' . . . 12
SCRIPTS primary, defined . . . 57
SCRIPTS, installation directories . . . 57
Selecting the linker automatically . . . 54
Shared libraries, support for . . . 34
'site.exp' . . . 74
SOURCES primary, defined . . . 31
Special Automake comment . . . 2
std-options . . . 78
Strictness, command line . . . 10
Strictness, defined . . . 2
Strictness, 'foreign' . . . 2
Strictness, 'gnits' . . . 2
Strictness, 'gnu' . . . 2
subdir-objects . . . 78
Subdirectories, building conditionally . . . 25
Subdirectories, configured conditionally . . . 27
Subdirectories, not distributed . . . 28
Subdirectory, objects in . . . 41
SUBDIRS and AC_SUBST . . . 26
SUBDIRS and AM_CONDITIONAL . . . 26
SUBDIRS, conditional . . . 25
SUBDIRS, explained . . . 25
Subpackages . . . 29
suffix '.la', defined . . . 35
suffix '.lo', defined . . . 35
SUFFIXES, adding . . . 80
Support for C++ . . . 49
Support for Fortran . . . 49
Support for GNU Gettext . . . 63
Support for Java . . . 54

T

tags . . . 79
'TAGS' support . . . 79
tar formats . . . 78
tar-pax . . . 78
tar-ustar . . . 78
tar-v7 . . . 78
Target, install-info . . . 66
Target, install-man . . . 68
termios POSIX headers . . . 21
Test suites . . . 73

Tests, expected failure 74
Texinfo file handling example 8
Texinfo flag, EDITION 65
Texinfo flag, UPDATED 65
Texinfo flag, UPDATED-MONTH 65
Texinfo flag, VERSION 65
'texinfo.tex' 66
TEXINFOS primary, defined 65
third-party files and CVS 92
Third-party packages, interfacing with 85
timestamps and CVS 90
true Example 8

U

underquoted AC_DEFUN 22
Uniform naming scheme 3
uninstall 69, 84
uninstall-hook 85
uninstall-local 84
UPDATED Texinfo flag 65
UPDATED-MONTH Texinfo flag 65
user variables 4
ustar format 78

V

v7 tar format 78
variables, conflicting 84
Variables, overriding 2
variables, reserved for the user 4
VERSION Texinfo flag 65
VERSION, prevent definition 20
'version.m4', example 75
'version.sh', example 75
versioned binaries, installing 85

W

wildcards 93
Windows 56

Y

yacc, multiple parsers 47
YFLAGS and AM_YFLAGS 96
ylwrap 47

Z

zardoz example 7

Table of Contents

1 Introduction **1**

2 General ideas **1**

2.1 General Operation 1
2.2 Strictness 2
2.3 The Uniform Naming Scheme 3
2.4 How derived variables are named 4
2.5 Variables reserved for the user 4
2.6 Programs automake might require 5

3 Some example packages **6**

3.1 A simple example, start to finish 6
3.2 A classic program 7
3.3 Building true and false 8

4 Creating a ‘`Makefile.in`’ **10**

5 Scanning ‘`configure.ac`’ **12**

5.1 Configuration requirements 12
5.2 Other things Automake recognizes 13
5.3 Auto-generating aclocal.m4 15
5.4 aclocal options 16
5.5 Macro search path 17
5.5.1 Modifying the macro search path: `--acdir` 17
5.5.2 Modifying the macro search path: `-I` *`dir`* 17
5.5.3 Modifying the macro search path: ‘`dirlist`’ 18
5.6 Autoconf macros supplied with Automake 19
5.6.1 Public macros 19
5.6.2 Obsolete macros 21
5.6.3 Private macros 21
5.7 Writing your own aclocal macros 22
5.8 Handling Local Macros 23
5.9 The Future of `aclocal` 24

6 Directories **24**

6.1 Recursing subdirectories 25
6.2 Conditional Subdirectories 25
6.2.1 `SUBDIRS` vs. `DIST_SUBDIRS` 26
6.2.2 Conditional subdirectories with `AM_CONDITIONAL` 26
6.2.3 Conditional Subdirectories with `AC_SUBST` 26
6.2.4 Non-configured Subdirectories 27
6.3 An Alternative Approach to Subdirectories 28
6.4 Nesting Packages 29

