automake

Node: Top, Next: Introduction, Previous: (dir), Up: (dir)

GNU Automake

This file documents the GNU Automake package. Automake is a program which creates GNU standards-compliant Makefiles from template files. This edition documents version 1.7.8.

Node: Introduction, Next: Generalities, Previous: Top, Up: Top

Introduction

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

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

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

Automake does constrain a project in certain ways; for instance it assumes that the project uses Autoconf (see Introduction), and enforces certain restrictions on the `configure.in` contents[2].

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

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

Node: Generalities, Next: Examples, Previous: Introduction, Up: Top

General ideas

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

Node: General Operation, Next: Strictness, Previous: Generalities, Up: Generalities

General Operation

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

The variable 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 most GNU make extensions are not recognized by Automake. Using such extensions in a `Makefile.am` will lead to errors or confusing behavior.

A special exception is that the GNU make append operator, `+=`, is supported. This operator appends its right hand argument to the variable specified on the left. Automake will translate the operator into an ordinary `=` operator; `+=` will thus work with any make program.

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

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` or `AC_SUBST`'ed from `configure.in` 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. 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
```

Node: Strictness, Next: Uniform, Previous: General Operation, Up: Generalities

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 Gnits.

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 Cygnus.

Node: Uniform, Next: Canonicalization, Previous: Strictness, Up: Generalities

The Uniform Naming Scheme

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

At `make` time, certain variables are used to determine which objects are to be built. The variable names are made of several pieces 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 Directory Variables). 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:

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

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

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

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

Some primaries also allow additional prefixes which control other aspects of `automake`'s behavior. The currently defined prefixes are `dist_`, `nodist_`, and `nobase_`. These prefixes are explained later (see Program and Library Variables).

Node: Canonicalization, Next: User Variables, Previous: Uniform, Up: Generalities

How derived variables are named

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

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

`libmumble++.a` should be listed in the `libmumble___a_SOURCES` variable.

The strudel is an addition, to make the use of Autoconf substitutions in variable names less obfuscating.

Node: User Variables, Next: Auxiliary Programs, Previous: Canonicalization, Up: Generalities

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`.

Node: Auxiliary Programs, Previous: User Variables, Up: Generalities

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 ANSI).

`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. These programs are updated regularly to support new architectures and fix probes broken by changes in new kernel versions. You are encouraged to fetch the latest versions of these files from <`ftp://ftp.gnu.org/gnu/config/`> before making a release.

`depcomp`
: This program understands how to run a compiler so that it will generate not only the desired output but also dependency information 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`, `make ps` and `make pdf` 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.

Node: Examples, Next: Invoking Automake, Previous: Generalities, Up: Top

Some example packages

Node: Complete, Next: Hello, Previous: Examples, Up: Examples

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`:

```
AC_INIT(zardoz, 1.0)
AM_INIT_AUTOMAKE
...
```

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

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

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

Now it is time to write your `Makefile.am` for `zardoz`. Since `zardoz` is a user program, you want to install it where the rest of the user programs go: `bindir`. Additionally, `zardoz` has some Texinfo documentation. Your `configure.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!

Node: Hello, Next: true, Previous: Complete, Up: Examples

A classic program

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

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

Here is the `configure.in` from GNU Hello. **Please note:** The calls to `AC_INIT` and `AM_INIT_AUTOMAKE` in this example use a deprecated syntax. For the current approach, see the description of `AM_INIT_AUTOMAKE` in Public macros.

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

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

     dnl Checks for programs.
     AC_PROG_CC
     AC_ISC_POSIX

     dnl Checks for libraries.

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

     dnl Checks for library functions.
     AC_FUNC_ALLOCA

     dnl Check for st_blksize in struct stat
     AC_ST_BLKSIZE
```

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

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

The top-level `Makefile.am`:

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

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

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

In `doc/Makefile.am` we see:

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

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

Here is `tests/Makefile.am`:

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

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

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

```
bin_PROGRAMS = hello
```

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

Node: true, Previous: Hello, Up: Examples

Building true and false

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

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

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

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

Note that there is no `true_SOURCES` definition. Automake will implicitly assume that there is a source file named `true.c`, and define rules to compile `true.o` and link `true`. The `true.o: true.c` rule supplied by the above `Makefile.am`, will override the Automake generated rule to build `true.o`.

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

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

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

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

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

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

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

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

```
bin_PROGRAMS = false true

false_SOURCES = true.c
false_CPPFLAGS = -DEXIT_CODE=1

true_SOURCES = true.c
true_CPPFLAGS = -DEXIT_CODE=0
```

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

Node: Invoking Automake, Next: configure, Previous: Examples, Up: Top

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 configure) 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`. (Alternatively, you may rely on

Autoconf's `autoreconf`, which is able to recurse your package tree and run `automake` where appropriate.)

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

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

`automake` accepts the following options:

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

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

`--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 Cygnus.

`-f`

`--force-missing`
: When used with `--add-missing`, causes standard files to be reinstalled even if they already exist in the source tree. This involves removing the file from the source tree before creating the new symlink (or, with `--copy`, copying the new file).

`--foreign`
: Set the global strictness to `foreign`. For more information, see Strictness.

`--gnits`
: Set the global strictness to `gnits`. For more information, see Gnits.

`--gnu`
: Set the global strictness to `gnu`. For more information, see Gnits. This is the default strictness.

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

`-i`
`--ignore-deps`
: This disables the dependency tracking feature in generated `Makefile`s; see Dependencies.

`--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.

  Due to a bug in its implementation, this option is currently ignored. It will be fixed in Automake 1.8.

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

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

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

`-W CATEGORY`

`--warnings=`*category*
: Output warnings falling in *category*. *category* can be one of:

  `gnu`
  : warnings related to the GNU Coding Standards (see Top).

`obsolete`
: obsolete features or constructions

`portability`
: portability issues (e.g., use of Make features which are known not portable)

`syntax`
: weird syntax, unused variables, typos

`unsupported`
: unsupported or incomplete features

`all`
: all the warnings

`none`
: turn off all the warnings

`error`
: treat warnings as errors

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

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

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

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

Node: configure, Next: Top level, Previous: Invoking Automake, Up: Top

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.

Node: Requirements, Next: Optional, Previous: configure, Up: configure

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 (see Macros).

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

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

  These files are all removed by `make distclean`.

Node: Optional, Next: Invoking aclocal, Previous: Requirements, Up: configure

Other things Automake recognizes

Every time Automake is run it calls Autoconf to trace `configure.in`. This way it can recognize the use of certain macros and tailor the generated `Makefile.in` appropriately. Currently recognized macros and their effects are:

`AC_CONFIG_HEADERS`
: Automake will generate rules to rebuild these headers. Older versions of Automake required the use of `AM_CONFIG_HEADER` (see Macros); this is no longer the case today.

`AC_CONFIG_AUX_DIR`

Automake will look for various helper scripts, such as `mkinstalldirs`, in the directory named in this macro invocation. (The full list of scripts is: `config.guess`, `config.sub`, `depcomp`, `elisp-comp`, `compile`, `install-sh`, `ltmain.sh`, `mdate-sh`, `missing`, `mkinstalldirs`, `py-compile`, `texinfo.tex`, and `ylwrap`.) Not all scripts are always searched for; some scripts will only be sought if the generated `Makefile.in` requires them.

If `AC_CONFIG_AUX_DIR` is not given, the scripts are looked for in their `standard` locations. For `mdate-sh`, `texinfo.tex`, and `ylwrap`, the standard location is the source directory corresponding to the current `Makefile.am`. For the rest, the standard location is the first one of `.`, `..`, or `../..` (relative to the top source directory) that provides any one of the helper scripts. See Finding `configure' Input.

Required files from `AC_CONFIG_AUX_DIR` are automatically distributed, even if there is no `Makefile.am` in this directory.

`AC_CANONICAL_HOST`

Automake will ensure that `config.guess` and `config.sub` exist. Also, the `Makefile` variables `host_alias` and `host_triplet` are introduced. See Getting the Canonical System Type.

`AC_CANONICAL_SYSTEM`

This is similar to `AC_CANONICAL_HOST`, but also defines the `Makefile` variables `build_alias` and `target_alias`. See Getting the Canonical System Type.

`AC_LIBSOURCE`
`AC_LIBSOURCES`
`AC_LIBOBJ`

Automake will automatically distribute any file listed in `AC_LIBSOURCE` or `AC_LIBSOURCES`.

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

By the way, direct assignments to `LIBOBJS` are no longer supported. You should always use `AC_LIBOBJ` for this purpose. See `AC_LIBOBJ` vs. `LIBOBJS`.

`AC_PROG_RANLIB`

This is required if any libraries are built in the package. See Particular Program Checks.

`AC_PROG_CXX`

This is required if any C++ source is included. See Particular Program Checks.

`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 Particular Program Checks.

`AC_F77_LIBRARY_LDFLAGS`
: This is required for programs and shared libraries that are a mixture of languages that include Fortran 77 (see Mixing Fortran 77 With C and C++). See Autoconf macros supplied with Automake.

`AC_PROG_LIBTOOL`
: Automake will turn on processing for `libtool` (see Introduction).

`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 Particular Program Checks).

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

`AC_SUBST`
: The first argument is automatically defined as a variable in each generated `Makefile.in`. See Setting Output Variables.

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

`AM_C_PROTOTYPES`
: This is required when using automatic de-ANSI-fication; see ANSI.

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

`AM_MAINTAINER_MODE`
: This macro adds a `--enable-maintainer-mode` option to `configure`. If this is used, `automake` will cause `maintainer-only` rules to be turned off by default in the generated `Makefile.in`s. This macro defines the `MAINTAINER_MODE` conditional, which you can use in your own `Makefile.am`.

Node: Invoking aclocal, Next: aclocal options, Previous: Optional, Up: configure

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 allows other packages to supply their own macros.

At startup, `aclocal` scans all the `.m4` files it can find, looking for macro definitions (see Macro search path). 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_DEFUN`s in the files it scans. It also tries to copy the full text of the scanned file into `aclocal.m4`, including both # and `dnl` comments. If you want to make a comment which will be completely ignored by `aclocal`, use ## as the comment leader.

