GNU Automake

For version 1.3, 3 April 1998

David MacKenzie and Tom Tromey

This is the first edition of the GNU Automake documentation,
and is consistent with GNU Automake 1.3.

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

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 files 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 "The Autoconf Manual" in *The Autoconf Manual*), and enforces certain restrictions on the 'configure.in' contents.

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 ⟨automake-bugs@gnu.org⟩.

2 General ideas

There are a few basic ideas that will help 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 instances a 'bin_PROGRAMS' macro definition will cause targets for compiling and linking to be generated.

The macro definitions and targets in the 'Makefile.am' are copied 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 (or variable 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, it is generally best to avoid making use of it, as sometimes the generated rules are very particular.

Similarly, a variable defined in 'Makefile.am' will override any definition of the variable 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 variables generated by automake are considered to be for internal use only, and their names might change in future releases.

When examining a variable definition, Automake will recursively examine variables referenced in the definition. E.g., 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 '##' 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 Depth

automake supports three kinds of directory hierarchy: "flat", "shallow", and "deep".

A *flat* package is one in which all the files are in a single directory. The 'Makefile.am' for such a package by definition lacks a SUBDIRS macro. An example of such a package is termutils.

A *deep* package is one in which all the source lies in subdirectories; the top level directory contains mainly configuration information. GNU cpio is a good example of such a package, as is GNU tar. The top level 'Makefile.am' for a deep package will contain a SUBDIRS macro, but no other macros to define objects which are built.

A *shallow* package is one in which the primary source resides in the top-level directory, while various parts (typically libraries) reside in subdirectories. Automake is one such package (as is GNU make, which does not currently use automake).

2.3 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.

For more information on the precise implications of the strictness level, see See Chapter 18 [Gnits], page 31.

2.4 The Uniform Naming Scheme

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

At `make` time, certain variables are used to determine which objects are to be built. These variables are called *primary* variables. For instance, the primary variable `PROGRAMS` holds a list of programs which are to be compiled and linked.

A different set of variables is used to decide where the built objects should be installed. These variables are named after the primary variables, but have a prefix indicating 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 `$(datadir)/@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 to 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 = @PROGRAMS@
```

Defining a primary variable without a prefix (eg `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 (eg 'zar') is valid if a variable of the same name with 'dir' appended is defined (eg 'zardir').

For instance, until HTML support is part of Automake, you could use this to install raw HTML documentation:

```
htmldir = $(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.

Possible primary names are 'PROGRAMS', 'LIBRARIES', 'LISP', 'SCRIPTS', 'DATA', 'HEADERS', 'MANS', and 'TEXINFOS'.

2.5 How derived variables are named

Sometimes a Makefile variable name is derived from some text the user supplies. For instance program names are rewritten into Makefile macro names. Automake canonicalizes this text, so that it does not have to follow Makefile variable naming rules. All characters in the name except for letters, numbers, and the underscore are turned into underscores when making macro references. E.g., if your program is named sniff-glue, the derived variable name would be sniff_glue_SOURCES, not sniff-glue_SOURCES.

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 simplest 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. aclocal 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. zardoz is a user program, so you want to install it where the rest of the user programs go. zardoz also has some Texinfo documentation. Your 'configure.in' script uses AC_REPLACE_FUNCS, so you need to link against '@LIBOBJS@'. So here's what you'd write:

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

info_TEXINFOS = zardoz.texi
```

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

3.2 A classic program

hello is renowned for its classic simplicity and versatility. This section shows how Automake could be used with the Hello package. The examples below are from the latest GNU Hello, but all the maintainer-only code has been 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. GNU Hello is a deep package.

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.O
SUBDIRS = doc intl po src tests
```

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

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

In 'doc/Makefile.am' we see:

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

This is sufficient to build, install, and distribute the 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; supporting that 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
```

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 9) 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.

`--amdir=`*dir*

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

`--build-dir=`*dir*
: Tell Automake where the build directory is. This option is used when including dependencies into a '`Makefile.in`' generated by `make dist`; it should not be used otherwise.

`--cygnus`
: Causes the generated '`Makefile.in`'s to follow Cygnus rules, instead of GNU or Gnits rules. See Chapter 19 [Cygnus], page 31 for more information.

`--foreign`
: Set the global strictness to '`foreign`'. See Section 2.3 [Strictness], page 2 for more information.

`--gnits`
: Set the global strictness to '`gnits`'. See Chapter 18 [Gnits], page 31 for more information.

`--gnu`
: Set the global strictness to '`gnu`'. See Chapter 18 [Gnits], page 31 for more information. This is the default strictness.

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

`-i`
`--include-deps`
: Include all automatically generated dependency information (see Section 7.9 [Dependencies], page 21) in the generated '`Makefile.in`'. This is generally done when making a distribution; see Chapter 13 [Dist], page 26.

`--generate-deps`
: Generate a file concatenating all automatically generated dependency information (see Section 7.9 [Dependencies], page 21) into one file, '`.dep_segment`'. This is generally done when making a distribution; see Chapter 13 [Dist], page 26. It is useful when maintaining a '`SMakefile`' or makefiles for other platforms ('`Makefile.DOS`', etc.) It can only be used in conjunction with `--include-deps`, `--srcdir-name`, and `--build-dir`. Note that if this option is given, no other processing is done.

`--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.

`--srcdir-name=`*dir*
: Tell Automake the name of the source directory associated with the current build. This option is used when including dependencies into a '`Makefile.in`' generated by `make dist`; it should not be used otherwise.

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

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

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 simplest way to meet the basic Automake requirements is to use the macro AM_INIT_AUTOMAKE (see Section 5.4 [Macros], page 12). But if you prefer, you can do the required steps by hand:

- Define the variables PACKAGE and VERSION with AC_SUBST. PACKAGE should be the name of the package as it appears when bundled for distribution. For instance, Automake defines PACKAGE to be 'automake'. VERSION should be the version number of the release that is being developed. We recommend that you make 'configure.in' the only place in your package where the version number is defined; this makes releases simpler.

  Automake doesn't do any interpretation of PACKAGE or VERSION, except in 'Gnits' mode (see Chapter 18 [Gnits], page 31).
- Use the macro AC_ARG_PROGRAM if a program or script is installed.
- Use AC_PROG_MAKE_SET if the package is not flat.
- Use AM_SANITY_CHECK to make sure the build environment is sane.
- Use AM_PROG_INSTALL if any scripts (see Section 8.1 [Scripts], page 22) are installed by the package. Otherwise, use AC_PROG_INSTALL.
- Use AM_MISSING_PROG to see whether the programs aclocal, autoconf, automake, autoheader, and makeinfo are in the build environment. Here is how this is done:

  ```
  missing_dir=`cd $ac_aux_dir && pwd`
  AM_MISSING_PROG(ACLOCAL, aclocal, $missing_dir)
  AM_MISSING_PROG(AUTOCONF, autoconf, $missing_dir)
  AM_MISSING_PROG(AUTOMAKE, automake, $missing_dir)
  AM_MISSING_PROG(AUTOHEADER, autoheader, $missing_dir)
  AM_MISSING_PROG(MAKEINFO, makeinfo, $missing_dir)
  ```

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. 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.

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 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). 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.

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.

AC_CANONICAL_SYSTEM
: This is similar to AC_CANONICAL_HOST, but also defines the 'Makefile' variables 'build_alias' and 'target_alias'.

AC_FUNC_ALLOCA
AC_FUNC_GETLOADAVG
AC_FUNC_MEMCMP
AC_STRUCT_ST_BLOCKS
AC_FUNC_FNMATCH
AM_FUNC_STRTOD
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 7.2 [A Library], page 17 for more information.

`LIBOBJS` Automake will detect statements which put '`.o`' files into `LIBOBJS`, and will treat these additional files as if they were discovered via `AC_REPLACE_FUNCS`.

`AC_PROG_RANLIB`
: This is required if any libraries are built in the package.

`AC_PROG_CXX`
: This is required if any C++ source is included.

`AM_PROG_LIBTOOL`
: Automake will turn on processing for `libtool` (see section “The Libtool Manual” 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.

`AC_DECL_YYTEXT`
: This macro is required if there is Lex source in the package.

`AC_PROG_LEX`
: If a Lex source file is seen, then this macro must be used.

`ALL_LINGUAS`
: If Automake sees that this variable is set in '`configure.in`', it will check the '`po`' directory to ensure that all the named '`.po`' files exist, and that all the '`.po`' files that exist are named.

`AM_C_PROTOTYPES`
: This is required when using automatic de-ANSI-fication, see Section 7.8 [ANSI], page 20.

`AM_GNU_GETTEXT`
: This macro is required for packages which use GNU gettext (see Section 9.2 [gettext], page 23). 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 18 [Gnits], page 31).

`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`'.

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` 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_CYGWIN32`
: Check to see if this `configure` is being run in the 'Cygwin32' environment. (FIXME xref). If so, define output variable `EXEEXT` to '.exe'; otherwise define it to the empty string. Automake recognizes this macro and uses it to generate 'Makefile.in's which will automatically work under 'Cygwin32'. In the

‘Cygwin32’ environment, gcc generates executables whose names end in ‘.exe’, even if this was not specified on the command line. Automake adds special code to ‘Makefile.in’ to gracefully deal with this.

AM_FUNC_STRTOD
: If the strtod function is not available, or does not work correctly (like the one on SunOS 5.4), add ‘strtod.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_MKTIME
: Check for a working mktime function. If not found, add ‘mktime.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_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_CC_STDC
: If the C compiler in 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.

AM_PROG_INSTALL
: Like AC_PROG_INSTALL, but also defines INSTALL_SCRIPT.

`AM_PROG_LEX`
: Like `AC_PROG_LEX` with `AC_DECL_YYTEXT`, but uses the `missing` script on systems that do not have lex. ‘`HP-UX 10`’ is one such system.

`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` package. If the user configures with ‘`--with-dmalloc`’, then define `WITH_DMALLOC` and add ‘`-ldmalloc`’ to `LIBS`. The `dmalloc` package can be found at ‘`ftp://ftp.letters.com/src/dmalloc/dmalloc.tar.gz`’

`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

Aclocal 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. Aclocal also understands `AC_REQUIRE`, so it is safe to put each macro in a separate file.

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

6 The top-level ‘`Makefile.am`’

In non-flat packages, 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 (eg `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 a deep package, the top-level 'Makefile.am' is often very short. For instance, here is the 'Makefile.am' from the Hello distribution:

```
EXTRA_DIST = BUGS ChangeLog.O 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 = @SUBDIRS@
```

Then in your 'configure.in' you can specify:

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

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.

SUBDIRS can contain configure substitutions (eg '@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.

The use of SUBDIRS is not restricted to just the top-level 'Makefile.am'. Automake can be used to construct packages of arbitrary depth.

7 Building Programs and Libraries

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

7.1 Building a program

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').

For instance:

```
bin_PROGRAMS = hello
```

In this simple case, the resulting 'Makefile.in' will contain code to generate a program named hello. 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 'prog_SOURCES' is needed, but not specified, then it defaults to the single file 'prog.c'. In the example above, the definition of hello_SOURCES is actually redundant.

Multiple programs can be built in a single directory. Multiple programs can share a single source file. The source file 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 an '_SOURCES' variable; this file should not be distributed. Lex ('.l') and yacc ('.y') files can also be listed; see Section 7.6 [Yacc and Lex], page 18.

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
```

Similarly, 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.

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 '-l' and '-L'). 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, '-l', and '-L' options removed. The configure substitutions that are left in are only '@LIBOBJS@' and '@ALLOCA@'; these are left because it is known that they will not cause an invalid value for 'prog_DEPENDENCIES' to be generated.

