

GNU Automake

For version 1.5, 22 August 2001

David MacKenzie and Tom Tromeey

Copyright © 1995, 1996, 2000, 2001 Free Software Foundation, Inc.

This is the first edition of the GNU Automake documentation,
and is consistent with GNU Automake 1.5.

Published by the Free Software Foundation
59 Temple Place - Suite 330,
Boston, MA 02111-1307 USA

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

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` macro definitions (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 macro 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.in`' contents¹.

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 macros and targets defined in the `Makefile.am`' instruct Automake to generate more specialized code; for instance, a `bin_PROGRAMS`' macro definition will cause targets for compiling and linking programs to be generated.

The macro definitions and targets 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 `cvs-dist` target, which the Automake maintainer uses to make distributions from his source control system.

Note that GNU make extensions are not recognized by Automake. Using such extensions in a `Makefile.am`' will lead to errors or confusing behavior.

Automake tries to group comments with adjoining targets and macro definitions in an intelligent way.

A target defined in `Makefile.am`' generally overrides any such target of a similar name that would be automatically generated by `automake`. Although this is a supported feature,

¹ Autoconf 2.50 promotes `configure.ac`' over `configure.in`'. The rest of this documentation will refer to `configure.in`' as this use is not yet spread, but Automake supports `configure.ac`' too.

it is generally best to avoid making use of it, as sometimes the generated rules are very particular.

Similarly, a macro defined in ‘`Makefile.am`’ will override any definition of the macro that `automake` would ordinarily create. This feature is more often useful than the ability to override a target definition. Be warned that many of the macros generated by `automake` are considered to be for internal use only, and their names might change in future releases.

When examining a macro definition, Automake will recursively examine macros 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 which 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 which 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).

For more information on the precise implications of the strictness level, see Chapter 21 [Gnits], page 49.

Automake also has a special “cygnus” mode which is similar to strictness but handled differently. This mode is useful for packages which are put into a “Cygnus” style tree (e.g., the GCC tree). For more information on this mode, see Chapter 22 [Cygnus], page 50.

2.3 The Uniform Naming Scheme

Automake macros (from here on referred to as *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 which are concatenated together.

The piece which tells automake what is being built is commonly called the *primary*. For instance, the primary `PROGRAMS` holds a list of programs which 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 which 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 which 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 are 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, until HTML support is part of Automake, you could use this to install raw HTML documentation:

```
html_dir = $(prefix)/html
html_DATA = automake.html
```

The special prefix ‘`noinst`’ indicates that the objects in question should not be installed at all.

The special prefix ‘`check`’ indicates that the objects in question should not be built until the `make check` command is run.

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

Some primaries also allow additional prefixes which control other aspects of `automake`’s behavior. The currently defined prefixes are ‘`dist_`’, ‘`nodist_`’, and ‘`nobase_`’. These prefixes are explained later.

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 macro naming rules. All characters in the name except for letters, numbers, the strudel (`@`), and the underscore are turned into underscores when making macro references.

For example, if your program is named `sniff-glue`, the derived variable name would be `sniff_glue_SOURCES`, not `sniff-glue_SOURCES`.

The strudel is an addition, to make the use of Autoconf substitutions in macro 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 – they don’t have to introduce a second variable into every target.

However, the package itself should never set a user variable, particularly not to include switches which 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`.

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.

```
ansi2knr.c
ansi2knr.1
```

These two files are used by the automatic de-ANSI-fication support (see Section 9.13 [ANSI], page 32).

- compile** This is a wrapper for compilers which don't accept both `-c` and `-o` at the same time. It is only used when absolutely required. Such compilers are rare.
- config.guess**
config.sub
These programs compute the canonical triplets for the given build, host, or target architecture.
- depcomp** This program understands how to run a compiler so that it will generate not only the desired output but also dependency information which 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 which 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 which 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 works around the fact that `mkdir -p` is not portable.
- py-compile**
This is used to byte-compile Python scripts.
- texinfo.tex**
Not a program, this file is required for `make dvi` to work when Texinfo sources are in the package.
- 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.in` to include the commands that `automake` needs. The way to do this is to add an `AM_INIT_AUTOMAKE` call just after `AC_INIT`:

```
AM_INIT_AUTOMAKE(zardoz, 1.0)
```

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... 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. Additionally, `zardoz` has some Texinfo documentation. Your `'configure.in'` script uses `AC_REPLACE_FUNCS`, so you need to link against `'@LIBOBJ@'`. So here's what you'd write:

```
bin_PROGRAMS = zardoz
zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c
zardoz_LDADD = @LIBOBJ@

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 (<ftp://prep.ai.mit.edu/pub/gnu/hello-1.3.tar.gz>) 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.in'` from GNU Hello:

```
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.0
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 etags and ctags

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

```

bin_PROGRAMS = etags ctags
ctags_SOURCES =
ctags_LDADD = ctags.o

etags.o: etags.c
    $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
    $(COMPILE) -DCTAGS -o ctags.o -c etags.c

```

Note that `ctags_SOURCES` is defined to be empty—that way no implicit value is substituted. The implicit value, however, is used to generate `etags` from `'etags.o'`.

`ctags_LDADD` is used to get `'ctags.o'` into the link line. `ctags_DEPENDENCIES` is generated by Automake.

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):

```

etags.o: etags.c ctags.o
    $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
    $(COMPILE) -DCTAGS -c etags.c && mv etags.o ctags.o

```

Also, these explicit rules do not work if the de-ANSI-fication feature is used (see Section 9.13 [ANSI], page 32). Supporting de-ANSI-fication requires a little more work:

```

etags._o: etags._c ctags.o
    $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags._o: etags._c
    $(COMPILE) -DCTAGS -c etags.c && mv etags._o ctags.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 `etags` and `ctags` in real life, you would probably use per-program compilation flags, like so:

```

bin_PROGRAMS = ctags etags

ctags_SOURCES = etags.c
ctags_CFLAGS = -DCTAGS

etags_SOURCES = etags.c
etags_CFLAGS = -DETAGS_REGEXPS

```

In this case Automake will cause `'etags.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 `'ctags-etags.c'` and `'etags-etags.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.in’; see Chapter 5 [configure], page 10) 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.in’, at the top. If your package has multiple ‘configure.in’s, then you must run `automake` in each directory holding a ‘configure.in’.

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.in’, and because `automake` uses the knowledge that a ‘Makefile.in’ is in a subdirectory to change its behavior in some cases.

`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.in’ runs `AC_CANONICAL_HOST`. Automake is distributed with several of these files; 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`.

‘--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 22 [Cygnus], page 50.

‘-f’

‘--force-missing’

When used with `--add-missing`, causes standard files to be rebuilt 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 21 [Gnits], page 49.
- ‘`--gnu`’ Set the global strictness to ‘`gnu`’. For more information, see Chapter 21 [Gnits], page 49. 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; see Section 9.14 [Dependencies], page 33.
- ‘`--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.in`’. This option causes it to only update those ‘`Makefile.in`’s which 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 used when making distributions.
- ‘`-v`’
- ‘`--verbose`’
Cause Automake to print information about which files are being read or created.
- ‘`--version`’
Print the version number of Automake and exit.
- ‘`--Werror`’
- ‘`--Wno-error`’
‘`--Werror`’ will cause all warnings issued by `automake` to become errors. Errors affect the exit status of `automake`, while warnings do not. ‘`--Wno-error`’, the default, causes warnings to be treated as warnings only.

5 Scanning ‘`configure.in`’

Automake scans the package’s ‘`configure.in`’ to determine certain information about the package. Some `autoconf` macros are required and some variables must be defined in ‘`configure.in`’. Automake will also use information from ‘`configure.in`’ 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.in’ call `AM_INIT_AUTOMAKE`. This macro does several things which are required for proper Automake operation.

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

AC_OUTPUT

Automake uses this to determine which files to create (see section “Creating Output Files” in *The Autoconf Manual*). Listed files named `Makefile` are treated as ‘`Makefile`’s. Other listed files are treated differently. Currently the only difference is that a ‘`Makefile`’ is removed by `make distclean`, while other files are removed by `make clean`.

You may need the following macros in some conditions, even though they are not required.

AC_CHECK_TOOL([STRIP],[strip])

Installed binaries are usually stripped using `strip` when you run `make install-strip`. However `strip` might not be the right tool to use in cross-compilation environments, therefore Automake will honor the `STRIP` environment variable to overrule the program used to perform stripping. Automake will not set `STRIP` itself. If your package is not setup for cross-compilation you do not have to care (`strip` is ok), otherwise you can set `STRIP` automatically by calling `AC_CHECK_TOOL([STRIP],[strip])` from your ‘configure.in’.

5.2 Other things Automake recognizes

Automake will also recognize the use of certain macros and tailor the generated ‘`Makefile.in`’ appropriately. Currently recognized macros and their effects are:

AC_CONFIG_HEADER

Automake requires the use of `AM_CONFIG_HEADER`, which is similar to `AC_CONFIG_HEADER` (see section “Configuration Header Files” in *The Autoconf Manual*), but does some useful Automake-specific work.

AC_CONFIG_AUX_DIR

Automake will look for various helper scripts, such as ‘`mkinstalldirs`’, in the directory named in this macro invocation. If not seen, the scripts are looked for in their ‘`standard`’ locations (either the top source directory, or in the source directory corresponding to the current ‘`Makefile.am`’, whichever is appropriate). See section “Finding ‘configure’ Input” in *The Autoconf Manual*. **FIXME:** give complete list of things looked for in this directory

AC_PATH_XTRA

Automake will insert definitions for the variables defined by `AC_PATH_XTRA` into each ‘`Makefile.in`’ that builds a C program or library. See section “System Services” in *The Autoconf Manual*.

AC_CANONICAL_HOST**AC_CHECK_TOOL**

Automake will ensure that ‘config.guess’ and ‘config.sub’ exist. Also, the ‘Makefile’ variables ‘host_alias’ and ‘host_triplet’ are introduced. See both section “Getting the Canonical System Type” in *The Autoconf Manual*, and section “Generic Program Checks” in *The Autoconf Manual*.

AC_CANONICAL_SYSTEM

This is similar to AC_CANONICAL_HOST, but also defines the ‘Makefile’ variables ‘build_alias’ and ‘target_alias’. See section “Getting the Canonical System Type” in *The Autoconf Manual*.

AC_FUNC_ALLOCA**AC_FUNC_GETLOADAVG****AC_FUNC_MEMCMP****AC_STRUCT_ST_BLOCKS****AC_FUNC_FNMATCH****AC_FUNC_MKTIME****AM_FUNC_STRTOI****AC_REPLACE_FUNCS****AC_REPLACE_GNU_GETOPT****AM_WITH_REGEX**

Automake will ensure that the appropriate dependencies are generated for the objects corresponding to these macros. Also, Automake will verify that the appropriate source files are part of the distribution. Note that Automake does not come with any of the C sources required to use these macros, so `automake -a` will not install the sources. See Section 9.2 [A Library], page 21, for more information. Also, see section “Particular Function Checks” in *The Autoconf Manual*.

LIBOBJJS Automake will detect statements which put ‘.o’ files into LIBOBJJS, and will treat these additional files as if they were discovered via AC_REPLACE_FUNCS. See section “Generic Function Checks” 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_F77_LIBRARY_LDFLAGS

This is required for programs and shared libraries that are a mixture of languages that include Fortran 77 (see Section 9.10.3 [Mixing Fortran 77 With C

and C++], page 29). See Section 5.4 [Autoconf macros supplied with Automake], page 14.

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.in’. The former is preferred (see section “Particular Program Checks” in *The Autoconf Manual*).

AC_DECL_YTEXT

This macro is required if there is Lex source in the package. 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*.

AM_C_PROTOTYPES

This is required when using automatic de-ANSI-fication; see Section 9.13 [ANSI], page 32.

AM_GNU_GETTEXT

This macro is required for packages which use GNU gettext (see Section 11.2 [gettext], page 36). 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 is disallowed in ‘Gnits’ mode (see Chapter 21 [Gnits], page 49). This macro defines the ‘MAINTAINER_MODE’ conditional, which you can use in your own ‘`Makefile.am`’.

AC_SUBST**AC_CHECK_TOOL****AC_CHECK_PROG****AC_CHECK_PROGS****AC_PATH_PROG****AC_PATH_PROGS**

For each of these macros, the first argument is automatically defined as a variable in each generated ‘`Makefile.in`’. See section “Setting Output Variables” in *The Autoconf Manual*, and section “Generic Program Checks” in *The Autoconf Manual*.

5.3 Auto-generating `aclocal.m4`

Automake includes a number of Autoconf macros which can be used in your package; 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.in`’. This provides a convenient way to get Automake-provided macros, without having to search around. Also, the `aclocal` mechanism is extensible for use by other packages.

At startup, `aclocal` scans all the ‘.m4’ files it can find, looking for macro definitions. Then it scans ‘`configure.in`’. 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`’.

The contents of ‘`acinclude.m4`’, if it exists, are also automatically included in ‘`aclocal.m4`’. This is useful for incorporating local macros into ‘`configure`’.

`aclocal` tries to be smart about looking for new `AC_DEFUNs` in the files it scans. It will warn if it finds duplicates. It also tries to copy the full text of the scanned file into ‘`aclocal.m4`’, including both ‘`#`’ and ‘`dn1`’ comments. If you want to make a comment which will be completely ignored by `aclocal`, use ‘`##`’ as the comment leader.

`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.
- `--output=file`
Cause the output to be put into *file* instead of ‘`aclocal.m4`’.
- `--print-ac-dir`
Prints the name of the directory which `aclocal` will search to find the ‘.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.

5.4 Autoconf macros supplied with Automake

`AM_CONFIG_HEADER`

Automake will generate rules to automatically regenerate the config header. If you do use this macro, you must create the file ‘`stamp-h.in`’ in your source directory. It can be empty.

`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 18.3 [Multilibs], page 48.

_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_FUNC_STRTOU

If the `strtou` function is not available, or does not work correctly (like the one on SunOS 5.4), add ‘`strtou.o`’ to output variable `LIBOBJS`.

AM_FUNC_ERROR_AT_LINE

If the function `error_at_line` is not found, then add ‘`error.o`’ to `LIBOBJS`.

AM_FUNC_OBSTACK

Check for the GNU obstacks code; if not found, add ‘`obstack.o`’ to ‘`LIBOBJS`’.

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_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>`’.

AM_INIT_AUTOMAKE

Runs many macros that most ‘`configure.in`’s need. This macro has two required arguments, the package and the version number. By default this macro `AC_DEFINE`s ‘`PACKAGE`’ and ‘`VERSION`’. This can be avoided by passing in a non-empty third argument.

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_PATH_LISPDIR

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

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 will set `ASFLAGS` 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_CC_STDC

If the C compiler is not in ANSI C mode by default, try to add an option to output variable `CC` to make it so. This macro tries various options that select ANSI C on some system or another. It considers the compiler to be in ANSI C mode if it handles function prototypes correctly.

If you use this macro, you should check after calling it whether the C compiler has been set to accept ANSI C; if not, the shell variable `am_cv_prog_cc_stdc` is set to ‘no’. If you wrote your source code in ANSI C, you can make an un-ANSIfied copy of it by using the `ansi2knr` option (see Section 9.13 [ANSI], page 32).

AM_PROG_LEX

Like `AC_PROG_LEX` with `AC_DECL_YTEXT` (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.

Autoconf 2.50 and higher, in order to simplify the interface, includes the body of `AC_DECL_YTEXT` in `AC_PROG_LEX`. To ensure backward compatibility, `AC_DECL_YTEXT` is nevertheless defined as an invocation of `AC_PROG_LEX`. Since `AM_PROG_LEX` invokes both, it causes an annoying but benign warning (`AC_PROG_LEX` invoked multiple times) which you should just ignore. In the future, once Automake requires Autoconf 2.50, this issue will be fixed, but the current compatibility with Autoconf 2.13 prevents this.

AM_PROG_GCJ

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

AM_PROG_INSTALL_STRIP

This is used to find a version of `install` which 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`.

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’.

AM_TYPE_PTRDIFF_T

Define ‘`HAVE_PTRDIFF_T`’ if the type ‘`ptrdiff_t`’ is defined in ‘`<stddef.h>`’.

AM_WITH_DMALLOC

Add support for the `dmalloc` (`ftp://ftp.letters.com/src/dmalloc/dmalloc.tar.gz`) package. If the user configures 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’.

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

A file of macros should be a series of `AC_DEFUN`’s. The `aclocal` programs also understands `AC_REQUIRE`, so it is safe to put each macro in a separate file. See section “Prerequisite Macros” in *The Autoconf Manual*, and section “Macro Definitions” in *The Autoconf Manual*.

A macro file’s name should end in ‘.m4’. Such files should be installed in ‘\$(datadir)/aclocal’.

6 The top-level ‘Makefile.am’

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` macro holds a list of subdirectories in which building of various sorts can occur. Many targets (e.g. `all`) in the generated ‘Makefile’ will run 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 which do not use Automake (such as `gettext`). The directories mentioned in `SUBDIRS` must be direct children of the current directory. For instance, you cannot put ‘src/subdir’ into `SUBDIRS`.

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.0 README-alpha
SUBDIRS = doc intl po src tests
```

It is possible to override the `SUBDIRS` variable if, like in the case of GNU `Inetutils`, you want to only build a subset of the entire package. In your ‘Makefile.am’ include:

```
SUBDIRS = @MY_SUBDIRS@
```

Then in your ‘configure.in’ you can specify:

```
MY_SUBDIRS="src doc lib po"
AC_SUBST(MY_SUBDIRS)
```

(Note that we don’t use the variable name `SUBDIRS` in our ‘configure.in’; that would cause Automake to believe that every ‘Makefile.in’ should recurse into the listed subdirectories.)