Node: aclocal options, Next: Macro search path, Previous: Invoking aclocal, Up: configure

aclocal options

`aclocal` accepts the following options:

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

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

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

`--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 third-party `.m4` files. When this option is given, normal processing is suppressed. This option can be used by a package to determine where to install a macro file.

`--verbose`

Print the names of the files it examines.

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

Node: Macro search path, Next: Macros, Previous: aclocal options, Up: configure

Macro search path

By default, `aclocal` searches for `.m4` files in the following directories, in this order:

acdir-APIVERSION
This is where the `.m4` macros distributed with automake itself are stored. *APIVERSION* depends on the automake release used; for automake 1.6.x, *APIVERSION* = `1.6`.

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

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

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

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

Modifying the macro search path: `--acdir`

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

1. `/opt/private/`

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

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

Any extra directories specified using `-I` options (see aclocal options) are *prepended* to this search list. Thus, `aclocal -I /foo -I /bar` results in the following search path:

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

Modifying the macro search path: `dirlist`

There is a third mechanism for customizing the search path. If a `dirlist` file exists in *acdir*, then that file is assumed to contain a list of directories, one per line, to be added to the search list. These directories are searched *after* all other directories.

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

```
/test1
/test2
```

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

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

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

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

1. `/usr/share/aclocal-1.6/`
2. `/usr/share/aclocal/`

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

```
/usr/share/aclocal/dirlist
```

which contains only the single line

```
/usr/local/share/aclocal
```

Now, the "default" search path on the affected system is

1. `/usr/share/aclocal-1.6/`
2. `/usr/share/aclocal/`
3. `/usr/local/share/aclocal/`

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

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

Node: Macros, Next: Extending aclocal, Previous: Macro search path, Up: configure

Autoconf macros supplied with Automake

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

Node: Public macros, Next: Private macros, Previous: Macros, Up: Macros

Public macros

`AM_CONFIG_HEADER`

Automake will generate rules to automatically regenerate the config header. This obsolete macro is a synonym of `AC_CONFIG_HEADERS` today (see Optional).

`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 Multilibs.

`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([OPTIONS])`
`AM_INIT_AUTOMAKE(PACKAGE, VERSION, [NO-DEFINE])`

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

This macro has two forms, the first of which is preferred. In this form, `AM_INIT_AUTOMAKE` is called with a single argument -- a space-separated list of Automake options which should be applied to every `Makefile.am` in the tree. The effect is as if each option were listed in `AUTOMAKE_OPTIONS`.

The second, deprecated, form of `AM_INIT_AUTOMAKE` has two required arguments: the package and the version number. This form is obsolete because the *package* and *version* can be obtained from Autoconf's `AC_INIT` macro (which itself has an old and a new form).

If your `configure.in` has:

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

you can modernize it as follows:

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

Note that if you're upgrading your `configure.in` from an earlier version of Automake, it is

not always correct to simply move the package and version arguments from AM_INIT_AUTOMAKE directly to AC_INIT, as in the example above. The first argument to AC_INIT should be the name of your package (e.g. GNU Automake), not the tarball name (e.g. automake) that you used to pass to AM_INIT_AUTOMAKE. Autoconf tries to derive a tarball name from the package name, which should work for most but not all package names. (If it doesn't work for yours, you can use the four-argument form of AC_INIT -- supported in Autoconf versions greater than 2.52g -- to provide the tarball name explicitly).

By default this macro AC_DEFINE's PACKAGE and VERSION. This can be avoided by passing the no-define option, as in:

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

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

AM_PATH_LISPDIR

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

Note that this test assumes the emacs found to be a version that supports Emacs Lisp (such as GNU Emacs or XEmacs). Other emacsen can cause this test to hang (some, like old versions of MicroEmacs, start up in interactive mode, requiring C-x C-c to exit, which is hardly obvious for a non-emacs user). In most cases, however, you should be able to use C-c to kill the test. In order to avoid problems, you can set EMACS to "no" in the environment, or use the --with-lispdir option to configure to explicitly set the correct path (if you're sure you have an emacs that supports Emacs Lisp.

AM_PROG_AS

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

AM_PROG_CC_C_O

This is like AC_PROG_CC_C_O, but it generates its results in the manner required by automake. You must use this instead of AC_PROG_CC_C_O when you need this functionality.

AM_PROG_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 ANSI).

This macro is a relic from the time Autoconf didn't offer such a feature. `AM_PROG_CC_STDC`'s logic has now been merged into Autoconf's `AC_PROG_CC` macro, therefore you should use the latter instead. Chances are you are already using `AC_PROG_CC`, so you can simply remove the `AM_PROG_CC_STDC` call and turn all occurrences of `$am_cv_prog_cc_stdc` into `$ac_cv_prog_cc_stdc`. `AM_PROG_CC_STDC` will be marked as obsolete (in the Autoconf sense) in Automake 1.8.

`AM_PROG_LEX`
: Like `AC_PROG_LEX` (see Particular Program Checks), but uses the `missing` script on systems that do not have `lex`. `HP-UX 10` is one such system.

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

`AM_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_WITH_DMALLOC`
: Add support for the dmalloc 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`.

Node: Private macros, Previous: Public macros, Up: Macros

Private macros

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

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

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

`AM_PROG_INSTALL_STRIP`
: This is used to find a version of `install` 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`.

Node: Extending aclocal, Previous: Macros, Up: configure

Writing your own aclocal macros

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

This can be used by libraries 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 macro file's name should end in `.m4`. Such files should be installed in `$(datadir)/aclocal`. This is as simple as writing:

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

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

Starting with Automake 1.8, `aclocal` will warn about all underquoted calls to `AC_DEFUN`. We realize this will annoy a lot of people, because `aclocal` was not so strict in the past and many third party macros are underquoted; and we have to apologize for this temporary inconvenience. The reason we have to be stricter is that a future implementation of `aclocal` will have to temporary include all these third party `.m4` files, maybe several times, even those which are not actually needed. Doing so should alleviate many problem of the current implementation, however it requires a stricter style from the

macro authors. Hopefully it is easy to revise the existing macros. For instance

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

should be rewritten as

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

Wrapping the `AC_PREREQ` call inside the macro ensures that Autoconf 2.57 will not be required if `AX_FOOBAR` is not actually used. Most importantly, quoting the first argument of `AC_DEFUN` allows the macro to be redefined or included twice (otherwise this first argument would be expansed during the second definition).

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

Node: Top level, Next: Alternative, Previous: configure, Up: Top

The top-level `Makefile.am`

Recursing subdirectories

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

The SUBDIRS variable holds a list of subdirectories in which building of various sorts can occur. 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.ams; only Makefiles (after configuration). This allows inclusion of libraries from packages which do not use Automake (such as gettext).

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

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

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

The directories mentioned in SUBDIRS must be direct children of the current directory. For instance, you cannot put src/subdir into SUBDIRS. Instead you should put SUBDIRS = subdir into src/Makefile.am. Automake can be used to construct packages of arbitrary depth this way.

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.

Conditional subdirectories

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

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

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

However make dist should always recurse into both src/ and opt/. Because opt/ should be distributed even if it is not needed in the current configuration. This means opt/Makefile should be created unconditionally. [3]

There are two ways to setup a project like this. You can use Automake conditionals (see Conditionals) or use Autoconf AC_SUBST variables (see Setting Output Variables). Using Automake conditionals is the preferred solution.

Conditional subdirectories with AM_CONDITIONAL

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

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

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

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

As you can see, running make will rightly recurse into src/ and maybe opt/.

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

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

Conditional subdirectories with AC_SUBST

Another idea is to define MAYBE_OPT from ./configure using AC_SUBST:

```
...
if test "$want_opt" = yes; then
  MAYBE_OPT=opt
else
  MAYBE_OPT=
```

```
fi
AC_SUBST([MAYBE_OPT])
AC_CONFIG_FILES([Makefile src/Makefile opt/Makefile])
...
```

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

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

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

How `DIST_SUBDIRS` is used

As shown in the above examples, `DIST_SUBDIRS` is used by targets that need to recurse in all directories, even those which have been conditionally left out of the build.

Precisely, `DIST_SUBDIRS` is used by `make dist`, `make distclean`, and `make maintainer-clean`. All other recursive targets use `SUBDIRS`.

Automake will define `DIST_SUBDIRS` automatically from the possibles values of `SUBDIRS` in all conditions.

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

Node: Alternative, Next: Rebuilding, Previous: Top level, Up: Top

An Alternative Approach to Subdirectories

If you've ever read Peter Miller's excellent paper, Recursive Make Considered Harmful, 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 support [4] to enable you to write a single `Makefile.am` for a complex multi-directory package.

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

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

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

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

`nobase_` should be specified first when used in conjunction with either `dist_` or `nodist_` (see Dist). For instance:

```
nobase_dist_pkgdata_DATA = images/vortex.pgm
```

Node: Rebuilding, Next: Programs, Previous: Alternative, Up: Top

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 variable is only useful in the top-level `Makefile.am`.

Node: Programs, Next: Other objects, Previous: Rebuilding, Up: Top

Building Programs and Libraries

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

Node: A Program, Next: A Library, Previous: Programs, Up: Programs

Building a program

In order to build a program, you need to tell Automake which sources are part of it, and which libraries it should be linked with.

This section also covers conditional compilation of sources or programs. Most of the comments about these also apply to libraries (see A Library) and libtool libraries (see A Shared Library).

Node: Program Sources, Next: Linking, Previous: A Program, Up: A Program

Defining program sources

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

For instance:

```
bin_PROGRAMS = hello
```

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

Associated with each program are several assisting variables 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 Yacc and Lex.