7.2 Building a library

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

See Section 7.4 [A Shared Library], page 17, 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 2.5 [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 = @LIBOBJS@ @ALLOCA@
```

7.3 Special handling for LIBOBJS and ALLOCA

Automake explicitly recognizes the use of `@LIBOBJS@` and `@ALLOCA@`, and uses this information, plus the list of `LIBOBJS` files derived from 'configure.in' to automatically include the appropriate source files in the distribution (see Chapter 13 [Dist], page 26). These source files are also automatically handled in the dependency-tracking scheme, see See Section 7.9 [Dependencies], page 21.

`@LIBOBJS@` and `@ALLOCA@` are specially recognized in any '_LDADD' or '_LIBADD' variable.

7.4 Building a Shared Library

Building shared libraries is a relatively complex matter. For this reason, GNU Libtool (see section "The Libtool Manual" 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 'noinst_LTLIBRARIES' and 'check_LTLIBRARIES' are not allowed.

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 @LIBOBJS@, a libtool library must use @LTLIBOBJS@. 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.

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

7.5 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:

INCLUDES
: A list of '-I' options. This can be set in your 'Makefile.am' if you have special directories you want to look in. automake already provides some '-I' options automatically. In particular it generates '-I$(srcdir)' and a '-I' pointing to the directory holding 'config.h' (if you've used AC_CONFIG_HEADER or AM_CONFIG_HEADER).

  INCLUDES can actually be used for other cpp options besides '-I'. For instance, it is sometimes used to pass arbitrary '-D' options to the compiler.

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.

7.6 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'.

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. Automake's lex support also requires that you use the 'AC_DECL_YYTEXT' macro—automake needs to know the value of 'LEX_OUTPUT_ROOT'.

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 subdirectory. This is necessary because yacc's output filename is fixed, and a parallel make could conceivably invoke more than one instance of yacc simultaneously. ylwrap is distributed with automake. It should appear in the directory specified by 'AC_CONFIG_AUX_DIR', or the current directory if that macro is not used in 'configure.in'.

For yacc, simply managing locking is insufficient. yacc output also 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
```

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.

7.7 C++ and other languages

Automake includes full support for C++, and rudimentary support for other languages. Support for other languages will be improved based on demand.

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.

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.

CXXCOMPILE
: The command used to actually compile a C++ source file. The file name is appended to form the complete command line.

CXXLINK
: The command used to actually link a C++ program.

7.8 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 SunOS).

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 (Chapter 15 [Options], page 28) 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.

De-ANSI-fication support 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.

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.

7.9 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, and distribute the dependencies in the generated 'Makefile.in'.

Currently this support requires the use of GNU make and gcc. It might become possible in the future to supply a different dependency generating program, if there is enough demand. In the meantime, this mode is enabled by default if any C program or library is defined in the current directory, so you may get a 'Must be a separator' error from non-GNU make.

When you decide to make a distribution, the dist target will re-run automake with '--include-deps' and other options. This will cause the previously generated dependencies to be inserted into the generated 'Makefile.in', and thus into the distribution. This step also turns off inclusion of the dependency generation code, so that those who download your distribution but don't use GNU make and gcc will not get errors.

When added to the 'Makefile.in', the dependencies have all system-specific dependencies automatically removed. This can be done by listing the files in 'OMIT_DEPENDENCIES'. For instance all references to system header files are removed by automake. Sometimes it is useful to specify that a certain header file should be removed. For instance if your 'configure.in' uses 'AM_WITH_REGEX', then any dependency on 'rx.h' or 'regex.h' should be removed, because the correct one cannot be known until the user configures the package.

As it turns out, automake is actually smart enough to handle the particular case of the regular expression header. It will also automatically omit 'libintl.h' if 'AM_GNU_GETTEXT' is used.

Automatic dependency tracking can be suppressed by putting no-dependencies in the variable AUTOMAKE_OPTIONS.

If you unpack a distribution made by make dist, and you want to turn on the dependency-tracking code again, simply re-run automake.

The actual dependency files are put under the build directory, in a subdirectory named '.deps'. These dependencies are machine specific. It is safe to delete them if you like; they will be automatically recreated during the next build.

8 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.

8.1 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 12 [Clean], page 26 for more information.

`automake` itself is a 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`.

8.2 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.

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 7.1 [A Program], page 15. Headers listed in a '_SOURCES' variable need not be listed in any '_HEADERS' variable.

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

8.3 Architecture-independent data files

Automake supports the installation of miscellaneous data files using the 'DATA' family of variables.