The upshot of this is that Automake is tricked into building the package to take the subdirs, but doesn’t actually bind that list until `configure` is run.

Although the `SUBDIRS` macro can contain configure substitutions (e.g. `@DIRS@`); Automake itself does not actually examine the contents of this variable.

If `SUBDIRS` is defined, then your `configure.in` must include `AC_PROG_MAKE_SET`. 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 use of `SUBDIRS` is not restricted to just the top-level `Makefile.am`. Automake can be used to construct packages of arbitrary depth.

By default, Automake generates `Makefiles` which work depth-first (`postfix`). 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. All `clean` targets are run in reverse order of build targets.

Sometimes, such as when running `make dist`, you want all possible subdirectories to be examined. In this case Automake will use `DIST_SUBDIRS`, instead of `SUBDIRS`, to determine where to recurse. This variable will also be used when the user runs `distclean` or `maintainer-clean`. It should be set to the full list of subdirectories in the project. If this macro is not set, Automake will attempt to set it for you.

7 An Alternative Approach to Subdirectories

If you've ever read Peter Miller's excellent paper, *Recursive Make Considered Harmful* (<http://www.pcug.org.au/~millerp/rmch/recu-make-cons-harm.html>), the preceding section 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 support² to enable you to write a single `Makefile.am` for a complex multi-directory package.

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

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

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

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

² We believe. This work is new and there are probably warts. See Chapter 1 [Introduction], page 1, for information on reporting bugs.

8 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.in’, 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 which are passed verbatim to `aclocal`. This macro is only useful in the top-level ‘Makefile.am’.

9 Building Programs and Libraries

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

9.1 Building a program

9.1.1 Introductory blathering

In a directory containing source that gets built into a program (as opposed to a library), 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 which 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`’; that is, the default is to compile a single C file whose base name is the name of the program itself. (This is a terrible default but we are stuck with it for historical reasons.)

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 9.7 [Yacc and Lex], page 26.

9.1.2 Conditional compilations

You can't put a configure substitution (e.g., '@FOO@') 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.

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 which are only conditionally built should be listed in the appropriate 'EXTRA_' variable. For instance, if 'hello-linux.c' were conditionally included in `hello`, the 'Makefile.am' would contain:

```
EXTRA_hello_SOURCES = hello-linux.c
```

In this case, 'hello-linux.o' would be added, via a 'configure' substitution, to `hello_LDADD` in order to cause it to be built and linked in.

An often simpler way to compile source files conditionally is to use Automake conditionals. For instance, you could use this construct to conditionally use 'hello-linux.c' or 'hello-generic.c' as the basis for your program 'hello':

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

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.

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.

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`.

Of course you can use Automake conditionals to determine the programs to be built.

9.1.3 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 actually can be used to add any options to the linker command line.

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 ‘-1’, ‘-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 which 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, ‘-1’, ‘-L’, ‘-dlopen’ and ‘-dlpreopen’ options removed. The configure substitutions that are left in are only ‘@LIBOBJ@’ and ‘@ALLOCA@’; these are left because it is known that they will not cause an invalid value for ‘*prog_DEPENDENCIES*’ to be generated.

9.2 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 9.3 [A Shared Library], page 22, 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
```

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 ‘`liblob.a`’ is ‘`liblob_a_SOURCES`’, not ‘`liblob.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 = @LIBOBJ@ @ALLOCA@
```

In addition, sources for extra objects that will not exist until configure-time must be added to the `BUILT_SOURCES` variable (see Section 10.4 [Sources], page 35).

9.3 Building a Shared Library

Building shared libraries 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.

Automake uses Libtool to build libraries declared with the ‘LTLIBRARIES’ primary. Each ‘_LTLIBRARIES’ variable is a list of shared libraries to build. For instance, to create a library named ‘libgettext.a’ and its corresponding shared libraries, and install them in ‘libdir’, write:

```
lib_LTLIBRARIES = libgettext.la
```

Note that shared libraries *must* be installed, so `check_LTLIBRARIES` is not allowed. However, `noinst_LTLIBRARIES` is allowed. This feature should be used for libtool “convenience libraries”.

For each library, the ‘*library_LIBADD*’ variable contains the names of extra libtool objects (‘.lo’ files) to add to the shared library. The ‘*library_LDFLAGS*’ variable contains any additional libtool flags, such as ‘-version-info’ or ‘-static’.

Where an ordinary library might include @LIBOBJ@, a libtool library must use @LTLIBOBJ@. This is required because the object files that libtool operates on do not necessarily end in ‘.o’. The libtool manual contains more details on this topic.

For libraries installed in some directory, Automake will automatically supply the appropriate ‘-rpath’ option. However, for libraries determined at configure time (and thus mentioned in EXTRA_LTLIBRARIES), Automake does not know the eventual installation directory; for such libraries you must add the ‘-rpath’ option to the appropriate ‘_LDFLAGS’ variable by hand.

Ordinarily, Automake requires that a shared library’s name start with ‘lib’. However, if you are building a dynamically loadable module then you might wish to use a “nonstandard” name. In this case, put `-module` into the ‘_LDFLAGS’ variable.

See section “The Libtool Manual” in *The Libtool Manual*, for more information.

9.4 Program and Library Variables

Associated with each program are a collection of variables which 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 which 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 9.12 [Support for Other Languages], page 32). 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 17 [Options], page 45).

`'EXTRA_maude_SOURCES'`

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

This variable also supports `dist_` and `nodist_` prefixes, e.g., `'nodist_EXTRA_maude_SOURCES'`.

`'maude_AR'`

A static library is created by default by invoking `$(AR) cru` 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 which should go into a library. For instance, the SGI C++ compiler likes this macro set like so:

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

`'maude_LIBADD'`

Extra objects can be added to a static library using the `'_LIBADD'` variable. This should be used for objects determined by `configure`. Note that `'_LIBADD'` is not used for shared libraries; there you must use `'_LDADD'`.

³ There are other, more obscure reasons reasons for this limitation as well.

`'maude_LDADD'`

Extra objects can be added to a shared library or a program by listing them in the `'_LDADD'` variable. This should be used for objects determined by `configure`.

`'_LDADD'` is 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.in'` 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_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 which 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_CFLAGS'`

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. The flags are `'_CFLAGS'`, `'_CXXFLAGS'`, `'_OBJCFLAGS'`, `'_YFLAGS'`, `'_ASFLAGS'`, `'_FFLAGS'`, `'_RFLAGS'`, and `'_GCJFLAGS'`.

When using a per-program 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'`.