Node: Linking, Next: Conditional Sources, Previous: Program Sources, Up: A Program

Linking the program

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

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

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

```
bin_PROGRAMS = cpio pax @MT@
libexec_PROGRAMS = @RMT@
EXTRA_PROGRAMS = mt rmt

LDADD = ../lib/libcpio.a @INTLLIBS@
rmt_LDADD =

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

prog`_LDADD` is inappropriate for passing program-specific linker flags (except for `-l`, `-L`, `-dlopen` and `-dlpreopen`). So, use the *prog*`_LDFLAGS` variable for this purpose.

It is also occasionally useful to have a program depend on some other target 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`, `-L`, `-dlopen` and `-dlpreopen` options removed. The configure substitutions that are left in are only `@LIBOBJS@` and `@ALLOCA@`; these are left because it is known that they will not cause an invalid value for *prog*`_DEPENDENCIES` to be generated.

Node: Conditional Sources, Next: Conditional Programs, Previous: Linking, Up: A Program

Conditional compilation of sources

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.

Fortunately there are two other ways to achieve the same result. One is to use configure substitutions in `_LDADD` variables, the other is to use an Automake conditional.

Conditional compilation using `_LDADD` substitutions

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

```
bin_PROGRAMS = hello
hello_SOURCES = hello-common.c
EXTRA_hello_SOURCES = hello-linux.c hello-generic.c
hello_LDADD = @HELLO_SYSTEM@
hello_DEPENDENCIES = @HELLO_SYSTEM@
```

You can then setup the `@HELLO_SYSTEM@` substitution from `configure.in`:

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

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

Conditional compilation using Automake conditionals

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

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

In this case, your `configure.in` should setup the `LINUX` conditional using `AM_CONDITIONAL` (see Conditionals).

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

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

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

Node: Conditional Programs, Previous: Conditional Sources, Up: A Program

Conditional compilation of programs

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

Conditional programs using `configure` substitutions

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

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

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

Conditional programs using Automake conditionals

You can also use Automake conditionals (see Conditionals) to select programs to be built. In this case you don't have to worry about `$(EXEEXT)` or `EXTRA_PROGRAMS`.

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

Node: A Library, Next: A Shared Library, Previous: A Program, Up: Programs

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 A Shared Library, 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 Canonicalization), 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)
```

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

Node: A Shared Library, Next: Program and Library Variables, Previous: A Library, Up: Programs

Building a Shared Library

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

Node: Libtool Concept, Next: Libtool Libraries, Previous: A Shared Library, Up: A Shared Library

The Libtool Concept

Libtool abstracts shared and static libraries into a unified concept henceforth called *libtool libraries*. Libtool libraries are files using the `.la` suffix, and can designate a static library, a shared library, or maybe both. Their exact nature cannot be determined until `./configure` is run: not all platforms support all kinds of libraries, and users can explicitly select which libraries should be built. (However

the package's maintainers can tune the default, See The AC_PROG_LIBTOOL macro.)

Because object files for shared and static libraries must be compiled differently, libtool is also used during compilation. Object files built by libtool are called *libtool objects*: these are files using the `.lo` suffix. Libtool libraries are built from these libtool objects.

You should not assume anything about the structure of `.la` or `.lo` files and how libtool constructs them: this is libtool's concern, and the last thing one wants is to learn about libtool's guts. However the existence of these files matters, because they are used as targets and dependencies in `Makefile`s when building libtool libraries. There are situations where you may have to refer to these, for instance when expressing dependencies for building source files conditionally (see Conditional Libtool Sources).

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

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

Node: Libtool Libraries, Next: Conditional Libtool Libraries, Previous: Libtool Concept, Up: A Shared Library

Building Libtool Libraries

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

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

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

Node: Conditional Libtool Libraries, Next: Conditional Libtool Sources, Previous: Libtool Libraries, Up: A Shared Library

Building Libtool Libraries Conditionally

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

The important implementation detail you have to be aware of is that the place where a library will be installed matters to libtool: it needs to be indicated *at link-time* using the `-rpath` option.

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

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

The examples below illustrate the differences between these two methods.

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

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

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

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

```
libbar_la_SOURCES = bar.c ...
```

Node: Conditional Libtool Sources, Next: Libtool Convenience Libraries, Previous: Conditional Libtool Libraries, Up: A Shared Library

Libtool Libraries with Conditional Sources

Conditional compilation of sources in a library can be achieved in the same way as conditional compilation of sources in a program (see Conditional Sources). The only difference is that `_LIBADD` should be used instead of `_LDADD` and that it should mention libtool objects (`.lo` files).

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

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

And make sure `$(HELLO_SYSTEM)` is set to either `hello-linux.lo` or `hello-generic.lo` in `./configure`.

Or we could simply use an Automake conditional as follows.

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

Node: Libtool Convenience Libraries, Next: Libtool Modules, Previous: Conditional Libtool Sources, Up: A Shared Library

Libtool Convenience Libraries

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

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

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

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

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

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

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

Node: Libtool Modules, Next: Libtool Flags, Previous: Libtool Convenience Libraries, Up: A Shared Library

Libtool Modules

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

```
pkglib_LTLIBRARIES = mymodule.la
mymodule_la_SOURCES = doit.c
mymodule_LDFLAGS = -module
```

Ordinarily, Automake requires that a Library's name starts with `lib`. However, when building a dynamically loadable module you might wish to use a "nonstandard" name.

Node: Libtool Flags, Next: LTLIBOBJ, Previous: Libtool Modules, Up: A Shared Library

_LIBADD and _LDFLAGS

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

The *library*`_LDFLAGS` variable is the place to list additional libtool flags, such as `-version-info`, `-static`, and a lot more. See See Using libltdl.

Node: LTLIBOBJ, Next: Libtool Issues, Previous: Libtool Flags, Up: A Shared Library

LTLIBOBJS

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`.

Nowadays, the computation of `LTLIBOBJS` from `LIBOBJS` is performed automatically by Autoconf (see AC_LIBOBJ vs. LIBOBJS).

Node: Libtool Issues, Previous: LTLIBOBJ, Up: A Shared Library

Common Issues Related to Libtool's Use

`required file `./ltmain.sh' not found`

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

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

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

Objects `created with both libtool and without`

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

Let's consider the following `Makefile.am`.

```
bin_PROGRAMS = prog
prog_SOURCES = prog.c foo.c ...

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

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

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

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

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

A workaround for this issue is to ensure that these two objects get different basenames. As explained in renamed objects, this happens automatically when per-targets flags are used.

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

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

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

Node: Program and Library Variables, Next: LIBOBJS, Previous: A Shared Library, Up: Programs

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 Support for Other Languages). 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 Options).

`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. [5] 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 variable set like so:

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

`maude_LIBADD`

Extra objects can be added to a *library* using the `_LIBADD` variable. For instance this should be

used for objects determined by `configure` (see A Library).

`maude_LDADD`

Extra objects can be added to a *program* by listing them in the `_LDADD` variable. For instance this should be used for objects determined by `configure` (see Linking).

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

For instance, if your `configure.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_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` or `_LIBADD`, with most configure substitutions, `-l`, `-L`, `-dlopen` and `-dlpreopen` options removed. The configure substitutions that are left in are only `$(LIBOBJS)` and `$(ALLOCA)`; these are left because it is known that they will not cause an invalid value for `_DEPENDENCIES` to be generated.

`maude_LINK`

You can override the linker on a per-program basis. By default the linker is chosen according to the languages used by the program. For instance, a program that includes C++ source code would use the C++ compiler to link. The `_LINK` variable must hold the name of a command 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_CCASFLAGS`
`maude_CFLAGS`
`maude_CPPFLAGS`

maude_CXXFLAGS
maude_FFLAGS
maude_GCJFLAGS
maude_LFLAGS
maude_OBJCFLAGS
maude_RFLAGS
maude_YFLAGS

Automake allows you to set compilation flags on a per-program (or per-library) basis. A single source file can be included in several programs, and it will potentially be compiled with different flags for each program. This works for any language directly supported by Automake. These *per-target compilation flags* are _CCASFLAGS, _CFLAGS, _CPPFLAGS, _CXXFLAGS, _FFLAGS, _GCJFLAGS, _LFLAGS, _OBJCFLAGS, _RFLAGS, and _YFLAGS.

When using a per-target compilation flag, Automake will choose a different name for the intermediate object files. Ordinarily a file like sample.c will be compiled to produce sample.o. However, if the program's _CFLAGS variable is set, then the object file will be named, for instance, maude-sample.o. (See also renamed objects.)

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

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

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

maude_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.

Node: LIBOBJS, Next: Program variables, Previous: Program and Library Variables, Up: Programs

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 Dist). These source files are also automatically handled in the dependency-tracking scheme; see See Dependencies.

$(LIBOBJS) and $(ALLOCA) are specially recognized in any _LDADD or _LIBADD variable.

Node: Program variables, Next: Yacc and Lex, Previous: LIBOBJS, Up: Programs

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 variable 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_HEADERS or AM_CONFIG_HEADER). You can disable the default -I options using the nostdinc option.

  AM_CPPFLAGS is ignored in preference to a per-executable (or per-library) _CPPFLAGS variable if it is defined.

`INCLUDES`
: This does the same job as `AM_CPPFLAGS`. It is an older name for the same functionality. This variable 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.

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

`LINK`
: This is the command used to actually link a C program. It already includes `-o $@` and the usual variable references (for instance, `CFLAGS`); it takes as "arguments" the names of the object files and libraries to link in.

Node: Yacc and Lex, Next: C++ Support, Previous: Program variables, Up: Programs

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 Particular Program Checks).

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

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

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

If a `lex` source file is seen, then your `configure.in` must define the variable `LEX`. You can use `AC_PROG_LEX` to do this (see Particular Program Checks), but using `AM_PROG_LEX` macro (see Macros) is recommended.

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

Automake makes it possible to include multiple `yacc` (or `lex`) source files in a single program. When there is more than one distinct `yacc` (or `lex`) source file in a directory, Automake uses a small program called `ylwrap` to run `yacc` (or `lex`) in a subdirectory. This is necessary because yacc's output 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 Finding `configure' Input), 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 yyname   c_yyname
#define yyrule   c_yyrule
```

For each define, replace the `c_` prefix with whatever you like. These defines work for `bison`, `byacc`, and traditional `yacc`s. If you find a parser generator that uses a symbol not covered here, please report

the new name so it can be added to the list.

Node: C++ Support, Next: Assembly Support, Previous: Yacc and Lex, Up: Programs

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 Particular Program Checks).

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.

Node: Assembly Support, Next: Fortran 77 Support, Previous: C++ Support, Up: Programs

Assembly Support

Automake includes some support for assembly code.

The variable `CCAS` holds the name of the compiler used to build assembly code. This compiler must work a bit like a C compiler; in particular it must accept `-c` and `-o`. The value of `CCASFLAGS` is passed to the compilation.