Such data can be installed in the directories `datadir`, `sysconfdir`, `sharedstatedir`, `localstatedir`, or `pkgdatadir`.

By default, data files are not included in a distribution.

Here is how `automake` installs its auxiliary data files:

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

8.4 Built sources

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

Built sources are also not compiled by default. You must explicitly mention them in some other '`_SOURCES`' variable for this to happen.

Note that, in some cases, `BUILT_SOURCES` will work in somewhat suprising ways. In order to get the built sources to work with automatic dependency tracking, the '`Makefile`' must depend on `$(BUILT_SOURCES)`. This can cause these sources to be rebuilt at what might seem like funny times.

9 Other GNU Tools

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

9.1 Emacs Lisp

Automake provides some support for Emacs Lisp. The '`LISP`' primary is used to hold a list of '`.el`' files. Possible prefixes for this primary are '`lisp_`' and '`noinst_`'. Note that if `lisp_LISP` is defined, then '`configure.in`' must run `AM_PATH_LISPDIR` (see Section 5.4 [Macros], page 12).

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 recommand 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.

9.2 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 ensure that these directories exist and are mentioned in `SUBDIRS`.

Furthermore, Automake checks that the definition of '`ALL_LINGUAS`' in '`configure.in`' corresponds to all the valid '`.po`' files, and nothing more.

9.3 Guile

Automake provides some automatic support for writing Guile modules. Automake will turn on Guile support if the `AM_INIT_GUILE_MODULE` macro is used in '`configure.in`'.

Right now Guile support just means that the `AM_INIT_GUILE_MODULE` macro is understood to mean:

- `AM_INIT_AUTOMAKE` is run.
- `AC_CONFIG_AUX_DIR` is run, with a path of '`..`'.

As the Guile module code matures, no doubt the Automake support will grow as well.

9.4 Libtool

Automake provides support for GNU Libtool (see section "The Libtool Manual" in *The Libtool Manual*) with the '`LTLIBRARIES`' primary. See Section 7.4 [A Shared Library], page 17.

9.5 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.

10 Building documentation

Currently Automake provides support for Texinfo and man pages.

10.1 Texinfo

If the current directory contains Texinfo source, you must declare it with the '`TEXINFOS`' primary. Generally Texinfo files are converted into info, and thus the `info_TEXINFOS` macro is most commonly used here. Note that any Texinfo source file must end in the '`.texi`' or '`.texinfo`' extension.

If the '`.texi`' file `@include`s '`version.texi`', then that file will be automatically generated. '`version.texi`' defines three Texinfo macros you can reference: `EDITION`, `VERSION`, and `UPDATED`. The first two hold the version number of your package (but are kept separate for clarity); the last is the date the primary file was last modified. The '`version.texi`' support requires the `mdate-sh` program; this program is supplied with Automake.

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 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', 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.

10.2 Man pages

A package can also include man pages. (Though 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.

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 documentation is handled in GNU cpio (which includes both Texinfo documentation and man pages):

```
info_TEXINFOS = cpio.texi
man_MANS = cpio.1 mt.1
```

Texinfo source and info pages are all considered to be source for the purposes of making a distribution.

Man pages are not currently considered to be source, because it is not uncommon for man pages to be automatically generated.

11 What Gets Installed

Naturally, Automake handles the details of actually installing your program once it has been built. All PROGRAMS, SCRIPTS, LIBRARIES, LISP, DATA and HEADERS are automatically installed in the appropriate places.

Automake also handles installing any specified info and man pages.

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. The `install` target depends on both of these targets.

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

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.

Variables using the standard directory prefixes '`data`', '`info`', '`man`', '`include`', '`oldinclude`', '`pkgdata`', or '`pkginclude`' (eg '`data_DATA`') are installed by '`install-data`'.

Variables using the standard directory prefixes '`bin`', '`sbin`', '`libexec`', '`sysconf`', '`localstate`', '`lib`', or '`pkglib`' (eg '`bin_PROGRAMS`') are installed by '`install-exec`'.

Any variable using a user-defined directory prefix with '`exec`' in the name (eg '`myexecbin_PROGRAMS`' is installed by '`install-exec`'. All other user-defined prefixes are installed by '`install-data`'.

