# Cygnus configure

**K. Richard Pixley**
**Cygnus Support**

Edited January, 1993, by Jeffrey Osier, Cygnus Support.

# 1 What `configure` does

This manual documents Cygnus `configure`, a program which helps to automate much of the setup activity associated with building large suites of programs, such the Cygnus Support Developer's Kit. This manual is therefore geared toward readers who are likely to face the problem of configuring software in source form before compiling and installing it. We assume you are an experienced programmer or system administrator. For further background on this topic, see *On Configuring Development Tools* by K. Richard Pixley.

When `configure` runs, it does the following things:

- *creates build directories*

  When you run `configure` with the '`--srcdir`' option, it uses the current directory as the *build directory*, creating under it a directory tree that parallels the directory structure of the source directory. If you don't specify a '`srcdir`', `configure` first assumes that the source code you wish to configure is in your current directory; if it finds no '`configure.in`' input file there, it searches in the directory `configure` itself lies in. (For details, see Section 3.1.2 [Build directories], page 10.)

- *generates* '`Makefile`'

  A '`Makefile`' template from the source directory, usually called '`Makefile.in`', is copied to an output file in the build directory which is most often named '`Makefile`'. `configure` places definitions for a number of standard '`Makefile`' macros at the beginning of the output file. If '`--prefix=`*dir*' or '`--exec_prefix=`*dir*' are specified on the `configure` command line, corresponding '`Makefile`' variables are set accordingly. If host, target, or site-specific '`Makefile`' fragments exist, these are inserted into the output file. (For details, see Section 3.1.3 [`Makefile` generation], page 11.)

- *generates* '`.gdbinit`'

  If the source directory contains a '`.gdbinit`' file and the build directory is not the same as the source directory, a '`.gdbinit`' file is created in the build directory. This '`.gdbinit`' file contains commands which allow the source directory to be read when debugging with the GNU debugger, `gdb`. (See section "Command Files" in *Debugging With GDB*.)

- *makes symbolic links*

  Most build directories require that some symbolic links with generic names are built pointing to specific files in the source directory. If the system where `configure` runs cannot support symbolic links, hard links are used instead. (For details, see Section 3.2 [The `configure.in` input file], page 13.)

- *generates* ‘`config.status`’

  `configure` creates a shell script named ‘`config.status`’ in the build directory. This shell script, when run from the build directory (usually from within a ‘`Makefile`’), will reconfigure the build directory (but not its subdirectories). This is most often used to have a ‘`Makefile`’ update itself automatically if a new source directory is available.

- *calls itself recursively*

  If the source directory has subdirectories that should also be configured, `configure` is called for each.

# 2 Invoking configure

Cygnus `configure` is a shell script which resides in a source tree. The usual way to invoke `configure` is from the shell, as follows:

```
eg$ ./configure hosttype
```

This prepares the source in the current directory (`.`) to be compiled for a *hosttype* environment. It assumes that you wish to build programs and files in the default *build directory* (also the current directory, '`.`'). If you do not specify a value for *hosttype*, Cygnus `configure` will attempt to discover this information by itself (see Section 3.1.4 [Determining system information], page 12). For information on *hosttype* environments, See Section 3.4 [Host], page 22.

All GNU software is packaged with one or more `configure` script(s) (see section "How Configuration Should Work" in *GNU Coding Standards*). By using `configure` you prepare the source for your specific environment by selecting and using '`Makefile`' fragments and fragments of shell scripts, which are prepared in advance and stored with the source.

`configure`'s command-line options also allow you to specify other aspects of the source configuration:

```
configure hosttype  [--target=target] [--srcdir=dir] [--rm]
          [--site=site] [--prefix=dir] [--exec-prefix=dir]
          [--program-prefix=string] [--tmpdir=dir]
          [--with-package[=yes/no]] [--without-package]
          [--enable-feature[=yes/no]] [--disable-feature]
          [--norecursion] [--nfp] [-s] [-v] [-V | --version] [--help]
```

`--target=`*target*

Requests that the sources be configured to target the *target* machine. If no target is specified explicitly, the target is assumed to be the same as the host (i.e., a *native* configuration). See Section 3.4 [Host], page 22, and Section 3.5 [Target], page 22, for discussions of each.

`--srcdir=`*dir*

Direct each generated '`Makefile`' to use the sources located in directory *dir*. Use this option whenever you wish the object code to reside in a different place from the source code. The *build directory* is always assumed to be the directory you call `configure` from. See Section 3.1.2 [Build directories], page 10, for an example. If the source directory is not specified, `configure` assumes that the source is in your current directory. If `configure` finds no '`configure.in`' there, it searches in the same directory that the `configure` script itself lies in. Pathnames specified (Values for *dir*) can be either absolute relative to the *build* directory.

`--rm`
: *Remove* the configuration specified by *hosttype* and the other command-line options, rather than create it.

> *Note:* We recommend that you use '`make distclean`' rather than use this option; see section "Invoking `make`" in *GNU Make*, for details on '`make distclean`'.

`--site=`*site*
: Generate the '`Makefile`' using site-specific '`Makefile`' fragments for *site*. See Section 3.6 [Adding information about local conventions], page 22.

`--prefix=`*dir*
: Configure the source to install programs and files under directory *dir*.

This option sets the variable '`prefix`'. Each generated '`Makefile`' will have its '`prefix`' variables set to this value. (See Section 3.1 [What `configure` really does], page 7.)

`--exec-prefix=`*dir*
: Configure the source to install *host dependent* files in *dir*.