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

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

Node: Fortran 77 Support, Next: Java Support, Previous: Assembly Support, Up: Programs

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 Particular Program Checks). See Fortran 77 and Autoconf.

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

`F77`
: The name of the Fortran 77 compiler.

`FFLAGS`
: Any flags to pass to the Fortran 77 compiler.

`AM_FFLAGS`
: The maintainer's variant of `FFLAGS`.

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

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

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

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

Automake can handle preprocessing Fortran 77 and Ratfor source files in addition to compiling them[6]. Automake also contains some support for creating programs and shared libraries that are a mixture of Fortran 77 and other languages (see Mixing Fortran 77 With C and C++).

These issues are covered in the following sections.

- Preprocessing Fortran 77:
- Compiling Fortran 77 Files:
- Mixing Fortran 77 With C and C++:

- Fortran 77 and Autoconf:

Node: Preprocessing Fortran 77, Next: Compiling Fortran 77 Files, Previous: Fortran 77 Support, Up: Fortran 77 Support

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)
```

Node: Compiling Fortran 77 Files, Next: Mixing Fortran 77 With C and C++, Previous: Preprocessing Fortran 77, Up: Fortran 77 Support

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)
```

Node: Mixing Fortran 77 With C and C++, Next: Fortran 77 and Autoconf, Previous: Compiling Fortran 77 Files, Up: Fortran 77 Support

Mixing Fortran 77 With C and C++

Automake currently provides *limited* support for creating programs and shared libraries that are a mixture of Fortran 77 and C and/or C++. However, there are many other issues related to mixing Fortran 77 with other languages that are *not* (currently) handled by Automake, but that are handled by other packages[7].

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 Fortran 77 Compiler Characteristics.

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.

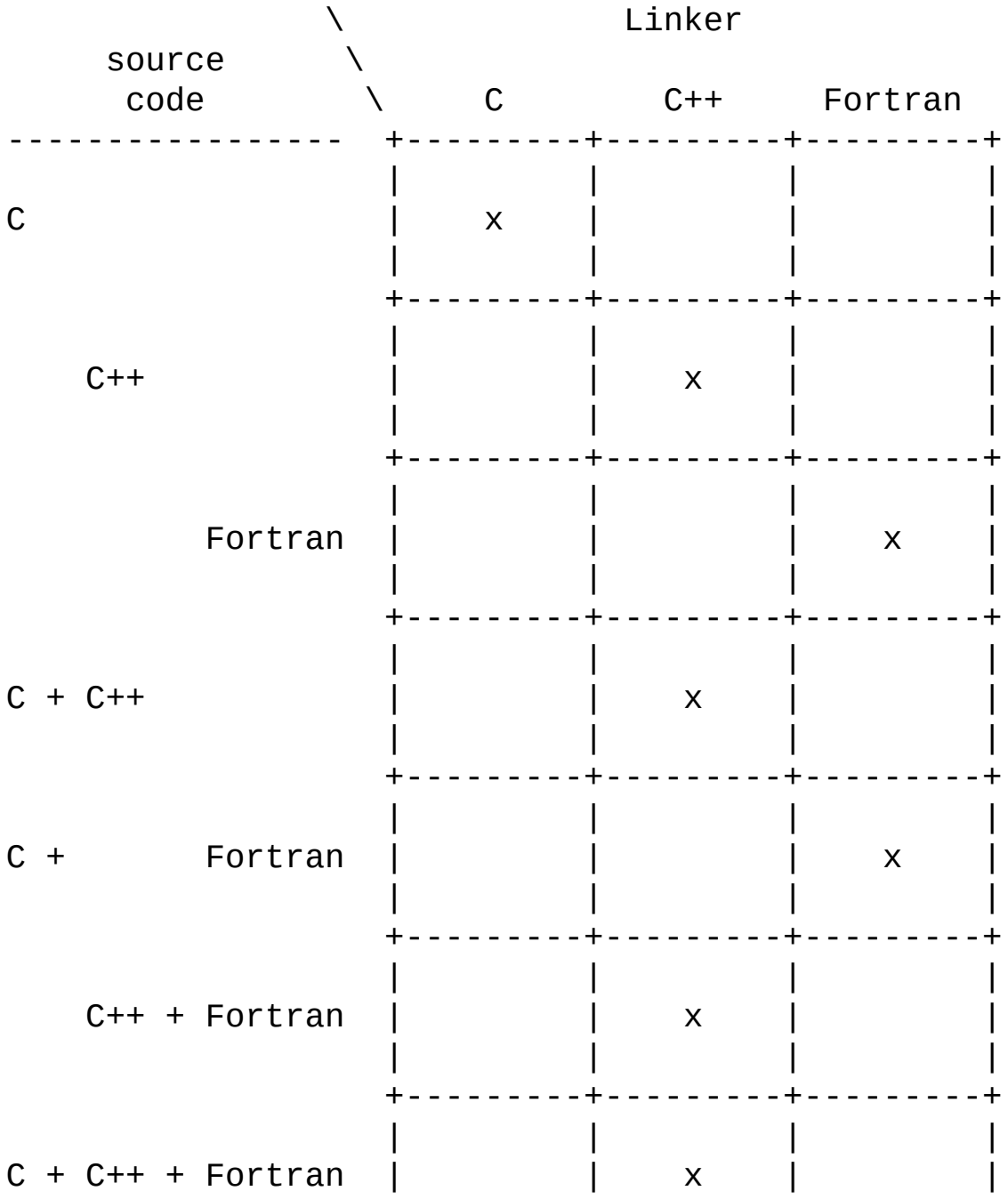
- How the Linker is Chosen:

Node: How the Linker is Chosen, Previous: Mixing Fortran 77 With C and C++, Up: Mixing Fortran 77 With C and C++

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`.



Node: Fortran 77 and Autoconf, Previous: Mixing Fortran 77 With C and C++, Up: Fortran 77 Support

Fortran 77 and Autoconf

The current Automake support for Fortran 77 requires a recent enough version of 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.

Node: Java Support, Next: Support for Other Languages, Previous: Fortran 77 Support, Up: Programs

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 variable, and not the user variable `GCJFLAGS`, should be used.

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

When linking, `gcj` requires that the main class be specified using the `--main=` option. The easiest way to do this is to use the `_LDFLAGS` variable for the program.

Node: Support for Other Languages, Next: ANSI, Previous: Java Support, Up: Programs

Support for Other Languages

Automake currently only includes full support for C, C++ (see C++ Support), Fortran 77 (see Fortran 77 Support), and Java (see Java Support). 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 Suffixes.

Node: ANSI, Next: Dependencies, Previous: Support for Other Languages, Up: Programs

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 Options) 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 Macros).

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

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

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

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

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

Node: Dependencies, Next: EXEEXT, Previous: ANSI, Up: Programs

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 [8] taught us that it is not reliable to generate dependencies only on the maintainer's system, as configurations vary too much. So instead Automake implements dependency tracking at build time.

Automatic dependency tracking can be suppressed by putting `no-dependencies` in the variable `AUTOMAKE_OPTIONS`, or passing `no-dependencies` as an argument to `AM_INIT_AUTOMAKE` (this should be the preferred way). Or, you can invoke `automake` with the `-i` option. Dependency tracking is enabled by default.

The person building your package also can choose to disable dependency tracking by configuring with `--disable-dependency-tracking`.

Node: EXEEXT, Previous: Dependencies, Up: Programs

Support for executable extensions

On some platforms, such as Windows, executables are expected to have an extension such as `.exe`. On these platforms, some compilers (GCC among them) will automatically generate `foo.exe` when asked to generate `foo`.

Automake provides mostly-transparent support for this. Unfortunately *mostly* doesn't yet mean *fully*. Until the English dictionary is revised, you will have to assist Automake if your package must support those platforms.

One thing you must be aware of is that, internally, Automake rewrites something like this:

```
bin_PROGRAMS = liver
```

to this:

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

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

Node: Other objects, Next: Other GNU Tools, Previous: Programs, Up: Top

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.

Node: Scripts, Next: Headers, Previous: Other objects, Up: Other objects

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 Clean).

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, and it is also automatically cleaned (despite the fact it's a script).

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

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

Node: Headers, Next: Data, Previous: Scripts, Up: Other objects

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. [9]

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 A Program. Headers listed in a `_SOURCES` variable need not be listed in any `_HEADERS` variable.

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

Node: Data, Next: Sources, Previous: Headers, Up: Other objects

Architecture-independent data files

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

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

By default, data files are *not* included in a distribution. Of course, you can use the `dist_` prefix to change this on a per-variable basis.

Here is how Automake declares its auxiliary data files:

```
dist_pkgdata_DATA = clean-kr.am clean.am ...
```

Node: Sources, Previous: Data, Up: Other objects

Built sources

Because Automake's automatic dependency tracking works as a side-effect of compilation (see Dependencies) there is a bootstrap issue: a target should not be compiled before its dependencies are made, but these dependencies are unknown until the target is first compiled.

Ordinarily this is not a problem, because dependencies are distributed sources: they preexist and do not need to be built. Suppose that `foo.c` includes `foo.h`. When it first compiles `foo.o`, `make` only knows that `foo.o` depends on `foo.c`. As a side-effect of this compilation `depcomp` records the `foo.h` dependency so that following invocations of `make` will honor it. In these conditions, it's clear there is no problem: either `foo.o` doesn't exist and has to be built (regardless of the dependencies), either accurate dependencies exist and they can be used to decide whether `foo.o` should be rebuilt.

It's a different story if `foo.h` doesn't exist by the first `make` run. For instance there might be a rule to build `foo.h`. This time `file.o`'s build will fail because the compiler can't find `foo.h`. `make` failed to trigger the rule to build `foo.h` first by lack of dependency information.

The `BUILT_SOURCES` variable is a workaround for this problem. A source file listed in `BUILT_SOURCES` is made on `make all` or `make check` (or even `make install`) before other targets are processed. However, such a source file is not *compiled* unless explicitly requested by mentioning it in some other `_SOURCES` variable.

So, to conclude our introductory example, we could use `BUILT_SOURCES = foo.h` to ensure `foo.h` gets built before any other target (including `foo.o`) during `make all` or `make check`.