Automake generates support for the '`DESTDIR`' variable in all install rules; see See section "Makefile Conventions" in *The GNU Coding Standards*.

12 What Gets Cleaned

The GNU Makefile Standards specify a number of different clean rules. Generally the files that can 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`.

13 What Goes in a 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`'.

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` (ie, 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. Note that `EXTRA_DIST` can only handle files in the current directory; files in other directories will cause `make dist` runtime failures.

If you define `SUBDIRS`, automake will recursively include the subdirectories in the distribution. If `SUBDIRS` is defined conditionally (see Chapter 17 [Conditionals], page 30), 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 random/a1 random/a2 $(distdir)/random
```

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

14 Support for test suites

Automake supports a two forms of test suite.

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 ⟨undefined⟩ [Build Directories], page ⟨undefined⟩), and in particular for the `distcheck` target (see Chapter 13 [Dist], page 26).

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 (eg '`$(SHELL) -x`'); this can be useful for debugging the tests.

If '`dejagnu`' appears in `AUTOMAKE_OPTIONS`, then the a `dejagnu`-based test suite is assumed. The value of the variable `DEJATOOL` is passed as the `--tool` argument to `runtest`; 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`, `RUNTEST` and `RUNTESTFLAGS` 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.

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

15 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 7.8 [ANSI], page 20. If preceeded by a path, the generated '`Makefile.in`' will look in the specified directory to find the '`ansi2knr`' program. Generally the path should be a relative path to another directory in the same distribution (though Automake currently does not check this).

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

`dejagnu`
: Cause `dejagnu`-specific rules to be generated. See Chapter 14 [Tests], page 27.

`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; a traditional `tar` and `compress` will be assumed. Warning: if you are actually using `GNU tar`, then the generated archive might contain nonportable constructs.

`no-dependencies`
: This is similar to using '`--include-deps`' on the command line, but is useful for those situations where you don't have the necessary bits to make automatic dependency tracking work See Section 7.9 [Dependencies], page 21. In this case the effect is to effectively disable automatic dependency tracking.

`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.

`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.

version
A version number (eg '`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`.

16 Miscellaneous Rules

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

16.1 Interfacing to `etags`

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

If any C 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.

Also, if the variable `ETAGS_ARGS` is defined, a `tags` target will be generated. 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.

16.2 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, currently automake does not provide any Java support. If you wrote a macro to generate '`.class`' files from '`.java`' source files, you would also need to add these suffixes to the list:

```
SUFFIXES = .java .class
```

17 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. The `AM_CONDITIONAL` macro takes two arguments.

The first argument to `AM_CONDITIONAL` is the name of the conditional. This should be a simple string starting with a letter and containing only letters, digits, and underscores.

The second argument to `AM_CONDITIONAL` is a shell condition, suitable for use in a shell if statement. The condition is evaluated when `configure` is run.

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 7.1 [A Program], page 15).

You may only test a single variable in an `if` statement. The `else` statement may be omitted. Conditionals may be nested to any depth.

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.

18 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.
- The file ‘COPYING.LIB’ is prohibited. The LGPL is apparently considered a failed experiment.
- ‘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.

19 The effect of --cygnus

Cygnus Solutions 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’ will look for files in the build directory as well as the source directory. This is required to support putting info files into the build directory.

- 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.

20 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 non-flat package. 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
```

21 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; examine each file to see.

22 Some ideas for the future

Here are some things that might happen in the future:

- HTML support.
- The output will be cleaned up. For instance, only variables which are actually used will appear in the generated 'Makefile.in'.
- There will be support for automatically recoding a distribution. The intent is to allow a maintainer to use whatever character set is most convenient locally, but for all distributions to be Unicode or ISO 10646 with the UTF-8 encoding.
- Support for automatically generating packages (eg Debian packages, RPM packages, Solaris packages, etc). This will happen more quickly if someone with package-building experience can tell me what would be helpful.
- Rewrite in Guile. This won't happen in the near future, but it will eventually happen.

Index

U

V

Y

Table of Contents