This option sets the variable '`exec_prefix`'. Each generated '`Makefile`' will have its '`exec_prefix`' variables set to this value. (See Section 3.1 [What `configure` really does], page 7.)

`--program-prefix=`*string*
: Configure the source to install certain programs using *string* as a prefix. This applies to programs which might be used for cross-compilation, such as the compiler and the binary utilities, and also to programs which have the same names as common Unix programs, such as `make`.

This option sets the variable '`program_prefix`'. Each generated '`Makefile`' will have its '`program_prefix`' variables set to this value. (See Section 3.1 [What `configure` really does], page 7.)

`--tmpdir=`*tmpdir*
: Use the directory *tmpdir* for `configure`'s temporary files. The default is the value of the environment variable `TMPDIR`, or '`/tmp`' if the environment variable is not set.

`--with-`*package*`[=`*yes/no*`]`
`--without-`*package*
: Indicate that *package* is present, or not present, depending on *yes/no*. If *yes/no* is nonexistent, its value is assumed to be `yes`. '`--without-`*package*' is equivalent to '`--with-`*package*`=no`'.

For example, if you wish to configure the program `gcc` for a Sun SPARCstation running SunOS 4.x, and you want `gcc` to use the GNU linker `ld`, you can configure `gcc` using

```
eg$ configure --with-gnu-ld sun4
```

See Section 3.1 [What `configure` really does], page 7, for details. See the installation or release notes for your particular package for details on which other *package* options are recognized.

`--enable-`*feature*`[=`*yes/no*`]`
`--disable-`*feature*

Include *feature*, or not, depending on *yes/no*. If *yes/no* is nonexistent, its value is assumed to be `yes`. '`--disable-`*feature*' is equivalent to '`--enable-`*feature*`=no`'.

See Section 3.1 [What `configure` really does], page 7, for details. See the installation or release notes for your particular package for details on which other *feature* options are recognized.

`--norecursion`

Configure only this directory; ignore any subdirectories. This is used by the executable shell script '`config.status`' to reconfigure only the current directory; it is most often used non-interactively, when `make` is invoked. (See Section 3.1.5 [`config.status`], page 12.)

`--nfp` Assume that the intended *hosttype* has no floating point unit.

`-s` Suppress status output. This option is used internally by `configure` when calling itself recursively in subdirectories. You can override this option with the `--verbose` option.

`-v`
`--verbose`

Print status lines for each directory configured. Normally, only the status lines for the initial working directory are printed.

`--version`
`-V` Print the `configure` version number.

`--help` Print a short summary of how to invoke `configure`.

*Note:* You may introduce options with a single dash, '`-`', rather than two dashes, '`--`'. However, you may not be able to truncate long option names when using a single dash. When using two dashes, options may be abbreviated as long as each option can be uniquely identified. For example,

```
eg$ configure --s=/u/me/src hosttype
```

is ambiguous, as '`--s`' could refer to either '`--site`' or '`--srcdir`'. However,

```
eg$ configure --src=/u/me/src hosttype
```

is a valid abbreviation.

# 3 Using configure

configure prepares source directories for building programs in them. "Configuring" is the process of preparing software to compile correctly on a given *host*, for a given *target*.

configure subsequently writes a configured 'Makefile' from a pre-built template; configure uses variables that have been set in the configuring process to determine the values of some variables in the 'Makefile'. Because of this we will refer to both configure variables and 'Makefile' variables. This convention allows us to determine where the variable should be set initially, in either 'configure.in' or 'Makefile.in'.

## 3.1 What configure really does

Cygnus configure is a shell script that sets up an environment in which your programs will compile correctly for your machine and operating system, and will install in proper places. configure accomplishes this task by doing the following:

- it generates a 'Makefile' from a custom template called 'Makefile.in' in each relevant source directory;
- it customizes the build process to your specifications; you set certain variables for configure, either on the command line or in the file 'configure.in', which subsequently sets variables in each generated 'Makefile' to be used by make when actually building the software;
- it creates *build directories*, places for your code to be compiled in before being installed;
- it generates a '.gdbinit' in the build directory, if needed, to communicate to gdb where to find the program's source code;
- it generates a shell script called 'config.status' which is used most often by the 'Makefile' to reconfigure itself;
- it recurses in subdirectories, setting up entire trees so that they build correctly; if configure finds another configure script further down in a given source tree, it knows to use this script and not recur.

For the sake of safety (i.e., in order to prevent broken installations), the GNU coding standards call for software to be *configured* in such a way that an end user trying to build a given package will be able to do so by affecting a finite number of variables. All GNU software comes with an executable configure shell script which sets up an environment within a build directory which will correctly compile your new package for your host (or, alternatively, whatever host you specify to configure). For further background on this topic, see *On Configuring Development Tools* by K. Richard Pixley.

Use configure to set for the build process:

- correct values for certain variables;
- which type of host you wish to configure a given package for (see Section 3.4 [Host], page 22);
- where you want to install this package (by using 'prefix', 'exec-prefix' and 'program-prefix'; see Section 3.3.3 [Full descriptions of all installation directories], page 20);
- optionally, which type of machine you wish to *target* this package's output to (see Section 3.5 [Target], page 22);
- which other GNU packages are already installed and available to this particular build (by using the '--with-*package*' option; see Chapter 2 [Invoking configure], page 3);
- where to place temporary files (by using the '--tmpdir=*dir*' option; see Chapter 2 [Invoking configure], page 3);
- whether to recur in subdirectories (changeable through the '--norecursion' option; see Chapter 2 [Invoking configure], page 3).

configure uses a few other files to complete its tasks. These are discussed in detail where noted.

configure.in
: Input file for configure. Shell script fragments reside here. See Section 3.2 [The configure.in input file], page 13.