In compilations with per-program 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)
```

`'maude_DEPENDENCIES'`

It is also occasionally useful to have a program depend on some other target which 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'`, with most `configure`

substitutions, ‘-l’, ‘-L’, ‘-dlopen’ and ‘-dlpreopen’ options removed. The configure substitutions that are left in are only ‘@LIBOBJ@’ and ‘@ALLOCA@’; these are left because it is known that they will not cause an invalid value for ‘_DEPENDENCIES’ to be generated.

‘maude_SHORTNAME’

On some platforms the allowable file names are very short. In order to support these systems and per-program compilation flags at the same time, Automake allows you to set a “short name” which will influence how intermediate object files are named. For instance, if you set ‘maude_SHORTNAME’ to ‘m’, then in the above per-program compilation flag example 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.

9.5 Special handling for LIBOBJ and ALLOCA

Automake explicitly recognizes the use of @LIBOBJ@ and @ALLOCA@, and uses this information, plus the list of LIBOBJ files derived from ‘configure.in’ to automatically include the appropriate source files in the distribution (see Chapter 15 [Dist], page 42). These source files are also automatically handled in the dependency-tracking scheme; see Section 9.14 [Dependencies], page 33.

@LIBOBJ@ and @ALLOCA@ are specially recognized in any ‘_LDADD’ or ‘_LIBADD’ variable.

9.6 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 which Automake itself defines:

AM_CPPFLAGS

The contents of this macro are passed to every compilation which 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_HEADER or AM_CONFIG_HEADER). You can disable the default ‘-I’ options using the ‘nostdinc’ option.

INCLUDES This does the same job as ‘AM_CPPFLAGS’. It is an older name for the same functionality. This macro is deprecated; we suggest using ‘AM_CPPFLAGS’ instead.

AM_CFLAGS

This is the variable which 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 filename is appended to form the complete command line.
- 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.

9.7 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.in` 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.

Similarly, if a `lex` source file is seen, then your `configure.in` must define the variable `LEX`. You can use `AC_PROG_LEX` to do this (see section “Particular Program Checks” in *The Autoconf Manual*). Automake's `lex` support also requires that you use the `AC_DECL_YTEXT` macro—automake needs to know the value of `LEX_OUTPUT_ROOT`. This is all handled for you if you use the `AM_PROG_LEX` macro (see Section 5.4 [Macros], page 14).

When `yacc` 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. Automake uses a small program called `ylwrap` to run `yacc` (or `lex`) in a sub-directory. This is necessary because `yacc`'s output filename 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` (see section “Finding ‘configure’ Input” in *The Autoconf Manual*), or the current directory if that macro is not used in `configure.in`.

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 yynname c_yynname
#define yyrule c_yyrule
```

For each define, replace the 'c_' prefix with whatever you like. These defines work for bison, byacc, and traditional yaccs. If you find a parser generator that uses a symbol not covered here, please report the new name so it can be added to the list.

9.8 C++ Support

Automake includes full support for C++.

Any package including C++ code must define the output variable ‘CXX’ in ‘`configure.in`’; 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.

9.9 Assembly Support

Automake includes some support for assembly code.

The variable **AS** 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 value of **ASFLAGS** is passed to the compilation.

You are required to set **AS** and **ASFLAGS** via ‘`configure.in`’. The autoconf macro `AM_PROG_AS` will do this for you. Unless they are already set, it simply sets **AS** to the C compiler and **ASFLAGS** to the C compiler flags.

9.10 Fortran 77 Support

Automake includes full support for Fortran 77.

Any package including Fortran 77 code must define the output variable ‘F77’ in ‘`configure.in`’; the simplest way to do this is to use the `AC_PROG_F77` macro (see section “Particular Program Checks” in *The Autoconf Manual*). See Section 9.10.4 [Fortran 77 and Autoconf], page 31.

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

F77 The name of the Fortran 77 compiler.

FFLAGS Any flags to pass to the Fortran 77 compiler.

AM_FFLAGS
 The maintainer’s variant of **FFLAGS**.

RFLAGS Any flags to pass to the Ratfor compiler.

AM_RFLAGS

The maintainer's variant of **RFLAGS**.

F77COMPILE

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

FLINK The command used to actually link a pure Fortran 77 program or shared library.

Automake can handle preprocessing Fortran 77 and Ratfor source files in addition to compiling them⁴. Automake also contains some support for creating programs and shared libraries that are a mixture of Fortran 77 and other languages (see Section 9.10.3 [Mixing Fortran 77 With C and C++], page 29).

These issues are covered in the following sections.

9.10.1 Preprocessing Fortran 77

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

```
'.F'      $(F77) -F $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_
           FFLAGS) $(FFLAGS)
'.r'      $(F77) -F $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)
```

9.10.2 Compiling Fortran 77 Files

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

```
'.f'      $(F77) -c $(AM_FFLAGS) $(FFLAGS)
'.F'      $(F77) -c $(DEFS) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) $(AM_
           FFLAGS) $(FFLAGS)
'.r'      $(F77) -c $(AM_FFLAGS) $(FFLAGS) $(AM_RFLAGS) $(RFLAGS)
```

9.10.3 Mixing Fortran 77 With C and C++

Automake currently provides *limited* support for creating programs and shared libraries that are a mixture of Fortran 77 and C and/or C++. However, there are many other issues related to mixing Fortran 77 with other languages that are *not* (currently) handled by Automake, but that are handled by other packages⁵.

⁴ Much, if not most, of the information in the following sections pertaining to preprocessing Fortran 77 programs was taken almost verbatim from section "Catalogue of Rules" in *The GNU Make Manual*.

⁵ For example, the `cfortran` package (<http://www-zeus.desy.de/~burow/cfortran/>) addresses all of these inter-language issues, and runs under nearly all Fortran 77, C and C++ compilers on nearly all platforms. However, `cfortran` is not yet Free Software, but it will be in the next major release.

Automake can help in two ways:

1. Automatic selection of the linker depending on which combinations of source code.
2. Automatic selection of the appropriate linker flags (e.g. '-L' and '-l') to pass to the automatically selected linker in order to link in the appropriate Fortran 77 intrinsic and run-time libraries.

These extra Fortran 77 linker flags are supplied in the output variable `FLIBS` by the `AC_F77_LIBRARY_LDFLAGS` Autoconf macro supplied with newer versions of Autoconf (Autoconf version 2.13 and later). See section “Fortran 77 Compiler Characteristics” in *The Autoconf*.

If Automake detects that a program or shared library (as mentioned in some `_PROGRAMS` or `_LTLIBRARIES` primary) contains source code that is a mixture of Fortran 77 and C and/or C++, then it requires that the macro `AC_F77_LIBRARY_LDFLAGS` be called in `configure.in`, and that either `$(FLIBS)` or `@FLIBS@` appear in the appropriate `_LDADD` (for programs) or `_LIBADD` (for shared libraries) variables. It is the responsibility of the person writing the `Makefile.am` to make sure that `$(FLIBS)` or `@FLIBS@` appears in the appropriate `_LDADD` or `_LIBADD` variable.

For example, consider the following `Makefile.am`:

```
bin_PROGRAMS = foo
foo_SOURCES  = main.cc foo.f
foo_LDADD    = libfoo.la @FLIBS@

pkglib_LTLIBRARIES = libfoo.la
libfoo_la_SOURCES  = bar.f baz.c zardoz.cc
libfoo_la_LIBADD   = $(FLIBS)
```

In this case, Automake will insist that `AC_F77_LIBRARY_LDFLAGS` is mentioned in `configure.in`. Also, if `@FLIBS@` hadn't been mentioned in `foo_LDADD` and `libfoo_la_LIBADD`, then Automake would have issued a warning.

9.10.3.1 How the Linker is Chosen

The following diagram demonstrates under what conditions a particular linker is chosen by Automake.

For example, if Fortran 77, C and C++ source code were to be 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'`.

source code	Linker		
	C	C++	Fortran
C	x		
C++		x	
Fortran			x
C + C++		x	
C + Fortran			x
C++ + Fortran		x	
C + C++ + Fortran		x	

9.10.4 Fortran 77 and Autoconf

The current Automake support for Fortran 77 requires a recent enough version Autoconf that also includes support for Fortran 77. Full Fortran 77 support was added to Autoconf 2.13, so you will want to use that version of Autoconf or later.

9.11 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.in'`; the variable `'GCJFLAGS'` must also be defined somehow (either in `'configure.in'` 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 macro, and not the user macro `'GCJFLAGS'`, should be used.

`gcj` can be used to compile `'java'`, `'class'`, `'zip'`, or `'jar'` files.

9.12 Support for Other Languages

Automake currently only includes full support for C, C++ (see Section 9.8 [C++ Support], page 28), Fortran 77 (see Section 9.10 [Fortran 77 Support], page 28), and Java (see Section 9.11 [Java Support], page 32). There is only rudimentary support for other languages, support for which will be improved based on user demand.

Some limited support for adding your own languages is available via the suffix rule handling; see Section 18.2 [Suffixes], page 47.

9.13 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 17 [Options], page 45) 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.in'` must call the macro `AM_C_PROTOTYPES` (see Section 5.4 [Macros], page 14).