`BUILT_SOURCES` is actually a bit of a misnomer, as any file which must be created early in the build process can be listed in this variable. Moreover, all built sources do not necessarily have to be listed in `BUILT_SOURCES`. For instance a generated `.c` file doesn't need to appear in `BUILT_SOURCES` (unless it is included by another source), because it's a known dependency of the associated object.

It might be important to emphasize that `BUILT_SOURCES` is honored only by `make all`, `make check` and `make install`. This means you cannot build a specific target (e.g., `make foo`) in a clean tree if it depends on a built source. However it will succeed if you have run `make all` earlier, because accurate dependencies are already available.

The next section illustrates and discusses the handling of built sources on a toy example.

Node: Built sources example, Previous: Sources, Up: Sources

Built sources example

Suppose that `foo.c` includes `bindir.h`, which is installation-dependent and not distributed: it needs to be built. Here `bindir.h` defines the preprocessor macro `bindir` to the value of the `make` variable `bindir` (inherited from `configure`).

We suggest several implementations below. It's not meant to be an exhaustive listing of all ways to handle built sources, but it will give you a few ideas if you encounter this issue.

First try

This first implementation will illustrate the bootstrap issue mentioned in the previous section (see Sources).

Here is a tentative `Makefile.am`.

```
# This won't work.
bin_PROGRAMS = foo
foo_SOURCES = foo.c
nodist_foo_SOURCES = bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

This setup doesn't work, because Automake doesn't know that `foo.c` includes `bindir.h`. Remember, automatic dependency tracking works as a side-effect of compilation, so the dependencies of `foo.o` will be known only after `foo.o` has been compiled (see Dependencies). The symptom is as follows.

```
% make
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

Using `BUILT_SOURCES`

A solution is to require `bindir.h` to be built before anything else. This is what `BUILT_SOURCES` is meant for (see Sources).

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
BUILT_SOURCES = bindir.h
CLEANFILES = bindir.h
```

```
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

See how `bindir.h` get built first:

```
% make
echo '#define bindir "/usr/local/bin"' >bindir.h
make  all-am
make[1]: Entering directory `/home/adl/tmp'
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
gcc  -g -O2   -o foo  foo.o
make[1]: Leaving directory `/home/adl/tmp'
```

However, as said earlier, `BUILT_SOURCES` applies only to the `all`, `check`, and `install` targets. It still fails if you try to run `make foo` explicitly:

```
% make clean
test -z "bindir.h" || rm -f bindir.h
test -z "foo" || rm -f foo
rm -f *.o core *.core
% : > .deps/foo.Po # Suppress previously recorded dependencies
% make foo
source='foo.c' object='foo.o' libtool=no \
depfile='.deps/foo.Po' tmpdepfile='.deps/foo.TPo' \
depmode=gcc /bin/sh ./depcomp \
gcc -I. -I. -g -O2 -c `test -f 'foo.c' || echo './'`foo.c
foo.c:2: bindir.h: No such file or directory
make: *** [foo.o] Error 1
```

Recording dependencies manually

Usually people are happy enough with `BUILT_SOURCES` because they never run targets such as `make foo` before `make all`, as in the previous example. However if this matters to you, you can avoid `BUILT_SOURCES` and record such dependencies explicitly in the `Makefile.am`.

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

```
foo.$(OBJEXT): bindir.h
CLEANFILES = bindir.h
bindir.h: Makefile
        echo '#define bindir "$(bindir)"' >$@
```

You don't have to list *all* the dependencies of `foo.o` explicitly, only those which might need to be built. If a dependency already exists, it will not hinder the first compilation and will be recorded by the normal dependency tracking code. (Note that after this first compilation the dependency tracking code will also have recorded the dependency between `foo.o` and `bindir.h`; so our explicit dependency is really useful to the first build only.)

Adding explicit dependencies like this can be a bit dangerous if you are not careful enough. This is due to the way Automake tries not to overwrite your rules (it assumes you know better than it). `foo.$(OBJEXT): bindir.h` supersedes any rule Automake may want to output to build `foo.$(OBJEXT)`. It happens to work in this case because Automake doesn't have to output any `foo.$(OBJEXT):` target: it relies on a suffix rule instead (i.e., `.c.$(OBJEXT):`). Always check the generated `Makefile.in` if you do this.

Build `bindir.h` from `configure`

It's possible to define this preprocessor macro from `configure`, either in `config.h` (see Defining Directories), or by processing a `bindir.h.in` file using `AC_CONFIG_FILES` (see Configuration Actions).

At this point it should be clear that building `bindir.h` from `configure` work well for this example. `bindir.h` will exist before you build any target, hence will not cause any dependency issue.

The Makefile can be shrunk as follows. We do not even have to mention `bindir.h`.

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

However, it's not always possible to build sources from `configure`, especially when these sources are generated by a tool that needs to be built first...

Build `bindir.c`, not `bindir.h`.

Another attractive idea is to define `bindir` as a variable or function exported from `bindir.o`, and build `bindir.c` instead of `bindir.h`.

```
noinst_PROGRAMS = foo
foo_SOURCES = foo.c bindir.h
nodist_foo_SOURCES = bindir.c
CLEANFILES = bindir.c
bindir.c: Makefile
        echo 'const char bindir[] = "$(bindir)";' >$
```

`bindir.h` contains just the variable's declaration and doesn't need to be built, so it won't cause any trouble. `bindir.o` is always dependent on `bindir.c`, so `bindir.c` will get built first.

Which is best?

There is no panacea, of course. Each solution has its merits and drawbacks.

You cannot use `BUILT_SOURCES` if the ability to run `make foo` on a clean tree is important to you.

You won't add explicit dependencies if you are leery of overriding an Automake target by mistake.

Building files from `./configure` is not always possible, neither is converting `.h` files into `.c` files.

Node: Other GNU Tools, Next: Documentation, Previous: Other objects, Up: Top

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.

- Emacs Lisp: Emacs Lisp
- gettext: Gettext
- Libtool: Libtool
- Java: Java
- Python: Python

Node: Emacs Lisp, Next: gettext, Previous: Other GNU Tools, Up: Other GNU Tools

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 Macros).

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.

Lisp sources are not distributed by default. You can prefix the `LISP` primary with `dist_`, as in `dist_lisp_LISP` or `dist_noinst_LISP`, to indicate that these files should be distributed.

Node: gettext, Next: Libtool, Previous: Emacs Lisp, Up: Other GNU Tools

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 GNU Gettext).

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`.

Node: Libtool, Next: Java, Previous: gettext, Up: Other GNU Tools

Libtool

Automake provides support for GNU Libtool (see Introduction) with the `LTLIBRARIES` primary. See A Shared Library.

Node: Java, Next: Python, Previous: Libtool, Up: Other GNU Tools

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 User Variables).

`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.)

Node: Python, Previous: Java, Up: Other GNU Tools

Python

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

Any files listed in a `_PYTHON` variable will be byte-compiled with `py-compile` at install time. `py-compile` actually creates both standard (`.pyc`) and byte-compiled (`.pyo`) versions of the source files. Note that because byte-compilation occurs at install time, any files listed in `noinst_PYTHON` will not be compiled. Python source files are included in the distribution by default.

Automake ships with an Autoconf macro called `AM_PATH_PYTHON` which will determine some Python-related directory variables (see below). If you have called `AM_PATH_PYTHON` from `configure.in`, then you may use the following variables to list you Python source files in your variables: `python_PYTHON`, `pkgpython_PYTHON`, `pyexecdir_PYTHON`, `pkgpyexecdir_PYTHON`,

depending where you want your files installed.

`AM_PATH_PYTHON` 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 thePython 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 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)`.

All these directory variables have values that start with either `${prefix}` or `${exec_prefix}` unexpanded. This works fine in `Makefiles`, but it makes these variables hard to use in `configure`. This is mandated by the GNU coding standards, so that the user can run `make prefix=/foo install`. The Autoconf manual has a section with more details on this topic (see Installation Directory Variables).

Node: Documentation, Next: Install, Previous: Other GNU Tools, Up: Top

Building documentation

Currently Automake provides support for Texinfo and man pages.

- Texinfo: Texinfo
- Man pages: Man pages

Node: Texinfo, Next: Man pages, Previous: Documentation, Up: Documentation

Texinfo

If the current directory contains Texinfo source, you must declare it with the `TEXINFOS` primary. Generally Texinfo files are converted into info, and thus the `info_TEXINFOS` variable is most commonly used here. Any Texinfo source file must end in the `.texi`, `.txi`, or `.texinfo` extension. We recommend `.texi` for new manuals.

Automake generates rules to build `.info`, `.dvi`, `.ps`, and `.pdf` files from your Texinfo sources. The `.info` files are built by `make all` and installed by `make install` (unless you use `no-installinfo`, see below). The other files can be built on request by `make dvi`, `make ps`, and `make pdf`.

If the `.texi` file `@includes` `version.texi`, then that file will be automatically generated. The file `version.texi` defines four Texinfo flag you can reference using `@value{EDITION}`, `@value{VERSION}`, `@value{UPDATED}`, and `@value{UPDATED-MONTH}`.

`EDITION`
`VERSION`
: Both of these flags hold the version number of your program. They are kept separate for clarity.

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

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

The `version.texi` support requires the `mdate-sh` 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 User Variables) 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 Finding `configure' Input), 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`, `ps`, and `pdf` targets 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.

Node: Man pages, Previous: Texinfo, Up: Documentation

Man pages

A package can also include man pages (but see the GNU standards on this matter, Man Pages.) Man pages are declared using the `MANS` primary. Generally the `man_MANS` variable is used. Man pages are automatically installed in the correct subdirectory of `mandir`, based on the file extension.

File extensions such as `.1c` are handled by looking for the valid part of the extension and using that to determine the correct subdirectory of `mandir`. Valid section names are the digits `0` through `9`, and the letters `l` and `n`.

Sometimes developers prefer to name a man page something like `foo.man` in the source, and then rename it to have the correct suffix, e.g. `foo.1`, when installing the file. Automake also supports this mode. For a valid section named *SECTION*, there is a corresponding directory named `man`*SECTION*`dir`, and a corresponding `_MANS` variable. Files listed in such a variable are installed in the indicated section. If the file already has a valid suffix, then it is installed as-is; otherwise the file suffix is changed to match the section.