Makefile.in
: Template which configure uses to build a file called 'Makefile' in the *build directory*. See Section 3.1.3 [Makefile generation], page 11.

config.sub
: Shell script used by configure to expand referents to the *hosttype* argument into a single specification of the form *cpu-vendor-os*. For instance, on the command line you can specify

```
eg$ ./configure sun4
```

to configure for a Sun SPARCstation running SunOS 4.x. configure consults config.sub to find that the three-part specification for this is

```
sparc-sun-sunos4.1.1
```

which notes the *cpu* as 'sparc', the *manufacturer* as 'sun' (Sun Microsystems), and the *os* (operating system) as 'sunos4.1.1', the SunOS 4.1.1 release. See Section 3.2.1 [Variables available to configure], page 13.

`config.guess`
: If you do not put the *hosttype* argument on the command line, `configure` uses the `config.guess` shell script to make an analysis of your machine (it assumes that you wish to configure your software for the type of machine on which you are running). The output of `config.guess` is a three-part identifier as described above.

`config.status`
: The final step in configuring a directory is to create a shell script, `config.status`. The main purpose of this file is to allow the '`Makefile`' for the current directory to rebuild itself, if necessary. See Section 3.1.5 [`config.status`], page 12.

`config/*` `configure` uses three types of '`Makefile`' *fragments*, which reside in the directory '*srcdir*`/config/`'. See Section 3.6 [Adding information about local conventions], page 22.

### 3.1.1 Build variables

There are several variables in the build process which you can control through build programs such as `make`. These include machine definitions, local conventions, installation locations, locations for temporary files, etc. This data is accessible through certain variables which are configurable in the build process; we refer to them as *build variables*.

For lists of build variables which you can affect by using `configure`, see Section 3.2.1 [Variables available to `configure.in`], page 13, and Section 3.3.3 [Full descriptions of all installation directories], page 20.

Generally, build variables, which are used by the '`Makefile`' to determine various aspects of the build and installation processes, are changeable with command-line options to `configure`. In most large suites of programs, like the Cygnus Support Developer's Kit, the individual programs reside in several subdirectories of a single source code "tree". All of these subdirectories need to be configured with information relative to the *build directory*, which is not known until `configure` is run. Unless specified otherwise, `configure` recursively configures every subdirectory in the source tree.

Build variables are passed from `configure` directly into the '`Makefile`', and use the same names (except that dashes are transformed into underbars; for example, when you specify the option '`--exec-prefix`' on the command line, the '`Makefile`' variable '`exec_prefix`' is set). In other words, if you specify

```
eg$ ./configure --prefix=/usr/gnu/local ... hosttype
```

on the command line, `configure` sets an variable called '`prefix`' to '`/usr/gnu/local`', and passes this into the '`Makefile`' in the same manner. After this command, each '`Makefile`' generated by `configure` will contain a line that reads:

```
prefix = /usr/gnu/local
```

For a list of the '`Makefile`' variables `configure` can change, and instructions on how to change them, see Section 3.2.1 [Variables available to `configure.in`], page 13, and Chapter 2 [Invoking `configure`], page 3.

### 3.1.2 Build directories

By default, `configure` builds a '`Makefile`' and symbolic links in the same directory as the source files. This default works for many cases, but it has limitations. For instance, using this approach, you can only build object code for one host at a time.

We refer to each directory where `configure` builds a '`Makefile`' as a *build directory*.

The build directory for any given build is always the directory from which you call `configure`, or '`.`' relative to your prompt. The default *source directory*, the place `configure` looks to find source code, is also '`.`'. For instance, if we have a directory '`/gnu-stuff/src/`' that is the top branch of a tree of GNU source code we wish to configure, then the program we will use to configure this code is '`/gnu-stuff/src/configure`', as follows. (Assume for the sake of argument that our machine is a sun4.)

```
eg$ cd /gnu-stuff/src
eg$ ./configure sun4
Created "Makefile" in /gnu-stuff/src
eg$
```

We just configured the code in '`/gnu-stuff/src`' to run on a Sun SPARCstation using SunOS 4.x by creating a '`Makefile`' in '`/gnu-stuff/src`'. By default, we also specified that when this code is built, the object code should reside in the same directory, '`/gnu-stuff/src`'.

However, if we wanted to build this code for more than one host, we would be in trouble, because the new configuration would write over the old one, destroying it in the process. What we can do is to make a new *build directory* and configure from there. Running `configure` from the new directory will place a correct 'Makefile' and a 'config.status' in this new file. That is all configure does; we must run make to generate any object code.

The new 'Makefile' in '/gnu-stuff/sun4-obj', created from the template file '/gnu-stuff/src/Makefile.in' contains all the information needed to build the program.

```
eg$ mkdir /gnu-stuff/sun4-obj
eg$ cd /gnu-stuff/sun4-obj
eg$ ../src/configure --srcdir=../src sun4
Created "Makefile" in /gnu-stuff/sun4-obj
eg$ ls
Makefile       config.status
eg$ make all info install install-info clean
compilation messages...
eg$ mkdir /gnu-stuff/solaris2
eg$ cd /gnu-stuff/solaris2
eg$ ../src/configure --srcdir=../src sol2
Created "Makefile" in /gnu-stuff/solaris2
eg$ ls
Makefile       config.status
eg$ make all info install install-info clean
compilation messages...
```

We can repeat this for other configurations of the same software simply by making a new build directory and reconfiguring from inside it. If you do not specify the *hosttype* argument, configure will attempt to figure out what kind of machine and operating system you happen to be using. See Section 3.1.4 [Determining system information], page 12. Of course, this may not always be the configuration you wish to build.