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'` subdirs. 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.

Files mentioned in `LIBBOBJS` which need de-ANSI-fication will not be automatically handled. That's because `configure` will generate an object name like `'regex.o'`, while `make` will be looking for `'regex.o'` (when de-ANSI-fying). Eventually this problem will be fixed via `autoconf` magic, but for now you must put this code into your `'configure.in'`, just before the `AC_OUTPUT` call:

```
# This is necessary so that .o files in LIBBOBJS are also built via
# the ANSI2KNR-filtering rules.
LIBBOBJS='echo $LIBBOBJS|sed 's/\.o /\$U.o /g;s/\.o$/\$U.o/''
```

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.

9.14 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`¹ 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, 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`.

9.15 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'`.

¹ See <http://sources.redhat.com/automake/dependencies.html> for more information on the history and experiences with automatic dependency tracking in `Automake`

Automake provides mostly-transparent support for this. Unfortunately the support isn't completely transparent; if you want your package to support these platforms then you must assist.

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. EXEEXT

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.in`' 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 target with the same name as 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 17 [Options], page 45) will disable this feature. This works in a fairly ugly way; if `no-exeext` is seen, then the presence of a target named `foo` in '`Makefile.am`' will override an automake-generated target of the form `foo$(EXEEXT)`. Without the `no-exeext` option, this use will give an error.

10 Other Derived Objects

Automake can handle derived objects which 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.

10.1 Executable Scripts

It is possible to define and install programs which 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 14 [Clean], page 42).

The `automake` program itself is a Perl script that is generated at configure time from '`automake.in`'. Here is how this is handled:

```
bin_SCRIPTS = automake
```

Since `automake` appears in the `AC_OUTPUT` macro, a target for it is automatically generated.

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

10.2 Header files

Header files are specified by the ‘HEADERS’ family of variables. Generally header files are not installed, so the `noinst_HEADERS` variable will be the most used.²

All header files must be listed somewhere; missing ones will not appear in the distribution. Often it is clearest to list uninstalled headers with the rest of the sources for a program. See Section 9.1 [A Program], page 19. Headers listed in a ‘_SOURCES’ variable need not be listed in any ‘_HEADERS’ variable.

Headers can be installed in `includedir`, `oldincludedir`, or `pkgincludedir`.

10.3 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 installs its auxiliary data files:

```
pkgdata_DATA = clean-kr.am clean.am ...
```

10.4 Built sources

Occasionally a file which would otherwise be called ‘`source`’ (e.g. a C ‘`.h`’ file) is actually derived from some other file. Such files should be listed in the `BUILT_SOURCES` variable.

`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.

A source file listed in `BUILT_SOURCES` is created before the other `all` targets are made. However, such a source file is not compiled unless explicitly requested by mentioning it in some other ‘_SOURCES’ variable.

So, for instance, if you had header files which were created by a script run at build time, then you would list these headers in `BUILT_SOURCES`, to ensure that they would be built before any other compilations (perhaps ones using these headers) were started.

² However, for the case of a non-installed header file that is actually used by a particular program, we recommend listing it in the program’s ‘_SOURCES’ variable instead of in `noinst_HEADERS`. We believe this is more clear.

11 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.

11.1 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.in` must run `AM_PATH_LISPDIR` (see Section 5.4 [Macros], page 14).

By default Automake will byte-compile all Emacs Lisp source files using the Emacs found by `AM_PATH_LISPDIR`. If you wish to avoid byte-compiling, simply define the variable `ELCFILES` to be empty. 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 leave it enabled by default. It is probably better for sites with strange setups to cope for themselves than to make the installation less nice for everybody else.

11.2 Gettext

If `AM_GNU_GETTEXT` is seen in `configure.in`, 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`.

11.3 Libtool

Automake provides support for GNU Libtool (see section “Introduction” in *The Libtool Manual*) with the `LTLIBRARIES` primary. See Section 9.3 [A Shared Library], page 22.

11.4 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, `.class` files are not included in the distribution.

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 which 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 which is used to set the **CLASSPATH** environment variable on the `javac` command line. (In the future we will probably handle class path setting differently.)

11.5 Python

Automake provides support for Python modules. Automake will turn on Python support if the **AM_PATH_PYTHON** macro is used in ‘`configure.in`’. The ‘**PYTHON**’ primary is used to hold a list of ‘`.py`’ files. Possible prefixes for this primary are ‘`python_`’ and ‘`noinst_`’. Note that if **python_PYTHON** is defined, then ‘`configure.in`’ must run **AM_PATH_PYTHON**. Python source files are included in the distribution by default.

AM_PATH_PYTHON takes a single optional argument. This argument, if present, is the minimum version of Python which can be used for this package. If the version of Python found on the system is older than the required version, then **AM_PATH_PYTHON** will cause an error.

AM_PATH_PYTHON creates several output variables based on the Python installation found during configuration.

- PYTHON** The name of the Python executable.
- 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 `$prefix`. This term may be used in future work which needs the contents of Python’s `sys.prefix`, but general consensus is to always use the value from `configure`.
- PYTHON_EXEC_PREFIX**
The string `$exec_prefix`. This term may be used in future work which 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` which 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 which is defined as ‘`$(pyexecdir)/$(PACKAGE)`’.

By default Automake will byte-compile all Python source files to both ‘`.pyc`’ and ‘`.pyo`’ forms. If you wish to avoid generating the optimized byte-code files, simply define the variable `PYOFILES` to be empty. Similarly, if you don’t wish to generate the standard byte-compiled files, define the variable `PYCFILES` to be empty.

12 Building documentation

Currently Automake provides support for Texinfo and man pages.

12.1 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` macro is most commonly used here. Any Texinfo source file must end in the ‘`.texi`’, ‘`.txi`’, or ‘`.texinfo`’ extension. We recommend ‘`.texi`’ for new manuals.

If the ‘`.texi`’ file `@includes` ‘`version.texi`’, then that file will be automatically generated. The file ‘`version.texi`’ defines four Texinfo macros you can reference:

EDITION

VERSION Both of these macros 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` program; this program 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.

When an info file is rebuilt, the program named by the `MAKEINFO` variable is used to invoke it. If the `makeinfo` program is found on the system then it will be used by default; otherwise `missing` will be used instead. The flags in the variables `MAKEINFOFLAGS` and `AM_MAKEINFOFLAGS` will be passed to the `makeinfo` invocation; the first of these is intended for use by the user (see Section 2.5 [User Variables], page 4) and the second by the `'Makefile.am'` writer.

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. However, if you used `AC_CONFIG_AUX_DIR` in `'configure.in'` (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.

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
```

The option `'no-texinfo.tex'` can be used to eliminate the requirement for `'texinfo.tex'`. Use of the variable `TEXINFO_TEX` is preferable, however, because that allows the `dvi` target to still work.

Automake generates an `install-info` target; some people apparently use this. By default, info pages are installed by `'make install'`. This can be prevented via the `no-installinfo` option.

12.2 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` macro 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, e.g. `'foo.1'`, when installing the file. Au-

tomake 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.

13 What Gets Installed

13.1 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 which 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`.

13.2 The two parts of install

Automake generates separate `install-data` and `install-exec` targets, in case the installer is installing on multiple machines which 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'` (e.g. `'data_DATA'`) are installed by `'install-data'`.

Variables using the standard directory prefixes `'bin'`, `'sbin'`, `'libexec'`, `'sysconf'`, `'localstate'`, `'lib'`, or `'pkglib'` (e.g. `'bin_PROGRAMS'`) 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'`.

13.3 Extending installation

It is possible to extend this mechanism by defining an `install-exec-local` or `install-data-local` target. If these targets 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.

13.4 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:

```
make DESTDIR=/tmp/staging install
```

This 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'`.

13.5 Rules for the user

Automake also generates an `uninstall` target, an `installdirs` target, and an `install-strip` target.

Automake supports `uninstall-local` and `uninstall-hook`. There is no notion of separate uninstalls for “exec” and “data”, as that does not make sense.

Note that `uninstall` is not meant as a replacement for a real packaging tool.

14 What Gets Cleaned

The GNU Makefile Standards specify a number of different clean rules. 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`.

As the GNU Standards aren't always explicit as to which files should be removed by which target, we've adopted a heuristic which 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, then `maintainer-clean` should delete it.

We recommend that you follow this same set of heuristics in your `'Makefile.am'`.

15 What Goes in a Distribution

15.1 Basics of distribution

The `dist` target in the generated `'Makefile.in'` can be used to generate a gzip'd tar file for distribution. The tar file is named based on the `'PACKAGE'` and `'VERSION'` variables; more precisely it 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 which, if present in the current directory, are automatically included. This list is printed by `'automake --help'`. Also, files which are read by `configure` (i.e. the source files corresponding to the files specified in the `AC_OUTPUT` invocation) are automatically distributed.

Still, sometimes there are files which 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.

15.2 Fine-grained distribution control

Sometimes you need tighter control over what does *not* go into the distribution; for instance you might have source files which are generated and which 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
```

15.3 The dist hook

Another way to use this is for removing unnecessary files that get recursively included by specifying a directory in `EXTRA_DIST`:

```
EXTRA_DIST = doc

dist-hook:
rm -rf 'find $(distdir)/doc -name CVS'
```

If you define `SUBDIRS`, Automake will recursively include the subdirectories in the distribution. If `SUBDIRS` is defined conditionally (see Chapter 20 [Conditionals], page 48), 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.