For instance, consider this example:

```
man1_MANS = rename.man thesame.1 alsothesame.1c
```

In this case, `rename.man` will be renamed to `rename.1` when installed, but the other files will keep their names.

By default, man pages are installed by `make install`. However, since the GNU project does not require man pages, many maintainers do not expend effort to keep the man pages up to date. In these cases, the `no-installman` option will prevent the man pages from being installed by default. The user can still explicitly install them via `make install-man`.

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

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

Man pages are not currently considered to be source, because it is not uncommon for man pages to be automatically generated. Therefore they are not automatically included in the distribution. However, this can be changed by use of the `dist_` prefix.

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

Node: Install, Next: Clean, Previous: Documentation, Up: Top

What Gets Installed

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`.

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`.

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.

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 Makefile Conventions.

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`.

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 these features would not provide additional functionality.

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

Node: Clean, Next: Dist, Previous: Install, Up: Top

What Gets Cleaned

The GNU Makefile Standards specify a number of different clean rules. See See Standard Targets for Users.

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 (for instance, a `.info` file), then `maintainer-clean` should delete it. However `maintainer-clean` should not delete anything that needs to exist in order to run `./configure && make`.

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

Node: Dist, Next: Tests, Previous: Clean, Up: Top

What Goes in a Distribution

Basics of distribution

The `dist` target in the generated `Makefile.in` can be used to generate a gzip'd `tar` file and other flavors of archive for distribution. The files is named based on the `PACKAGE` and `VERSION` variables defined by `AM_INIT_AUTOMAKE` (see Macros); more precisely the gzip'd `tar` file is named *package*`-`*version*`.tar.gz`. You can use the `make` variable `GZIP_ENV` to control how gzip is run. The default setting is `--best`.

For the most part, the files to distribute are automatically found by Automake: all source files are automatically included in a distribution, as are all `Makefile.am`s and `Makefile.in`s. Automake also has a built-in list of commonly used files which are automatically included if they are found in the current directory (either physically, or as the target of a `Makefile.am` rule). This list is printed by `automake --help`. Also, files which are read by `configure` (i.e. the source files corresponding to the files specified in various Autoconf macros such as `AC_CONFIG_FILES` and siblings) are automatically distributed. Helper scripts installed with `automake --add-missing` are also 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.

If you define `SUBDIRS`, Automake will recursively include the subdirectories in the distribution. If `SUBDIRS` is defined conditionally (see Conditionals), Automake will normally include all directories that could possibly appear in `SUBDIRS` in the distribution. If you need to specify the set of directories conditionally, you can set the variable `DIST_SUBDIRS` to the exact list of subdirectories to include in the distribution (see Top level).

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
```

The dist hook

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
```

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

```
EXTRA_DIST = doc

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

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, then tries to do a `VPATH` build, run the test suite, and finally make another tarfile to ensure the distribution is self-contained.

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

If the 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.

Speaking about potential distribution errors, `distcheck` will also ensure that the `distclean` target actually removes all built files. This is done by running `make distcleancheck` at the end of the `VPATH` build. By default, `distcleancheck` will run `distclean` and then make sure the build tree has been emptied by running `$(distcleancheck_listfiles)`. Usually this check will find generated files that you forgot to add to the `DISTCLEANFILES` variable (see Clean).

The `distcleancheck` behavior should be OK for most packages, otherwise you have the possibility to override the definition of either the `distcleancheck` target, or the `$(distcleancheck_listfiles)` variable. For instance to disable `distcleancheck` completely, add the following rule to your top-level `Makefile.am`:

```
distcleancheck:
        @:
```

If you want `distcleancheck` to ignore built files which have not been cleaned because they are also part of the distribution, add the following definition instead:

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

The above definition is not the default because it's usually an error if your Makefiles cause some distributed files to be rebuilt when the user build the package. (Think about the user missing the tool required to build the file; or if the required tool is built by your package, consider the cross-compilation case where it can't be run.) There is a FAQ entry about this (see distcleancheck), make sure you read it before playing with `distcleancheck_listfiles`.

`distcheck` also checks that the `uninstall` target works properly, both for ordinary and `DESTDIR` builds. It does this by invoking `make uninstall`, and then it checks the install tree to see if any files are left over. This check will make sure that you correctly coded your `uninstall`-related targets.

By default, the checking is done by the `distuninstallcheck` target, and the list of files in the install tree is generated by `$(distuninstallcheck_listfiles)` (this is a variable whose value is a shell command to run that prints the list of files to stdout).

Either of these can be overridden to modify the behavior of `distcheck`. For instance, to disable this check completely, you would write:

```
distuninstallcheck:
        @:
```

The types of distributions

Automake generates a `.tar.gz` file when asked to create a distribution and other archives formats, Options. The target `dist-gzip` generates the `.tar.gz` file only.

Node: Tests, Next: Options, Previous: Dist, Up: Top

Support for test suites

Automake supports two forms of test suites.

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 Build Directories), and in particular for the `distcheck` target (see Dist).

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.

DejaGnu Tests

If `dejagnu` 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 User Variables). If you need to set `runtest` flags in `Makefile.am`, you can use AM_RUNTESTFLAGS instead.

Automake will generate rules to create a local `site.exp` file, defining various variables detected by `./configure`. This file is automatically read by DejaGnu. It is OK for the user of a package to edit this file in order to tune the test suite. However this is not the place where the test suite author should define new variables: this should be done elsewhere in the real test suite code. Especially, `site.exp` should not be distributed.

For more information regarding DejaGnu test suites, see See Top.

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

Install Tests

The `installcheck` target is available to the user as a way to run any tests after the package has been installed. You can add tests to this by writing an `installcheck-local` target.

Node: Options, Next: Miscellaneous, Previous: Tests, Up: Top

Changing Automake's Behavior

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

`dist-bzip2`
: Generate a `dist-bzip2` target, creating a bzip2 tar archive of the distribution. `dist` will create it in addition to the other formats. bzip2 archives are frequently smaller than gzipped archives.

`dist-shar`
: Generate a `dist-shar` target, creating a shar archive of the distribution. `dist` will create it in addition to the other formats.

`dist-zip`
: Generate a `dist-zip` target, creating a zip archive of the distribution. `dist` will create it in addition to the other formats.

`dist-tarZ`
: Generate a `dist-tarZ` target, creating a compressed tar archive of the distribution. `dist` will create it in addition to the other formats.

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

`no-dependencies`
: This is similar to using `--include-deps` on the command line, but is useful for those situations where you don't have the necessary bits to make automatic dependency tracking work See Dependencies. 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.

`std-options`
: Make the `installcheck` target check that installed scripts and programs support the `--help` and `--version` options. This also provides a basic check that the program's run-time dependencies are satisfied after installation.

  In a few situations, programs (or scripts) have to be exempted from this test. For instance `false` (from GNU sh-utils) is never successful, even for `--help` or `--version`. You can list such programs in the variable `AM_INSTALLCHECK_STD_OPTIONS_EXEMPT`. Programs (not scripts) listed in this variable should be suffixed by `$(EXEEXT)` for the sake of Win32 or OS/2. For instance suppose we build `false` as a program but `true.sh` as a script, and that neither of them support `--help` or `--version`:

```
AUTOMAKE_OPTIONS = std-options
bin_PROGRAMS = false ...
bin_SCRIPTS = true.sh ...
AM_INSTALLCHECK_STD_OPTIONS_EXEMPT = false$(EXEEXT)
```

  `true.sh`

`subdir-objects`
: If this option is specified, then objects are placed into the subdirectory of the build directory corresponding to the subdirectory of the source file. For instance if the source file is `subdir/file.cxx`, then the output file would be `subdir/file.o`.

version
: A version number (e.g. `0.30`) can be specified. If Automake is not newer than the version specified, creation of the `Makefile.in` will be suppressed.

`-W`*category* or `--warnings=`*category*
: These options behave exactly like their command-line counterpart (see Invoking Automake).

This allows you to enable or disable some warning categories on a per-file basis. You can also setup some warnings for your entire project; for instance try `AM_INIT_AUTOMAKE([-Wall])` in your `configure.in`.

Unrecognized options are diagnosed by `automake`.

If you want an option to apply to all the files in the tree, you can use the `AM_INIT_AUTOMAKE` macro in `configure.in`. See Macros.

Node: Miscellaneous, Next: Include, Previous: Options, Up: Top

Miscellaneous Rules

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

- Tags: Interfacing to etags and mkid
- Suffixes: Handling new file extensions
- Multilibs: Support for multilibs.

Node: Tags, Next: Suffixes, Previous: Miscellaneous, Up: Miscellaneous

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. The user can use the `ETAGSFLAGS` to pass additional flags to `etags`; `AM_ETAGSFLAGS` is also available for use in `Makefile.am`.

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

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

If you add 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 also generates a `ctags` target which can be used to build `vi`-style `tags` files. The variable `CTAGS` is the name of the program to invoke (by default `ctags`); `CTAGSFLAGS` can be used by the user to pass additional flags, and `AM_CTAGSFLAGS` can be used by the `Makefile.am`.

Automake will also generate an `ID` 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. 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`.

Node: Suffixes, Next: Multilibs, Previous: Tags, Up: Miscellaneous

Handling new file extensions

It is sometimes useful to introduce a new implicit rule to handle a file type that Automake does not know about.

For instance, suppose you had a compiler which could compile `.foo` files to `.o` files. You would simply define an suffix rule for your language:

```
.foo.o:
        foocc -c -o $@ $<
```

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

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

This was the simpler and more common case. In other cases, you will have to help Automake to figure which extensions you are defining your suffix rule for. This usually happens when your extensions does

not start with a dot. Then, all you have to do is to put a list of new suffixes in the SUFFIXES variable **before** you define your implicit rule.

For instance the following definition prevents Automake to misinterpret .idlC.cpp: as an attempt to transform .idlC into .cpp.