*Caution:* If you build more than one configuration for a single program, remember that you must also specify a different '--prefix' for each configuration at configure-time. Otherwise, both configurations will be installed in the same default location ('/usr/local'); the configuration to be installed last would overwrite previously installed configurations.

### 3.1.3 Makefile generation

Cygnus configure creates a file called 'Makefile' in the build directory which can be used with make to automatically build a given program or package. configure also builds a 'Makefile' for each relevant subdirectory for a given program or package (irrelevant subdirectories would be those which contain no code which needs configuring, and which therefore have no `configure` input file '`configure.in`' and no '`Makefile`' template '`Makefile.in`'). See section "How to Run `make`" in *GNU Make*, for details on using `make` to compile your source code.

Each '`Makefile`' contains variables which have been configured for a specific build. These build variables are determined when `configure` is run. All build variables have defaults. By default, `configure` generates a '`Makefile`' which specifies:

- a *native* build, which is to occur
- in the current directory, and which will be installed
- in the default installation directory ('`/usr/local`') when the code is compiled with `make`.

Variables are changeable through command-line options to `configure` (see Chapter 2 [Invoking `configure`], page 3).

If you are porting a new program and intend to use `configure`, see Chapter 4 [Porting with `configure`], page 25, as well as section "Writing Makefiles" in *GNU Make*, and section "Makefile Conventions" in *GNU Coding Standards*.

### 3.1.4 Determining system information

The shell script `config.guess` is called when you do not specify a *hosttype* on the command line to `configure`. `config.guess` acquires available system information from your local machine through the shell command `uname`. It compares this information to a database and attempts to determine a usable three-part system identifier (known as a *triple*) to use as your *hosttype*. See Section 3.1 [What `configure` really does], page 7, to see how this information is used.

*Note:* If you do not specify a *hosttype* on the command line, `configure` will attempt to configure your software to run on the machine you happen to be using. This may not be the configuration you desire.

### 3.1.5 `config.status`

The final step in configuring a directory is to create an executable shell script, '`config.status`'. The main purpose of this file is to allow the '`Makefile`' for the current directory to rebuild itself, if necessary. It is usually run from within the 'Makefile'. See Section 3.7 [Extensions to the GNU coding standards], page 23.

'config.status' also contains a record of the configure session which created it.

## 3.2 The configure.in input file

A 'configure.in' file for Cygnus configure consists of a *per-invocation* section, followed by a *per-host* section, followed by a *per-target* section, optionally followed by a *post-target* section. Each section is a shell script fragment, which is executed by the configure shell script at an appropriate time. Values are passed among configure and the shell fragments through a set of shell variables. When each section is being interpreted by the shell, the shell's current directory is the build directory, and any files created by the section (or referred to by the section) will be relative to the build directory. To reference files in other places (such as the source directory), prepend a shell variable such as '$(srcdir)/' to the desired file name.

The beginning of the 'configure.in' file begins the *per-invocation* section.

A line beginning with '# per-host:' begins the *per-host* section.

A line beginning with '# per-target:' begins the *per-target* section.

If it exists, the *post-target* section begins with '# post-target:'.

### 3.2.1 Variables available to configure.in

The following variables pass information between the standard parts of configure and the shell-script fragments in 'configure.in':

srctrigger
: Contains the name of a source file that is expected to live in the source directory. You must usually set this in the *per-invocation* section of 'configure.in'. configure tests to see that this file exists. If the file does not exist, configure prints an error message. This is used as a sanity check that 'configure.in' matches the source directory.

srcname
: Contains the name of the source collection contained in the source directory. You must usually set this in the *per-invocation* section of 'configure.in'. If the file named in

'srctrigger' does not exist, configure uses the value of 'srcname' when it prints the error message.

configdirs
: Contains the names of any subdirectories in which configure should recurse. You must usually set this in the *per-invocation* section of 'configure.in'. If 'Makefile.in' contains a line starting with 'SUBDIRS =', then it will be replaced with an assignment to 'SUBDIRS' using the value of 'configdirs' (if 'subdirs' is empty). This can be used to determine which directories to configure and build depending on the host and target configurations. Use 'configdirs' (instead of the 'subdirs' variable described below) if you want to be able to partition the subdirectories, or use independent 'Makefile' fragments. Each subdirectory can be independent, and independently reconfigured.

subdirs
: Contains the names of any subdirectories where configure should create a 'Makefile' (in addition to the current directory), *without* recursively running configure. Use 'subdirs' (instead of the 'configdirs' variable described above) if you want to configure all of the directories as a unit. Since there is a single invocation of configure that configures many directories, all the directories can use the same 'Makefile' fragments, and the same configure.in.

host
: Contains the full configuration name for the host (generated by the script 'config.sub' from the name that you entered). This is a three-part name (commonly referred to as a *triple*) of the form *cpu-vendor-os*.

  There are separate variables 'host_cpu', 'host_vendor', and 'host_os' that you can use to test each of the three parts; this variable is useful, however, for error messages, and for testing combinations of the three components.

host_cpu
: Contains the first element of the canonical triple representing the host as returned by 'config.sub'. This is occasionally used to distinguish between minor variations of a particular vendor's operating system and sometimes to determine variations in binary format between the host and the target.

host_vendor
: Contains the second element of the canonical triple representing the host as returned by 'config.sub'. This is usually used to distinguish among the numerous variations of *common* operating systems.

host_os
: Contains the the third element of the canonical triple representing the host as returned by 'config.sub'.

target
: Contains the full configuration name (generated by the script 'config.sub' from the name that you entered) for the target. Like the host, this is a three-part name of the form *cpu-vendor-os*.