7 Building Programs and Libraries 30
7.1 Building a program 30
7.1.1 Defining program sources 30
7.1.2 Linking the program 31
7.1.3 Conditional compilation of sources 32
7.1.3.1 Conditional compilation using `_LDADD` substitutions . . 32
7.1.3.2 Conditional compilation using Automake conditionals 32
7.1.4 Conditional compilation of programs 33
7.1.4.1 Conditional programs using `configure` substitutions 33
7.1.4.2 Conditional programs using Automake conditionals . . 33
7.2 Building a library 34
7.3 Building a Shared Library 34
7.3.1 The Libtool Concept 35
7.3.2 Building Libtool Libraries 35
7.3.3 Building Libtool Libraries Conditionally 36
7.3.4 Libtool Libraries with Conditional Sources 37
7.3.5 Libtool Convenience Libraries 37
7.3.6 Libtool Modules 38
7.3.7 _LIBADD and _LDFLAGS 39
7.3.8 `LTLIBOBJS` and `LTALLOCA` 39
7.3.9 Common Issues Related to Libtool's Use 39
7.3.9.1 '`required file './ltmain.sh' not found`' 39
7.3.9.2 Objects '`created with both libtool and without`' . . 40
7.4 Program and Library Variables 40
7.5 Default `_SOURCES` 43
7.6 Special handling for `LIBOBJS` and `ALLOCA` 44
7.7 Variables used when building a program 46
7.8 Yacc and Lex support 47
7.9 C++ Support 49
7.10 Assembly Support 49
7.11 Fortran Support 49
7.11.1 Overview 49
7.11.2 Compiling Fortran Files 51
7.11.3 Preprocessing Fortran 51
7.11.4 Mixing Fortran With C and C++ 52
7.11.4.1 How the Linker is Chosen 54
7.12 Java Support 54
7.13 Support for Other Languages 54
7.14 Automatic de-ANSI-fication 55
7.15 Automatic dependency tracking 55
7.16 Support for executable extensions 56

8 Other Derived Objects 57
8.1 Executable Scripts 57
8.2 Header files 57
8.3 Architecture-independent data files 58
8.4 Built sources 58
8.4.1 Built sources example 59
First try 59
Using `BUILT_SOURCES` 60
Recording dependencies manually 60
Build '`bindir.h`' from '`configure`' 61
Build '`bindir.c`', not '`bindir.h`'. 61
Which is best? 62

9 Other GNU Tools 62
9.1 Emacs Lisp 62
9.2 Gettext 63
9.3 Libtool 63
9.4 Java 63
9.5 Python 64

10 Building documentation 65
10.1 Texinfo 65
10.2 Man pages 67

11 What Gets Installed 68
11.1 Basics of installation 68
11.2 The two parts of install 68
11.3 Extending installation 69
11.4 Staged installs 69
11.5 Rules for the user 69

12 What Gets Cleaned 70

13 What Goes in a Distribution 70
13.1 Basics of distribution 70
13.2 Fine-grained distribution control 71
13.3 The dist hook 71
13.4 Checking the distribution 72
13.5 The types of distributions 73

14 Support for test suites 73
14.1 Simple Tests 74
14.2 DejaGnu Tests 74
14.3 Install Tests 75

15 Rebuilding Makefiles **75**

16 Changing Automake's Behavior **76**

17 Miscellaneous Rules **79**

17.1 Interfacing to `etags` 79
17.2 Handling new file extensions 80
17.3 Support for Multilibs 81

18 Include **81**

19 Conditionals **81**

20 The effect of '`--gnu`' and '`--gnits`' **82**

21 The effect of '`--cygnus`' **83**

22 When Automake Isn't Enough **84**

22.1 Extending Automake Rules 84
22.2 Third-Party '`Makefile`'s 85

23 Distributing '`Makefile.in`'s **88**

24 Automake API versioning **88**

25 Upgrading a Package to a Newer Automake Version **89**

26 Frequently Asked Questions about Automake **90**

26.1 CVS and generated files 90
26.1.1 Background: distributed generated files 90
26.1.2 Background: CVS and timestamps 90
26.1.3 Living with CVS in Autoconfiscated projects 90
26.1.4 Third-party files 92
26.2 `missing` and `AM_MAINTAINER_MODE` 92
26.2.1 `missing` 92
26.2.2 `AM_MAINTAINER_MODE` 92
26.3 Why doesn't Automake support wildcards? 93
26.4 Files left in build directory after distclean 94
26.5 Flag Variables Ordering 96
26.5.1 Compile Flag Variables 96
26.5.2 Other Variables 98

26.6 Why are object files sometimes renamed? 98
26.7 Per-Object Flags Emulation 99
26.8 Handling Tools that Produce Many Outputs 100
26.9 Installing to Hard-Coded Locations 104

27 History of Automake 105

27.1 Timeline 106
27.2 Dependency Tracking in Automake 117
27.2.1 First Take 117
Description 117
Bugs 117
Historical Note 118
27.2.2 Dependencies As Side Effects 118
Description 118
Bugs 118
27.2.3 Dependencies for the User 119
Description 119
Bugs 119
27.2.4 Techniques for Computing Dependencies 120
27.2.5 Recommendations for Tool Writers 121
27.2.6 Future Directions for Automake’s Dependency Tracking 121
27.3 Release Statistics 121

Appendix A Copying This Manual 123

A.1 GNU Free Documentation License 123
A.1.1 ADDENDUM: How to use this License for your documents 129

Appendix B Indices 130

B.1 Macro Index 130
B.2 Variable Index 130
B.3 General Index 133