```
SUFFIXES = .idl C.cpp
.idlC.cpp:
        # whatever
```

As you may have noted, the SUFFIXES variable behaves like the .SUFFIXES special target of make. You should not touch .SUFFIXES yourself, but use SUFFIXES instead and let Automake generate the suffix list for .SUFFIXES. Any given SUFFIXES go at the start of the generated suffixes list, followed by Automake generated suffixes not already in the list.

Node: Multilibs, Previous: Suffixes, Up: Miscellaneous

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.

Node: Include, Next: Conditionals, Previous: Miscellaneous, Up: Top

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.

Makefile fragments included this way are always distributed because there are needed to rebuild `Makefile.in`.

Node: Conditionals, Next: Gnits, Previous: Include, Up: Top

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 Macros).

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 Conditional Programs).

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.

Node: Gnits, Next: Cygnus, Previous: Conditionals, Up: Top

The effect of `--gnu` and `--gnits`

The `--gnu` option (or `gnu` in the `AUTOMAKE_OPTIONS` variable) causes `automake` to check the following:

- The files `INSTALL`, `NEWS`, `README`, `AUTHORS`, and `ChangeLog`, plus one of `COPYING.LIB`, `COPYING.LESSER` or `COPYING`, are required at the topmost directory of the package.
- The options `no-installman` and `no-installinfo` are prohibited.

Note that this option will be extended in the future to do even more checking; it is advisable to be familiar with the precise requirements of the GNU standards. Also, `--gnu` can require certain non-standard GNU programs to exist for use by various maintainer-only 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 installcheck` will check to make sure that the `--help` and `--version` really print a usage message and a version string, respectively. This is the `std-options` option (see Options).
- `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.

Node: Cygnus, Next: Extending, Previous: Gnits, Up: Top

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.

Node: Extending, Next: Distributing, Previous: Cygnus, Up: Top

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`, `ps`, `pdf`, `check`, `install-data`, `install-exec`, `uninstall`, `installdirs`, `installcheck` and the various `clean` targets (`mostlyclean`, `clean`, `distclean`, and `maintainer-clean`). Note that there are no `uninstall-exec-local` or `uninstall-data-local` targets; just use `uninstall-local`. It doesn't make sense to uninstall just data or just executables.

For instance, here is one way to install a file in `/etc`:

```
install-data-local:
        $(INSTALL_DATA) $(srcdir)/afile $(DESTDIR)/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`, `uninstall`, `dist`, and `distcheck`.

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

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

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

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

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

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

Node: Distributing, Next: API versioning, Previous: Extending, Up: Top

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.

Node: API versioning, Next: FAQ, Previous: Distributing, Up: Top

Automake API versioning

New Automake releases usually include bug fixes and new features. Unfortunately they may also introduce new bugs and incompatibilities. This makes four reasons why a package may require a particular Automake version.

Things get worse when maintaining a large tree of packages, each one requiring a different version of Automake. In the past, this meant that any developer (and sometime users) had to install several versions of Automake in different places, and switch `$PATH` appropriately for each package.

Starting with version 1.6, Automake installs versioned binaries. This means you can install several versions of Automake in the same `$prefix`, and can select an arbitrary Automake version by running `automake-1.6` or `automake-1.7` without juggling with `$PATH`. Furthermore, `Makefile`'s generated by Automake 1.6 will use `automake-1.6` explicitly in their rebuild rules.

Note that `1.6` in `automake-1.6` is Automake's API version, not Automake's version. If a bug fix release is made, for instance Automake 1.6.1, the API version will remain 1.6. This means that a package which work with Automake 1.6 should also work with 1.6.1; after all, this is what people expect from bug fix releases.

Note that if your package relies on a feature or a bug fix introduced in a release, you can pass this version as an option to Automake to ensure older releases will not be used. For instance, use this in your `configure.in`:

```
AM_INIT_AUTOMAKE(1.6.1)    dnl Require Automake 1.6.1 or
better.
```

or, in a particular `Makefile.am`:

```
AUTOMAKE_OPTIONS = 1.6.1   # Require Automake 1.6.1 or better.
```

Automake will print an error message if its version is older than the requested version.

What is in the API

Automake's programming interface is not easy to define. Basically it should include at least all **documented** variables and targets that a `Makefile.am` author can use, any behavior associated with them (e.g. the places where `-hook`'s are run), the command line interface of `automake` and `aclocal`, ...

What is not in the API

Every undocumented variable, target, or command line option, is not part of the API. You should avoid using them, as they could change from one version to the other (even in bug fix releases, if this helps to fix a bug).

If it turns out you need to use such a undocumented feature, contact automake@gnu.org and try to get it documented and exercised by the test-suite.

Node: FAQ, Next: Macro and Variable Index, Previous: API versioning, Up: Top

Frequently Asked Questions about Automake

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

Node: CVS, Next: maintainer-mode, Previous: FAQ, Up: FAQ

CVS and generated files

Background: distributed generated files

Packages made with Autoconf and Automake ship with some generated files like `configure` or `Makefile.in`. These files were generated on the developer's host and are distributed so that end-users do not have to install the maintainer tools required to rebuild them. Other generated files like Lex scanners, Yacc parsers, or Info documentation, are usually distributed on similar grounds.

Automake outputs rules in `Makefile`s to rebuild these files. For instance `make` will run `autoconf` to rebuild `configure` whenever `configure.in` is changed. This makes development safer by ensuring a `configure` is never out-of-date with respect to `configure.in`.

As generated files shipped in packages are up-to-date, and because `tar` preserves timestamps, these

rebuild rules are not triggered when a user unpacks and builds a package.

Background: CVS and timestamps

Unless you use CVS keywords (in which case files must be updated at commit time), CVS preserves timestamps during `cvs commit` and `cvs import -d` operations.

When you check out a file using `cvs checkout` its timestamp is set to that of the revision which is being checked out.

However, during `cvs update`, files will have the date of the update, not the original timestamp of this revision. This is meant to make sure that `make` notices sources files have been updated.

This timestamp shift is troublesome when both sources and generated files are kept under CVS. Because CVS processes files in alphabetical order, `configure.in` will appear older than `configure` after a `cvs update` that updates both files, even if `configure` was newer than `configure.in` when it was checked in. Calling `make` will then trigger a spurious rebuild of `configure`.

Living with CVS in Autoconfiscated projects

There are basically two clans amongst maintainers: those who keep all distributed files under CVS, including generated files, and those who keep generated files *out* of CVS.

All files in CVS

- The CVS repository contains all distributed files so you know exactly what is distributed, and you can checkout any prior version entirely.
- Maintainers can see how generated files evolve (for instance you can see what happens to your `Makefile.in`s when you upgrade Automake and make sure they look OK).
- Users do not need the autotools to build a checkout of the project, it works just like a released tarball.
- If users use `cvs update` to update their copy, instead of `cvs checkout` to fetch a fresh one, timestamps will be inaccurate. Some rebuild rules will be triggered and attempt to run developer tools such as `autoconf` or `automake`.

  Actually, calls to such tools are all wrapped into a call to the `missing` script discussed later (see maintainer-mode). `missing` will take care of fixing the timestamps when these tools are not installed, so that the build can continue.

- In distributed development, developers are likely to have different version of the maintainer tools installed. In this case rebuilds triggered by timestamp lossage will lead to spurious changes to

generated files. There are several solutions to this:

- All developers should use the same versions, so that the rebuilt files are identical to files in CVS. (This starts to be difficult when each project you work on uses different versions.)
- Or people use a script to fix the timestamp after a checkout (the GCC folks have such a script).
- Or `configure.in` uses `AM_MAINTAINER_MODE`, which will disable all these rebuild rules by default. This is further discussed in maintainer-mode.

- Although we focused on spurious rebuilds, the converse can also happen. CVS's timestamp handling can also let you think an out-of-date file is up-to-date.

  For instance, suppose a developer has modified `Makefile.am` and rebuilt `Makefile.in`, and then decide to do a last-minute change to `Makefile.am` right before checking in both files (without rebuilding `Makefile.in` to account for the change).

  This last change to `Makefile.am` make the copy of `Makefile.in` out-of-date. Since CVS processes files alphabetically, when another developer `cvs update` his or her tree, `Makefile.in` will happen to be newer than `Makefile.am`. This other developer will not see `Makefile.in` is out-of-date.

Generated files out of CVS

One way to get CVS and `make` working peacefully is to never store generated files in CVS, i.e., do not CVS-control files which are `Makefile` targets (or *derived* files in Make terminology).

This way developers are not annoyed by changes to generated files. It does not matter if they all have different versions (assuming they are compatible, of course). And finally, timestamps are not lost, changes to sources files can't be missed as in the `Makefile.am`/`Makefile.in` example discussed earlier.

The drawback is that the CVS repository is not an exact copy of what is distributed and that users now need to install various development tools (maybe even specific versions) before they can build a checkout. But, after all, CVS's job is versioning, not distribution.

Allowing developers to use different versions of their tools can also hide bugs during distributed development. Indeed, developers will be using (hence testing) their own generated files, instead of the generated files that will be released actually. The developer who prepares the tarball might be using a version of the tool that produces bogus output (for instance a non-portable C file), something other developers could have noticed if they weren't using their own versions of this tool.

Third-party files

Another class of files not discussed here (because they do not cause timestamp issues) are files which are shipped with a package, but maintained elsewhere. For instance tools like `gettextize` and `autopoint` (from Gettext) or `libtoolize` (from Libtool), will install or update files in your package.

These files, whether they are kept under CVS or not, raise similar concerns about version mismatch between developers' tools. The Gettext manual has a section about this, see CVS Issues.

Node: maintainer-mode, Next: wildcards, Previous: CVS, Up: FAQ

`missing` and `AM_MAINTAINER_MODE`

`missing`

The `missing` script is a wrapper around several maintainer tools, designed to warn users if a maintainer tool is required but missing. Typical maintainer tools are `autoconf`, `automake`, `bison`, etc. Because file generated by these tools are shipped with the other sources of a package, these tools shouldn't be required during a user build and they are not checked for in `configure`.

However, if for some reason a rebuild rule is triggered and involves a missing tool, `missing` will notice it and warn the user. Besides the warning, when a tool is missing, `missing` will attempt to fix timestamps in a way which allow the build to continue. For instance `missing` will touch `configure` if `autoconf` is not installed. When all distributed files are kept under CVS, this feature of `missing` allows user *with no maintainer tools* to build a package off CVS, bypassing any timestamp inconsistency implied by `cvs update`.