  There are separate variables 'target_cpu', 'target_vendor', and 'target_os' that you can use to test each of the three parts; this variable is useful, however, for error messages, and for testing combinations of the three components.

`target_cpu`
: Contains the first element of the canonical triple representing the target as returned by 'config.sub'. This variable is used heavily by programs which are involved in building other programs, like the compiler, assembler, linker, etc. Most programs will not need the 'target' variables at all, but this one could conceivably be used to build a program, for instance, that operated on binary data files whose byte order or alignment differ from the system where the program is running.

`target_vendor`
: Contains the second element of the canonical triple representing the target as returned by 'config.sub'. This is usually used to distinguish among the numerous variations of *common* operating systems or object file formats. It is sometimes used to switch between different flavors of user interfaces.

`target_os`
: Contains the the third element of the canonical triple representing the target as returned by 'config.sub'. This variable is used by development tools to distinguish between subtle variations in object file formats that some vendors use across operating system releases. It might also be use to decide which libraries to build or what user interface the tool should provide.

`floating_point`
: Set to 'no' if you invoked `configure` with the '--nfp' command-line option, otherwise it is empty. This is a request to target machines with *no floating point* unit, even if the targets ordinarily have floating point units available.

`gas`
: Set to 'true' if you invoked `configure` with the '--with-gnu-as' command line option, otherwise it is empty. This is a request to assume that the specified *hosttype* machine has GNU `as` available even if it ordinarily does not.

`srcdir`
: Set to the name of the directory containing the source for this program. This will be different from '.' if you have specified the '--srcdir=*dir*' option. 'srcdir' can indicate either an absolute path or a path relative to the build directory.

`package_makefile_frag`
: If set in 'configure.in', this variable should be the name a file relative to 'srcdir' to be included in the resulting 'Makefile'. If the named file does not exist, `configure` will print a warning message. This variable is not set by `configure`.

`host_makefile_frag`
: If set in 'configure.in', this variable should be the name a file relative to 'srcdir' to be included in the resulting 'Makefile'. If the named file does not exist, `configure` will print a warning message. This variable is not set by `configure`.

`target_makefile_frag`
: If set in '`configure.in`', this variable should be the name of a file, relative to '`srcdir`', to be included in the resulting '`Makefile`'. If the named file does not exist, `configure` will print a warning message. This variable is not set by `configure`.

`site_makefile_frag`
: Set to a file name representing to the default '`Makefile`' fragment for this host. It may be set in '`configure.in`' to override this default. Normally '`site_makefile_frag`' is empty, but will have a value if you specify '`--site=`*site*' on the command line.

`Makefile`
: Set to the name of the generated '`Makefile`'. Normally this value is precisely '`Makefile`', but some programs may want something else.

`removing`
: Normally empty but will be set to some non-null value if you specified '`--rm`' on the command line. That is, if '`removing`' is not empty, then `configure` is *removing* a configuration rather than creating one.

`files`
: If this variable is not empty following the *per-target* section, then each word in its value will be the target of a symbolic link named in the corresponding word from the '`links`' variable.

`links`
: If the '`files`' variable is not empty following the *per-target* section, then `configure` creates symbolic links with the first word of '`links`' pointing to the first word of '`files`', the second word of '`links`' pointing to the second word of '`files`', and so on.

### 3.2.2 A minimal `configure.in`

A minimal '`configure.in`' consists of four lines.

```
srctrigger=foo.c
srcname="source for the foo program"
# per-host:
# per-target:
```

The '`# per-host:`' and '`# per-target:`' lines divide the file into the three required sections. The '`srctrigger`' line names a file. `configure` checks to see that this file exists in the source directory before configuring. If the '`srctrigger`' file does not exist, `configure` uses the value of '`srcname`' to print an error message about not finding the source.

This particular example uses no links, and only the default host, target, and site-specific '`Makefile`' fragments if they exist.

### 3.2.3 For each invocation

configure invokes the entire shell script fragment from the start of 'configure.in' up to a line beginning with '# per-host:' immediately after parsing command line arguments. The variables 'srctrigger' and 'srcname' *must* be set here.

You might also want to set the variables 'configdirs' and 'package_makefile_frag' here.

### 3.2.4 Host-specific instructions

The *per-host* section of 'configure.in' starts with the line that begins with '# per-host:' and ends before a line beginning with '# per-target:'. configure invokes the commands in the *per-host* section when determining host-specific information.

This section usually contains a big case statement using the variable 'host' to determine appropriate values for 'host_makefile_frag' and 'files', although 'files' is not usually set here. Usually, it is set at the end of the *per-target* section after determining the names of the target specific configuration files.

### 3.2.5 Target-specific instructions

The *per-target* section of 'configure.in' starts with the line that begins with '# per-target:' and ends before the line that begins with '# post-target:', if there is such a line. Otherwise the *per-target* section extends to the end of the file. configure invokes the commands in the *per-target* section when determining target-specific information, and before building any files, directories, or links.

This section usually contains a big case statement using the variable 'target' to determine appropriate values for 'target_makefile_frag' and 'files'. The last lines in the *per-target* section normally set the variables 'files' and 'links'.