Occasionally it is useful to be able to change the distribution before it is packaged up. If the `dist-hook` target 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
```

15.4 Checking the distribution

Automake also generates a `distcheck` target which can be of help to ensure that a given distribution will actually work. `distcheck` makes a distribution, and then tries to do a `VPATH` build.

If the target `distcheck-hook` is defined in your `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.

15.5 The types of distributions

By default Automake generates a `.tar.gz` file when asked to create a distribution. However, some projects prefer different packaging formats. Automake accomodates most of these using options; Chapter 17 [Options], page 45.

Automake also generates a `dist-all` target which can be used to make all the requested packaged distributions at once.

16 Support for test suites

Automake supports two forms of test suites.

16.1 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` target (see Chapter 15 [Dist], page 42).

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.

16.2 DejaGNU Tests

If `dejagnu` (`ftp://prep.ai.mit.edu/pub/gnu/dejagnu-1.3.tar.gz`) appears in `AUTOMAKE_OPTIONS`, then a `dejagnu`-based test suite is assumed. The variable `DEJATOOL`

is a list of names which 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.

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

17 Changing Automake's Behavior

Various features of Automake can be controlled by options in the ‘`Makefile.am`’. Such options are listed in a special variable named `AUTOMAKE_OPTIONS`. Currently understood options are:

`gnits`

`gnu`

`foreign`

`cygnus`

Set the strictness as appropriate. The `gnits` option also implies `readme-alpha` and `check-news`.

`ansi2knr`

`path/ansi2knr`

Turn on automatic de-ANSI-fication. See Section 9.13 [ANSI], page 32. 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 16 [Tests], page 44.

`dist-bzip2`

Generate a `dist-bzip2` target as well as the ordinary `dist` target. This new target will create a `bzip2` tar archive of the distribution. `bzip2` archives are frequently smaller than even gzipped archives.

`dist-shar`

Generate a `dist-shar` target as well as the ordinary `dist` target. This new target will create a `shar` archive of the distribution.

`dist-zip`

Generate a `dist-zip` target as well as the ordinary `dist` target. This new target will create a `zip` archive of the distribution.

dist-tarZ

Generate a **dist-tarZ** target as well as the ordinary **dist** target. This new target will create a compressed tar archive of the distribution.

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 9.14 [Dependencies], page 33. In this case the effect is to effectively disable automatic dependency tracking.

no-exeext

If your **Makefile.am** defines a target **foo**, it will override 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 which 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.

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**.

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.

Unrecognized options are diagnosed by **automake**.

18 Miscellaneous Rules

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

18.1 Interfacing to etags

Automake will generate rules to generate 'TAGS' files for use with GNU Emacs under some circumstances.

If any C, C++ or Fortran 77 source code or headers are present, then `tags` and `TAGS` targets will be generated for the directory.

At the topmost directory of a multi-directory package, a `tags` target file will be generated which, when run, will generate a 'TAGS' file that includes by reference all 'TAGS' files from subdirectories.

The `tags` target will also be generated if the variable `ETAGS_ARGS` is defined. This variable is intended for use in directories which contain taggable source that `etags` does not understand.

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 filenames to 'ETAGS_ARGS', you will probably also want to set 'TAGS_DEPENDENCIES'. The contents of this variable are added directly to the dependencies for the `tags` target.

Automake will also generate an ID target which will run `mkid` on the source. This is only supported on a directory-by-directory basis.

Automake also supports the GNU Global Tags program (<http://www.gnu.org/software/global/>). The `GTAGS` target runs Global Tags automatically and puts the result in the top build directory. The variable `GTAGS_ARGS` holds arguments which are passed to `gtags`.

18.2 Handling new file extensions

It is sometimes useful to introduce a new implicit rule to handle a file type that Automake does not know about. If this is done, you must notify GNU Make of the new suffixes. This can be done by putting a list of new suffixes in the `SUFFIXES` variable.

For instance, suppose you had a compiler which could compile '.foo' files to '.o' files. Then you would add '.foo' to your suffix list:

```
SUFFIXES = .foo
```

Then you could directly use a '.foo' file in a '_SOURCES' variable and expect the correct results:

```
bin_PROGRAMS = doit
doit_SOURCES = doit.foo
```

Any given `SUFFIXES` go at the start of the generated suffixes list, followed by automake generated suffixes not already in the list.

18.3 Support for Multilibs

Automake has support for an obscure feature called multilibs. A *multilib* is a library which 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.

19 Include

Automake supports an `include` directive which 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 which is found relative to the current source directory.

```
include $(top_srcdir)/file
```

Include a fragment which 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.

20 Conditionals

Automake supports a simple type of conditionals.

Before using a conditional, you must define it by using `AM_CONDITIONAL` in the `configure.in` file (see Section 5.4 [Macros], page 14).

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’ which 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 which the user provides to the `configure` script. Here is an example of how to write a conditional which 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 9.1 [A Program], page 19).

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 which 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.

21 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'`, `'COPYING'`, `'AUTHORS'`, and `'ChangeLog'` 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 targets; 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 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.

22 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.

23 When Automake Isn’t Enough

Automake’s implicit copying semantics means that many problems can be worked around by simply adding some `make` targets and rules to ‘Makefile.in’. Automake will ignore these additions.

There are some caveats to doing this. Although you can overload a target already used by Automake, it is often inadvisable, particularly in the topmost directory of a package with subdirectories. However, various useful targets have a ‘-local’ version you can specify in your ‘Makefile.in’. Automake will supplement the standard target with these user-supplied targets.

The targets that support a local version are `all`, `info`, `dvi`, `check`, `install-data`, `install-exec`, `uninstall`, 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 install a file in ‘/etc’:

```
install-data-local:
    $(INSTALL_DATA) $(srcdir)/afile /etc/afile
```

Some targets also have a way to run another target, 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`, `dist`, and `distcheck`.

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

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

24 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.

Macro and Variable Index

-		AM_INIT_AUTOMAKE	11
_LDADD	20	AM_JAVACFLAGS	37
_LDFLAGS	21	AM_MAINTAINER_MODE	13
_LIBADD	21	AM_MAKEINFOFLAGS	39
_SOURCES	19	AM_PATH_LISPDIR	15
_TEXINFOS	39	AM_PROG_GCJ	16
		AM_RFLAGS	29
A		AM_RUNTESTFLAGS	45
AC_CANONICAL_HOST	12	AM_WITH_REGEX	12
AC_CANONICAL_SYSTEM	12	AS	28
AC_CHECK_PROG	13	ASFLAGS	28
AC_CHECK_PROGS	13	AUTOMAKE_OPTIONS	32, 33, 45
AC_CHECK_TOOL	12, 13		
AC_CHECK_TOOL([STRIP], [strip])	11	B	
AC_CONFIG_AUX_DIR	11	bin_PROGRAMS	19
AC_CONFIG_HEADER	11	bin_SCRIPTS	35
AC_DECL_YTEXT	13	build_alias	12
AC_F77_LIBRARY_LDFLAGS	13	BUILT_SOURCES	35
AC_FUNC_ALLOCA	12		
AC_FUNC_FNMATCH	12	C	
AC_FUNC_GETLOADAVG	12	check_LTLIBRARIES	22
AC_FUNC_MEMCMP	12	CLASSPATH_ENV	37
AC_FUNC_MKTIME	12	CLEANFILES	42
AC_OUTPUT	11	COMPILE	26
AC_PATH_PROG	13	CXX	28
AC_PATH_PROGS	13	CXXCOMPILE	28
AC_PATH_XTRA	11	CXXFLAGS	28
AC_PROG_CXX	12	CXXLINK	28
AC_PROG_F77	12		
AC_PROG_LEX	13	D	
AC_PROG_LIBTOOL	13	DATA	4, 35
AC_PROG_RANLIB	12	data_DATA	35
AC_PROG_YACC	13	DEJATOOL	45
AC_REPLACE_FUNCS	12	DESTDIR	41
AC_REPLACE_GNU_GETOPT	12	dist_	43
AC_STRUCT_ST_BLOCKS	12	DIST_SUBDIRS	43
AC_SUBST	13	DISTCLEANFILES	42
AM_C_PROTOTYPES	13, 15, 32		
AM_CFLAGS	25	E	
AM_CONDITIONAL	48	ELCFILES	36
AM_CONFIG_HEADER	14	ETAGS_ARGS	47
AM_CPPFLAGS	25	EXPECT	45
am_cv_sys_posix_termios	16	EXTRA_DIST	43
AM_CXXFLAGS	28	EXTRA_PROGRAMS	20
AM_FFLAGS	28		
AM_FUNC_ERROR_AT_LINE	15		
AM_FUNC_OBSTACK	15		
AM_FUNC_STRTOD	12, 15		
AM_GCJFLAGS	32		
AM_GNU_GETTEXT	13		
AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL	15		

F

F77	28
F77COMPILE	29
FFLAGS	28
FLINK	29

G

GCJFLAGS	32
GTags_ARGS	47

H

HAVE_PTRDIFF_T	16
HEADERS	4, 35
host_alias	12
host_triplet	12

I

include_HEADERS	35
INCLUDES	25
info_TEXINFOS	38

J

JAVA	4
JAVAC	37
JAVACFLAGS	37
JAVAROOT	37

L

LDADD	20
LDFLAGS	25
lib_LIBRARIES	21
lib_LTLIBRARIES	22
LIBADD	21
libexec_PROGRAMS	19
libexec_SCRIPTS	35
LIBOBJS	12
LIBRARIES	4
LINK	26
LISP	4, 36
lisp_LISP	36
localstate_DATA	35

M

MAINTAINERCLEANFILES	42
MAKE	18
MAKEFLAGS	18
MAKEINFO	39
MAKEINFOFLAGS	39
man_MANS	39
MANS	4, 39
MOSTLYCLEANFILES	42

N

nodist_	43
noinst_HEADERS	35
noinst_LIBRARIES	21
noinst_LISP	36
noinst_LTLIBRARIES	22
noinst_PROGRAMS	19
noinst_SCRIPTS	35

O

oldinclude_HEADERS	35
--------------------	----

P

PACKAGE	3, 42
pkgdata_DATA	35
pkgdata_SCRIPTS	35
pkgdatadir	3
pkginclude_HEADERS	35
pkgincludedir	3
pkglib_LIBRARIES	21
pkglib_LTLIBRARIES	22
pkglib_PROGRAMS	19
pkglibdir	3
pkgpyexecdir	38
pkgpythondir	38
PROGRAMS	3, 4
ptrdiff_t	16
PYCFILES	38
pyexecdir	38
PYFILES	38
PYTHON	4, 37
PYTHON_EXEC_PREFIX	37
PYTHON_PLATFORM	38
PYTHON_PREFIX	37
PYTHON_VERSION	37
pythondir	38

R

RFLAGS	28
RUNTEST	45
RUNTESTDEFAULTFLAGS	45
RUNTESTFLAGS	45

S

sbin_PROGRAMS	19
sbin_SCRIPTS	35
SCRIPTS	4, 34
sharedstate_DATA	35
SOURCES	19
SUBDIRS	17
SUFFIXES	47
sysconf_DATA	35

T

TAGS_DEPENDENCIES	47
target_alias	12

TESTS	44
TESTS_ENVIRONMENT	44
TEXINFO_TEX	39
TEXINFOS	4, 38, 39

V

VERSION	42
---------------	----

W

WITH_DMALLOC	16
WITH_REGEX	16

X

XFAIL_TESTS	44
-------------------	----

Y

YACC	13
------------	----

General Index

#

(special Automake comment) 2

-

--acdir 14
 --add-missing 9
 --copy 9
 --cygnus 9
 --enable-debug, example 48
 --enable-maintainer-mode 13
 --force-missing 9
 --foreign 9
 --gnits 10
 --gnits, complete description 49
 --gnu 10
 --gnu, complete description 49
 --gnu, required files 49
 --help 10, 14
 --include-deps 10
 --libdir 9
 --no-force 10
 --output 14
 --output-dir 10
 --print-ac-dir 14
 --verbose 10, 14
 --version 10, 14
 --Werror 10
 --with-dmalloc 16
 --with-regex 16
 --Wno-error 10
 -a 9
 -c 9
 -f 9
 -hook targets 51
 -i 10
 -I 14
 -local targets 51
 -o 10
 -v 10

@

@ALLOCA@, special handling 25
 @LIBOBJS@, special handling 25
 @LTLIBOBJS@, special handling 22

-

_DATA primary, defined 35
 _DEPENDENCIES, defined 21
 _HEADERS primary, defined 35
 _JAVA primary, defined 36
 _LDFLAGS, defined 21
 _LIBADD primary, defined 21
 _LIBRARIES primary, defined 21
 _LISP primary, defined 36
 _LTLIBRARIES primary, defined 22
 _MANS primary, defined 39
 _PROGRAMS primary variable 3
 _PYTHON primary, defined 37
 _SCRIPTS primary, defined 34
 _SOURCES and header files 19
 _SOURCES primary, defined 19
 _TEXINFOS primary, defined 38

A

AC_OUTPUT, scanning 11
 acinclude.m4, defined 6
 aclocal program, introduction 6
 aclocal, extending 17
 aclocal, Invoking 13
 aclocal.m4, preexisting 6
 ACLOCAL_AMFLAGS 19
 Adding new SUFFIXES 47
 all 51
 all-local 51
 AM_INIT_AUTOMAKE, example use 5
 ansi2knr 32
 Automake constraints 1
 Automake options 9
 Automake requirements 1, 11
 Automake, invoking 9
 Automake, recursive operation 2
 Automatic dependency tracking 33
 Automatic linker selection 31
 Auxiliary programs 4
 Avoiding path stripping 18

B

BUGS, reporting 1
 BUILT_SOURCES, defined 35

C

C++ support	28
canonicalizing Automake macros	4
cfortran	29
check	51
check primary prefix, definition	3
check-local	51
check_LTLIBRARIES, not allowed	22
clean-local	51
Comment, special to Automake	2
Complete example	5
Conditional example, <code>-enable-debug</code>	48
Conditionals	48
config.guess	9
configure.in, from GNU Hello	6
configure.in, scanning	10
Constraints of Automake	1
cpio example	3
ctags Example	7
cvs-dist	1
cvs-dist, non-standard example	1
Cygnus strictness	50

D

DATA primary, defined	35
de-ANSI-fication, defined	32
dejagnu	45
depcomp	33
Dependency tracking	33
Dependency tracking, disabling	33
Disabling dependency tracking	33
dist	42
dist-bzip2	45
dist-hook	43, 51
dist-shar	45
dist-tarZ	46
dist-zip	45
distcheck	43
distclean-local	51
dmalloc, support for	16
dvi	51
dvi-local	51

E

E-mail, bug reports	1
EDITION Texinfo macro	38
else	49
endif	49
etags Example	7
Example conditional <code>-enable-debug</code>	48

Example of recursive operation	2
Example of shared libraries	22
Example, ctags and etags	7
Example, EXTRA_PROGRAMS	3
Example, GNU Hello	6
Example, handling Texinfo files	7
Example, mixed language	30
Example, regression test	7
Executable extension	33
Exit status 77, special interpretation	44
Expected test failure	44
Extending alocal	17
Extending list of installation directories	3
Extension, executable	33
Extra files distributed with Automake	9
EXTRA_, prepending	3
EXTRA_prog_SOURCES, defined	20
EXTRA_PROGRAMS, defined	3, 20

F

Files distributed with Automake	9
First line of Makefile.am	2
FLIBS, defined	30
foreign strictness	2
Fortran 77 support	28
Fortran 77, mixing with C and C++	29
Fortran 77, Preprocessing	29

G

Gettext support	36
gnits strictness	2
GNU Gettext support	36
GNU Hello, configure.in	6
GNU Hello, example	6
GNU make extensions	1
GNU Makefile standards	1