If the required tool is installed, `missing` will run it and won't attempt to continue after failures. This is correct during development: developers love fixing failures. However, users with wrong versions of maintainer tools may get an error when the rebuild rule is spuriously triggered, halting the build. This failure to let the build continue is one of the arguments of the `AM_MAINTAINER_MODE` advocates.

`AM_MAINTAINER_MODE`

`AM_MAINTAINER_MODE` disables the so called "rebuild rules" by default. If you have `AM_MAINTAINER_MODE` in `configure.ac`, and run `./configure && make`, then `make` will *never* attempt to rebuilt `configure`, `Makefile.in`s, Lex or Yacc outputs, etc. I.e., this disables build rules for files which are usually distributed and that users should normally not have to update.

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

People use AM_MAINTAINER_MODE either because they do want their users (or themselves) annoyed by timestamps lossage (see CVS), or because they simply can't stand the rebuild rules and prefer running maintainer tools explicitly.

AM_MAINTAINER_MODE also allows you to disable some custom build rules conditionally. Some developers use this feature to disable rules that need exotic tools that users may not have available.

Several years ago François Pinard pointed out several arguments against AM_MAINTAINER_MODE. Most of them relate to insecurity. By removing dependencies you get non-dependable builds: change to sources files can have no effect on generated files and this can be very confusing when unnoticed. He adds that security shouldn't be reserved to maintainers (what `--enable-maintainer-mode` suggests), on the contrary. If one user has to modify a `Makefile.am`, then either `Makefile.in` should be updated or a warning should be output (this is what Automake uses `missing` for) but the last thing you want is that nothing happens and the user doesn't notice it (this is what happens when rebuild rules are disabled by AM_MAINTAINER_MODE).

Jim Meyering, the inventor of the AM_MAINTAINER_MODE macro was swayed by François's arguments, and got rid of AM_MAINTAINER_MODE in all of his packages.

Still many people continue to use AM_MAINTAINER_MODE, because it helps them working on projects where all files are kept under CVS, and because `missing` isn't enough if you have the wrong version of the tools.

Node: wildcards, Next: distcleancheck, Previous: maintainer-mode, Up: FAQ

Why doesn't Automake support wildcards?

Developers are lazy. They often would like to use wildcards in `Makefile.am`s, so they don't need to remember they have to update `Makefile.am`s every time they add, delete, or rename a file.

There are several objections to this:

- When using CVS (or similar) developers need to remember they have to run `cvs add` or `cvs rm` anyway. Updating `Makefile.am` accordingly quickly becomes a reflex.

  Conversely, if your application doesn't compile because you forgot to add a file in `Makefile.am`, it will help you remember to `cvs add` it.

- Using wildcards makes easy to distribute files by mistake. For instance some code a developer is experimenting with (a test case, say) but which should not be part of the distribution.

- Using wildcards it's easy to omit some files by mistake. For instance one developer creates a new file, uses it at many places, but forget to commit it. Another developer then checkout the incomplete project and is able to run `make dist' successfully, even though a file is missing.
- Listing files, you control *exactly* what you distribute. If some file that should be distributed is missing from your tree, `make dist` will complain. Besides, you don't distribute more than what you listed.
- Finally it's really hard to `forget` adding a file to `Makefile.am`, because if you don't add it, it doesn't get compiled nor installed, so you can't even test it.

Still, these are philosophical objections, and as such you may disagree, or find enough value in wildcards to dismiss all of them. Before you start writing a patch against Automake to teach it about wildcards, let's see the main technical issue: portability.

Although `$(wildcard ...)` works with GNU `make`, it is not portable to other `make` implementations.

The only way Automake could support `$(wildcard ...)` is by expending `$(wildcard ...)` when `automake` is run. Resulting `Makefile.in`s would be portable since they would list all files and not use `$(wildcard ...)`. However that means developers need to remember they must run `automake` each time they add, delete, or rename files.

Compared to editing `Makefile.am`, this is really little win. Sure, it's easier and faster to type `automake; make` than to type `emacs Makefile.am; make`. But nobody bothered enough to write a patch add support for this syntax. Some people use scripts to generated file lists in `Makefile.am` or in separate `Makefile` fragments.

Even if you don't care about portability, and are tempted to use `$(wildcard ...)` anyway because you target only GNU Make, you should know there are many places where Automake need to know exactly which files should be processed. As Automake doesn't know how to expand `$(wildcard ...)`, you cannot use it in these places. `$(wildcard ...)` is a black box comparable to `AC_SUBST`ed variables as far Automake is concerned.

You can get warnings about `$(wildcard ...)` constructs using the `-Wportability` flag.

Node: distcleancheck, Next: renamed objects, Previous: wildcards, Up: FAQ

Files left in build directory after distclean

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

As explained in Dist, `make distcheck` attempts to build and check your package for errors like this one.

`make distcheck` will perform a `VPATH` build of your package, and then call `make distclean`. Files left in the build directory after `make distclean` has run are listed after this error.

This diagnostic really covers two kinds of errors:

- files that are forgotten by distclean;
- distributed files that are erroneously rebuilt.

The former left-over files are not distributed, so the fix is to mark them for cleaning (see Clean), this is obvious and doesn't deserve more explanations.

The latter bug is not always easy to understand and fix, so let's proceed with an example. Suppose our package contains a program for which we want to build a man page using `help2man`. GNU `help2man` produces simple manual pages from the `--help` and `--version` output of other commands (see Overview). Because we don't to force want our users to install `help2man`, we decide to distribute the generated man page using the following setup.

```
# This Makefile.am is bogus.
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo$(EXEEXT)
   help2man --output=foo.1 ./foo$(EXEEXT)
```

This will effectively distribute the man page. However, `make distcheck` will fail with:

```
ERROR: files left in build directory after distclean:
./foo.1
```

Why was `foo.1` rebuilt? Because although distributed, `foo.1` depends on a non-distributed built file: `foo$(EXEEXT)`. `foo$(EXEEXT)` is built by the user, so it will always appear to be newer than the distributed `foo.1`.

`make distcheck` caught an inconsistency in our package. Our intent was to distribute `foo.1` so users do not need installing `help2man`, however since this our rule causes this file to be always rebuilt, users *do* need `help2man`. Either we should ensure that `foo.1` is not rebuilt by users, or there is no

point in distributing `foo.1`.

More generally, the rule is that distributed files should never depend on non-distributed built files. If you distribute something generated, distribute its sources.

One way to fix the above example, while still distributing `foo.1` is to not depend on `foo$(EXEEXT)`. For instance, assuming `foo --version` and `foo --help` do not change unless `foo.c` or `configure.ac` change, we could write the following `Makefile.am`:

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
dist_man_MANS = foo.1

foo.1: foo.c $(top_srcdir)/configure.ac
        $(MAKE) $(AM_MAKEFLAGS) foo$(EXEEXT)
   help2man --output=foo.1 ./foo$(EXEEXT)
```

This way, `foo.1` will not get rebuilt every time `foo$(EXEEXT)` changes. The `make` call makes sure `foo$(EXEEXT)` is up-to-date before `help2man`. Another way to ensure this would be to use separate directories for binaries and man pages, and set `SUBDIRS` so that binaries are built before man pages.

We could also decide not to distribute `foo.1`. In this case it's fine to have `foo.1` dependent upon `foo$(EXEEXT)`, since both will have to be rebuilt. However it would be impossible to build the package in a cross-compilation, because building `foo.1` involves an *execution* of `foo$(EXEEXT)`.

Another context where such errors are common is when distributed files are built by tools which are built by the package. The pattern is similar:

```
distributed-file: built-tools distributed-sources
        build-command
```

should be changed to

```
distributed-file: distributed-sources
        $(MAKE) $(AM_MAKEFLAGS) built-tools
        build-command
```

or you could choose not to distribute `distributed-file`, if cross-compilation does not matter.

The points made through these examples are worth a summary:

- Distributed files should never depend upon non-distributed built files.
- Distributed files should be distributed will all their dependencies.
- If a file is *intended* be rebuilt by users, there is no point in distributing it.

For desperate cases, it's always possible to disable this check by setting `distcleancheck_listfiles` as documented in Dist. Make sure you do understand the reason why `make distcheck` complains before you do this. `distcleancheck_listfiles` is a way to *hide* errors, not to fix them. You can always do better.

Node: renamed objects, Previous: distcleancheck, Up: FAQ

Why are object files sometimes renamed?

This happens when per-target compilation flags are used. Object files need to be renamed just in case they would clash with object files compiled from the same sources, but with different flags. Consider the following example.

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

Obviously the two programs are built from the same source, but it would be bad if they shared the same object, because `generic.o` cannot be built with both `-DEXIT_CODE=0` *and* `-DEXIT_CODE=1`. Therefore `automake` outputs rules to build two different objects: `true-generic.o` and `false-generic.o`.

`automake` doesn't actually look whether sources files are shared to decide if it must rename objects. It will just rename all objects of a target as soon as it sees per-target compilation flags are used.

It's OK to share object files when per-target compilation flags are not used. For instance `true` and `false` will both use `version.o` in the following example.

```
AM_CPPFLAGS = -DVERSION=1.0
bin_PROGRAMS = true false
```

```
true_SOURCES = true.c version.c
false_SOURCES = false.c version.c
```

Note that the renaming of objects is also affected by the `_SHORTNAME` variable (see Program and Library Variables).

Node: Macro and Variable Index, Next: General Index, Previous: FAQ, Up: Top

Macro and Variable Index

Node: General Index, Previous: Macro and Variable Index, Up: Top

General Index

Table of Contents

Footnotes

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

2. 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.

3. Don't try seeking a solution where `opt/Makefile` is created conditionally, this is a lot trickier than the solutions presented here.

4. We believe. This work is new and there are probably warts. See Introduction, for information on reporting bugs.

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

6. Much, if not most, of the information in the following sections pertaining to preprocessing Fortran 77 programs was taken almost verbatim from Catalogue of Rules.

7. For example, the cfortran package addresses all of these inter-language issues, and runs under nearly all Fortran 77, C and C++ compilers on nearly all platforms. However, `cfortran` is not yet Free Software, but it will be in the next major release.

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

9. 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.