### 3.2.6 Instructions to be executed after target info

The *post-target* section is optional. If it exists, the 'post-target' section starts with a line beginning with '# Post-target:' and extends to the end of the file. If it exists, configure invokes this section once for each target after building all files, directories, or links.

This section is seldom needed, but you can use it to edit the 'Makefile' generated by configure.

### 3.2.7 An example configure.in

Here is a small example of a 'configure.in' file.

```
# This file is a collection of shell script fragments
# used to tailor a template configure script as
# appropriate for this directory.  For more information,
# see configure.texi.

configdirs=
srctrigger=warshall.c
srcname="bison"

# per-host:
case "${host}" in
m88k-motorola-*)
        host_makefile_frag=config/mh-delta88
        ;;
esac

# per-target:
files="bison_in.hairy"
links="bison.hairy"

# post-target:
```

## 3.3 Install locations

Using the default configuration, 'make install' creates a single tree of files, some of which are programs. The location of this tree is determined by the value of the variable 'prefix'. The default value of 'prefix' is '/usr/local'. This is often correct for native tools installed on only one host.

### 3.3.1 Changing the default install directory

In the default configuration, all files are installed in subdirectories of '/usr/local'. The location is determined by the value of the configure variable 'prefix'; in turn, this determines the value of the 'Makefile' variable of the same name ('prefix').

You can also set the value of the 'Makefile' variable 'prefix' explicitly each time you invoke make if you are so inclined. However, because many programs have this location compiled in, you must specify the 'prefix' value consistently on each invocation of make, or you will end up with a broken installation.

To make this easier, the value of the configure variable 'prefix' can be set on the command line to configure using the option '--prefix='.

### 3.3.2 Installing for multiple hosts

By default, host dependent files are installed in subdirectories of '$(exec_prefix)'. The location is determined by the value of the configure variable 'exec_prefix', which determines the value of the 'Makefile' variable 'exec_prefix'. This makes it easier to install for a single host, and simplifies changing the default location for the install tree. The default doesn't allow for multiple hosts to effectively share host independent files, however.

To configure so that multiple hosts can share common files, use something like:

```
configure host1 -prefix=/usr/gnu -exec_prefix=/usr/gnu/H-host1
make all info install install-info clean

configure host2 -prefix=/usr/gnu -exec_prefix=/usr/gnu/H-host2
make all info install install-info
```

The first line configures the source for *host1* to place host-specific programs in subdirectories of '/usr/gnu/H-*host1*'.

The second line builds and installs all programs for *host1*, including both host-independent and host-specific files, as well as removing the host-specific object files from of the build directory.

The third line reconfigures the source for *host2* to place host specific programs in subdirectories of '/usr/gnu/H-*host2*'.

The fourth line builds and installs all programs for *host2*. Host specific files are installed in new directories, but the host independent files are installed *on top of* the host independent files installed for *host1*. This results in a single copy of the host independent files, suitable for use by both hosts.

See Section 3.7 [Extensions to the GNU coding standards], page 23, for more information.

### 3.3.3 Full descriptions of all installation subdirectories

During any install, a number of standard directories are created. Their names are determined by '`Makefile`' variables. Some of the defaults for '`Makefile`' variables can be changed at configuration time using command line options to `configure`. For more information on the standard directories or the '`Makefile`' variables, please refer to section "Makefile Conventions" in *GNU Coding Standards*. See also Section 3.7 [Extensions to the GNU coding standards], page 23.

Note that `configure` does not create the directory indicated by the variable '`srcdir`' at any time. `$(srcdir)` is not an installation directory.

You can override all '`Makefile`' variables on the command line to `make`. (See section "Overriding Variables" in *GNU Make*.) If you do so, you will need to specify the value precisely the same way for each invocation of `make`, or you risk ending up with a broken installation. This is because many programs have the locations of other programs or files compiled into them. If you find yourself overriding any of the variables frequently, you should consider site dependent '`Makefile`' fragments. See also Section 4.3 [Adding site info], page 27.

During '`make install`', a number of standard directories are created and populated. The following '`Makefile`' variables define them. Those whose defaults are set by corresponding `configure` variables are marked "`Makefile` and `configure`".

`prefix (Makefile and configure)`
: The root of the installation tree. You can set its '`Makefile`' default with the '`--prefix=`' command line option to `configure` (see Chapter 2 [Invoking `configure`], page 3). The default value for '`prefix`' is '`/usr/local`'.

`bindir`
: A directory for binary programs that users can run. The default value for '`bindir`' depends on '`prefix`'; '`bindir`' is normally changed only indirectly through '`prefix`'. The default value for '`bindir`' is '`$(prefix)/bin`'.

exec_prefix (Makefile and configure)
: A directory for host dependent files. You can specify the 'Makefile' default value by using the '--exec_prefix=' option to configure. (See Chapter 2 [Invoking configure], page 3.) The default value for 'exec_prefix' is '$(prefix)'.

libdir
: A directory for libraries and support programs. The default value for 'libdir' depends on 'prefix'; 'libdir' is normally changed only indirectly through 'prefix'. The default value for 'libdir' is '$(prefix)/lib'.

mandir
: A directory for man format documentation ("man pages"). The default value for 'mandir' depends on 'prefix'; 'mandir' is normally changed only indirectly through 'prefix'. The default value for 'mandir' is '$(prefix)/man'.