H

Header files in <code>_SOURCES</code>	19
HEADERS primary, defined	35
HEADERS, installation directories	35
Hello example	6
Hello, configure.in	6
hook targets	51
HP-UX 10, lex problems	16
HTML support, example	3

I

id	47
if	49
include	48
INCLUDES, example usage	7
Including Makefile fragment	48
info	46, 51
info-local	51
install	41
Install hook	41
Install, two parts of	41
install-data	41, 51
install-data-hook	51
install-data-local	41, 51
install-exec	41, 51
install-exec-hook	51
install-exec-local	41, 51
install-info	39, 46
install-info target	39
install-man	40, 46
install-man target	40
install-strip	42
install-strip and STRIP	11
Installation directories, extending list	3
Installation support	40
installdirs	42
Installing headers	35
Installing scripts	35
Invoking aclocal	13
Invoking Automake	9

J

JAVA primary, defined	36
JAVA restrictions	36
Java support	32

L

lex problems with HP-UX 10	16
lex, multiple lexers	26
LIBADD primary, defined	21
LIBRARIES primary, defined	21
Linking Fortran 77 with C and C++	29
LISP primary, defined	36
local targets	51
LTLIBRARIES primary, defined	22

M

Macros Automake recognizes	11
Macros, overriding	2
make check	44
make clean support	42
make dist	42
make distcheck	42
make install support	40
Make targets, overriding	1
Makefile fragment, including	48
Makefile.am, first line	2
MANS primary, defined	39
mdate-sh	38
Mixed language example	30
Mixing Fortran 77 with C and C++	29
Mixing Fortran 77 with C and/or C++	29
mostlyclean-local	51
Multiple configure.in files	9
Multiple lex lexers	26
Multiple yacc parsers	26

N

no-dependencies	33
no-installinfo	39
no-installman	40
no-texinfo.tex	39
nobase_	18
noinst primary prefix, definition	3
noinstall-info target	39
noinstall-man target	40
Non-GNU packages	2
Non-standard targets	1

O

Objects in subdirectory	23
Option, ansi2knr	45
Option, check-news	45
Option, cygnus	45
Option, dejagnu	45
Option, dist-bzip2	45
Option, dist-shar	45
Option, dist-tarZ	46
Option, dist-zip	45
Option, foreign	45
Option, gnits	45
Option, gnu	45
Option, no-dependencies	46
Option, no-exeext	46
Option, no-installinfo	46
Option, no-installman	46

Option, no-texinfo	46
Option, nostdinc	46
Option, readme-alpha	46
Option, version	46
Options, Automake	9
Overriding make macros	2
Overriding make targets	1
Overriding SUBDIRS	17

P

Path stripping, avoiding	18
pkgdatadir, defined	3
pkgincludedir, defined	3
pkglibdir, defined	3
POSIX termios headers	16
Preprocessing Fortran 77	29
Primary variable, DATA	35
Primary variable, defined	3
Primary variable, HEADERS	35
Primary variable, JAVA	36
Primary variable, LIBADD	21
Primary variable, LIBRARIES	21
Primary variable, LISP	36
Primary variable, LTLIBRARIES	22
Primary variable, MANS	39
Primary variable, PROGRAMS	3
Primary variable, PYTHON	37
Primary variable, SCRIPTS	34
Primary variable, SOURCES	19
Primary variable, TEXINFOS	38
prog_LDADD, defined	20
PROGRAMS primary variable	3
Programs, auxiliary	4
PROGRAMS, bindir	19
PYTHON primary, defined	37

R

Ratfor programs	29
README-alpha	50
Recognized macros by Automake	11
Recursive operation of Automake	2
regex package	16
Regression test example	7
Reporting BUGS	1
Requirements of Automake	11
Requirements, Automake	1
Restrictions for JAVA	36
rx package	16

S

Scanning configure.in	10
SCRIPTS primary, defined	34
SCRIPTS, installation directories	35
Selecting the linker automatically	31
Shared libraries, support for	22
SOURCES primary, defined	19
Special Automake comment	2
Strictness, defined	2
Strictness, foreign	2
Strictness, gnits	2
Strictness, gnu	2
STRIP, how to setup	11
Subdirectory, objects in	23
SUBDIRS, explained	17
SUBDIRS, overriding	17
suffix .la, defined	22
suffix .lo, defined	22
SUFFIXES, adding	47
Support for C++	28
Support for Fortran 77	28
Support for GNU Gettext	36
Support for Java	32

T

tags	47
TAGS support	47
Target, install-info	39
Target, install-man	40
Target, noinstall-info	39
Target, noinstall-man	40
termios POSIX headers	16
Test suites	44
Tests, expected failure	44
Texinfo file handling example	7
Texinfo macro, EDITION	38
Texinfo macro, UPDATED	38
Texinfo macro, UPDATED-MONTH	38
Texinfo macro, VERSION	38
texinfo.tex	39
TEXINFOS primary, defined	38

U

Uniform naming scheme	3
uninstall	42, 51
uninstall-local	51
UPDATED Texinfo macro	38
UPDATED-MONTH Texinfo macro	38
user variables	4

V

variables, reserved for the user 4
VERSION Texinfo macro 38

W

Windows 33

Y

yacc, multiple parsers 26
ylwrap 26

Z

zardoz example 6

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	4
3	Some example packages	5
3.1	A simple example, start to finish	5
3.2	A classic program	6
3.3	Building etags and ctags	7
4	Creating a ‘Makefile.in’	9
5	Scanning ‘configure.in’	10
5.1	Configuration requirements	11
5.2	Other things Automake recognizes	11
5.3	Auto-generating aclocal.m4	13
5.4	Autoconf macros supplied with Automake	14
5.5	Writing your own aclocal macros	17
6	The top-level ‘Makefile.am’	17
7	An Alternative Approach to Subdirectories	18
8	Rebuilding Makefiles	19

9	Building Programs and Libraries	19
9.1	Building a program	19
9.1.1	Introductory blathering	19
9.1.2	Conditional compilations	20
9.1.3	Linking the program	20
9.2	Building a library	21
9.3	Building a Shared Library	22
9.4	Program and Library Variables	22
9.5	Special handling for LIBOBJ and ALLOCA	25
9.6	Variables used when building a program	25
9.7	Yacc and Lex support	26
9.8	C++ Support	28
9.9	Assembly Support	28
9.10	Fortran 77 Support	28
9.10.1	Preprocessing Fortran 77	29
9.10.2	Compiling Fortran 77 Files	29
9.10.3	Mixing Fortran 77 With C and C++	29
9.10.3.1	How the Linker is Chosen	31
9.10.4	Fortran 77 and Autoconf	31
9.11	Java Support	32
9.12	Support for Other Languages	32
9.13	Automatic de-ANSI-fication	32
9.14	Automatic dependency tracking	33
9.15	Support for executable extensions	33
10	Other Derived Objects	34
10.1	Executable Scripts	34
10.2	Header files	35
10.3	Architecture-independent data files	35
10.4	Built sources	35
11	Other GNU Tools	36
11.1	Emacs Lisp	36
11.2	Gettext	36
11.3	Libtool	36
11.4	Java	36
11.5	Python	37
12	Building documentation	38
12.1	Texinfo	38
12.2	Man pages	39

13	What Gets Installed	40
13.1	Basics of installation	40
13.2	The two parts of install	41
13.3	Extending installation	41
13.4	Staged installs	41
13.5	Rules for the user	42
14	What Gets Cleaned	42
15	What Goes in a Distribution	42
15.1	Basics of distribution	42
15.2	Fine-grained distribution control	43
15.3	The dist hook	43
15.4	Checking the distribution	43
15.5	The types of distributions	44
16	Support for test suites	44
16.1	Simple Tests	44
16.2	DejaGNU Tests	44
17	Changing Automake's Behavior	45
18	Miscellaneous Rules	47
18.1	Interfacing to <code>etags</code>	47
18.2	Handling new file extensions	47
18.3	Support for Multilibs	48
19	Include	48
20	Conditionals	48
21	The effect of <code>--gnu</code> and <code>--gnits</code>	49
22	The effect of <code>--cygnus</code>	50
23	When Automake Isn't Enough	51
24	Distributing 'Makefile.in's	51
	Macro and Variable Index	52
	General Index	55