man*N*dir
: These are eight variables named 'man1dir', 'man2dir', etc. They name the specific directories for each man page section. For example, 'man1dir' by default holds the filename '$(mandir)/man1'; this directory contains 'emacs.1' (the man page for GNU Emacs). Similarly, 'man5dir' contains the value '$(mandir)/man5', indicating the directory which holds 'rcsfile.5' (the man page describing the rcs data file format). The default value for any of the 'man*N*dir' variables depends indirectly on 'prefix', and is normally changed only through 'prefix'. The default value for 'man*N*dir' is '$(mandir)/man*N*'.

man*N*ext
: *Not supported by Cygnus* configure. The GNU *Coding Standards* do not call for 'man1ext', 'man2ext', so the intended use for manext is apparently not parallel to 'mandir'. Its use is not clear. (See also Section 3.7 [Extensions to the GNU coding standards], page 23.)

infodir
: A directory for info format documentation. The default value for 'infodir' depends indirectly on 'prefix'; 'infodir' is normally changed only through 'prefix'. The default value for 'infodir' is '$(prefix)/info'.

docdir
: A directory for any documentation that is in a format other than those used by info or man. The default value for 'docdir' depends indirectly on 'prefix'; 'docdir' is normally changed only through 'prefix'. The default value for 'docdir' is '$(datadir)/doc'. *This variable is an extension to the* GNU *coding standards.* (See also Section 3.7 [Extensions to the GNU coding standards], page 23.)

includedir
: A directory for the header files accompanying the libraries installed in 'libdir'. The default value for 'includedir' depends on 'prefix'; 'includedir' is normally changed only indirectly through 'prefix'. The default value for 'includedir' is '$(prefix)/include'.

## 3.4 Host

The arguments to `configure` are *hosttypes*. By *hosttype* we mean the *environment* in which the source will be compiled. This need not necessarily be the same as the physical machine involved, although it usually is.

For example, if some obscure machine had the GNU `POSIX` emulation libraries available, it would be possible to configure most GNU source for a `POSIX` system and build it on the obscure host.

For more on this topic, see section "Host Environments" in *On Configuring Development Tools*.

## 3.5 Target

For building native development tools, or most of the other GNU tools, you need not worry about the target. The *target* of a configuration defaults to the same as the *host*.

For building cross development tools, please see section "Building Development Environments" in *On Configuring Development Tools*.

## 3.6 Adding information about local conventions

If you find that a tool does not get configured to your liking, or if `configure`'s conventions differ from your local conventions, you should probably consider *site-specific '`Makefile`' fragments*. See also Section 4.3 [Adding site info], page 27.

These are probably not the right choice for options that can be set from the `configure` command line or for differences that are host or target dependent.

Cygnus `configure` uses three types of '`Makefile`' fragments. In a generated '`Makefile`' they appear in the order: *target fragment*, *host fragment*, and *site fragment*. This allows host fragments to override target fragments, and site fragments to override both.

Host-specific '`Makefile`' fragments conventionally reside in the '`./config/`' subdirectory with names of the form '`mh-`*hosttype*'. They are used for hosts that require odd options to the standard compiler and for compile time options based on the host configuration.

Target-specific 'Makefile' fragments conventionally reside in the './config/' subdirectory with names of the form 'mt-*target*'. They are used for target dependent compile time options.

Site specific 'Makefile' fragments conventionally reside in the './config/' subdirectory with names of the form 'ms-*site*'. They are used to override host- and target-independent compile time options. Note that you can also override these options on the make invocation line.

## 3.7 Extensions to the GNU coding standards

The following additions to the GNU coding standards are required for Cygnus configure to work properly.

- The 'Makefile' must contain exactly one line starting with '####'. This line should follow any default macro definitions but precede any rules. Host, target, and site-specific 'Makefile' fragments will be inserted immediately after this line. If the line is missing, the fragments will not be inserted.
- Cygnus adds the following targets to each 'Makefile'. Their existence is not required for Cygnus configure, but they are documented here for completeness.

  info
  : Build all info files from texinfo source.

  install-info
  : Install all info files.

  clean-info
  : Remove all info files and any intermediate files that can be generated from texinfo source.

  Makefile
  : Calls ./config.status to rebuild the 'Makefile' in this directory.
- The following 'Makefile' targets have revised semantics:

  install
  : Should *not* depend on the target 'all'. If the program is not already built, 'make install' should fail. This allows you to install programs even when make would otherwise determine them to be out of date. This can happen, for example, when the result of a 'make all' is transported via tape to another machine for installation.

  clean
  : Should remove any file that can be regenerated by the 'Makefile', excepting only the 'Makefile' itself, and any links created by configure. That is, make all clean should return all directories to their original condition. If this is not done, then the command sequence

```
configure host1 ; make all install clean ;
configure host2 ; make all install
```

will fail because of intermediate files intended for *host1*.

- Cygnus adds the following macros to all 'Makefile.in' files, but you are not required to use them to run Cygnus `configure`.

  `docdir` The directory in which to install any documentation that is not either a `man` page or an `info` file. For `man` pages, see '`mandir`'; for `info`, see '`infodir`'.

  `includedir`
  The directory in which to install any header files that should be made available to users. This is distinct from the `gcc` include directory, which is intended for `gcc` only. Files in '`includedir`' may be used by `cc` as well.

- The following macros have revised semantics. Most of them describe installation directories; see also Section 3.3.3 [Full description of all installation subdirectories], page 20.

  `datadir` is used for host independent data files.

  `mandir` The default path for '`mandir`' depends on '`prefix`'.

  `infodir` The default path for '`infodir`' depends on '`prefix`'.

  `BISON` is assumed to have a `yacc` calling convention. To use GNU `bison`, use '`BISON=bison -y`'.

- Each Cygnus '`Makefile`' also conforms to one additional restriction:

  When libraries are installed, the line containing the call to '`INSTALL_DATA`' should always be followed by a line containing a call to '`RANLIB`' on the installed library. This is to accommodate systems that use `ranlib`. Systems that do not use `ranlib` can set '`RANLIB`' to "`echo`" in a host specific '`Makefile`' fragment.

# 4 Porting with configure

This section explains how to add programs, host and target configuration names, and site-specific information to Cygnus configure.

## 4.1 Adding configure to new programs

If you are writing a new program, you probably shouldn't worry about porting or configuration issues until it is running reasonably on some host. Then refer back to this section.

If your program currently has a configure script that meets the GNU standards (see section "How Configuration Should Work" in *GNU Coding Standards*, please do not add Cygnus configure. It should be possible to add this program without change to a Cygnus configure style source tree.

If the program is not target dependent, please consider using autoconf instead of Cygnus configure. autoconf is available from the Free Software Foundation; it is a program which generates an executable shell script called 'configure' by automatically finding information on the system to be configured on and embedding this information in the shell script. 'configure' scripts generated by autoconf require no arguments, and accept the same options as Cygnus configure. For detailed instructions on using autoconf, see section "How to organize and produce Autoconf scripts" in *Autoconf*.

To add Cygnus configure to an existing program, do the following:

- Make sure the 'Makefile' conforms to the GNU standard

  The coding standard for writing a GNU 'Makefile' is described in section "Makefile Conventions" in *GNU Coding Standards*. For technical information on writing a 'Makefile', see section "Writing Makefiles" in *GNU Make*.

- Add Cygnus extensions to the 'Makefile'

  These are described in Section 3.7 [Extensions to the GNU coding standards], page 23.

- Collect package specific definitions in a single file

  Many packages are best configured using a common 'Makefile' fragment which is included by all of the makefiles in the different directories of the package. In order to accomplish this, set the variable 'package_makefile_fragment' to the name of the file. It will be inserted into the final 'Makefile' before the target-specific fragment.

- Move host support from '`Makefile`' to fragments

  This usually involves finding sections of the '`Makefile`' that say things like "uncomment these lines for host *hosttype*" and moving them to a new file called '`./config/mh-`*hosttype*'. For more information, see Section 4.2 [Adding hosts and targets], page 27.

- Choose defaults

  If the program has compile-time options that determine the way the program should behave, choose reasonable defaults and make these '`Makefile`' variables. Be sure the variables are assigned their default values before the '`####`' line so that site-specific '`Makefile`' fragments can override them (see Section 3.7 [Extensions to the GNU coding standards], page 23).

- Locate configuration files

  If there is configuration information in header files or source files, separate it in such a way that the files have generic names. Then move the specific instances of those files into the '`./config/`' subdirectory.

- Separate host and target information

  Some programs already have this information separated. If yours does not, you will need to separate these two kinds of configuration information. *Host specific* information is the information needed to compile the program. *Target specific* information is information on the format of data files that the program will read or write. This information should live in separate files in the '`./config/`' subdirectory with names that reflect the configuration for which they are intended.

  At this point you might skip this step and simply move on. If you do, you should end up with a program that can be configured only to build *native* tools, that is, tools for which the host system is also the target system. Later, you could attempt to build a cross tool and separate out the target-specific information by figuring out what went wrong. This is often simpler than combing through all of the source code.

- Write `configure.in`

  Usually this involves writing shell script fragments to map from canonical configuration names into the names of the configuration files. These files will then be linked at configure time from the specific instances of those files in '`./config`' to files in the build directory with more generic names. (See also Section 3.1.2 [Build directories], page 10.) The format of '`configure.in`' is described in Section 3.2 [The `configure.in` input file], page 13.

- Rename '`Makefile`' to '`Makefile.in`'

At this point you should have a program that can be configured using Cygnus `configure`.

## 4.2 Adding hosts and targets

To add a host or target to a program that already uses Cygnus `configure`, do the following.

- Make sure the new configuration name is represented in '`config.sub`'. If not, add it. For more details, see the comments in the shell script '`config.sub`'.
- If you are adding a host configuration, look in '`configure.in`', in the *per-host* section. Make sure that your configuration name is represented in the mapping from host configuration names to configuration files. If not, add it. Also see Section 3.2 [The `configure.in` input file], page 13.
- If you are adding a target configuration, look in '`configure.in`', in the *per-target* section. Make sure that your configuration name is represented in the mapping from target configuration names to configuration files. If not, add it. Also see Section 3.2 [The `configure.in` input file], page 13.
- Look in '`configure.in`' for the variables '`files`', '`links`', '`host_makefile_frag`', and '`target_makefile_frag`'. The values assigned to these variables are the names of the configuration files, (relative to '`srcdir`') that the program uses. Make sure that copies of the files exist for your host. If not, create them. See also Section 3.2.1 [Variables available to `configure.in`], page 13.

This should be enough to `configure` for a new host or target configuration name. Getting the program to compile and run properly represents the hardest work of any port.

## 4.3 Adding site info

If some of the '`Makefile`' defaults are not right for your site, you can build site-specific '`Makefile`' fragments. To do this, do the following.

- Choose a name for your site. It must currently be less than eleven characters.
- If the program source does not have a '`./config/`' subdirectory, create it.
- Create a file called '`./config/ms-`*site*' where *site* is the name of your site. In it, set whatever '`Makefile`' variables you need to override to match your site's conventions.
- Configure the program with:

  ```
  configure ... --site=site
  ```

# Variable Index

# Concept Index

## U



## V



## W

# Table of Contents