4

Package configuration

Programs don't run in a vacuum: they interface with the outside world. The view of this outside world differs from location to location: things like host names, system resources, and local conventions will be different. Theoretically, you could change the program sources every time you install a package on a new system, but besides being a pain, it's very error-prone. All modern packages supply a method of *configuration*, a simplified way of adapting the sources to the environment in which the program will run. In this chapter, we'll look at common configuration conventions. We can divide system differences into one of three categories:

- The kind of hardware the package will run on. A compiler needs to generate the correct machine instructions, and an X server needs to know how to transfer data to the display hardware. Less well-written programs have hardware dependencies that could have been avoided with some forethought. We'll look at this in more detail in Chapter 11, *Hardware dependencies*.
- The system software with which it will interact. Differences between UNIX systems are significant enough that it will be necessary to make certain decisions depending on the system flavour. For example, a communications program will need to know what kind of network interface your system has. Programs that come from other systems may need significant rewriting to conform with UNIX library calls. We'll look at these dependencies in part 2 of this book, from Chapter 12, *Kernel dependencies* to Chapter 21, *Object files and friends*.
- The local configuration. These may include obvious things like the system name, aspects of program behaviour, information about tools used locally, or local system conventions.

In this chapter, we'll look at what local configuration entails, and *how* we tell the package about our chosen configuration.

Installation paths

Your system configuration may place constraints on where you can install the software. This is not normally a problem for individual systems, but on a large, heterogeneous network it could require more consideration.

Traditionally, non-system software has been installed in the hierarchy */usr/local*. This is not an æsthetically pleasing location: the hierarchy can become quite large, and in a network many systems might share the directory.

One of the best thought-out descriptions of a modern file system structure is in the *UNIX System V Application Binary Interface*, which is also similar to structures used by SunOS and the newer BSD variants. In brief, it specifies the following top-level directories:

/	The root directory.
/dev	The directory tree containing device files.
/etc	Directory for machine-specific configuration files.
/opt	Directory for add-on software.
/usr	This directory used to be *the* other file system on a UNIX machine. In the System V ABI it has lost most of its importance. The ABI states uses only for */usr/bin* and */usr/share*, and the name */usr* has lost its original meaning: the ABI specifies */usr* only as a location for system files that users may wish to access.
/usr/bin	is intended for "Utility programs and commands for the use of all applications and users". In practice, it's better to use this directory for system programs only.
/usr/share	The System V ABI states that */usr/share* is intended for "architecture-independent shareable files". In practice, those versions of System V that still have *man* pages put them in */usr/share/man*, and *terminfo* data are stored in */usr/share/lib/terminfo*. The rest of the directory may contain a few other odds and ends, but these two directories make up over 99% of the content. The choice of the location */usr/share* is not a happy choice: firstly, it is frequently a separate file system, but it must be mounted on a non-root file system, and secondly the man pages aren't really architecture-independent. The choice makes more sense from the point of view of the Unix Systems Group, who are concerned only with pure System V: the man pages are mainly independent of hardware architecture. However, in a real-world net you probably have two or three different operating systems, each with their own man pages.
/var	This directory contains files that are frequently modified. Typical subdirectories are */var/tmp* for temporary files and */var/spool* for printer output, *uucp* and *news*.

The System V ABI does not say anything about where to store user files. The Seventh Edition typically stored them as a subdirectory of */usr*, but newer systems have tended to store them in a directory called */home*.

The */opt* hierarchy resembles that of */usr*. A typical structure is:

/opt/bin	for executables.
/opt/man	for *man* pages—not */opt/share/man*, unlike the structure in */usr*.
/opt/lib	for additional files used by executables. In particular, this directory could contain library archives for compilers, as well as the individual passes of the compilers.
/opt/<pkg>	This is where the System V ABI places individual package data. Not many other systems follow it.
/opt/lib/<pkg>	This is where most packages place private data.

Using the */opt* hierarchy has one disadvantage: you may not want to have a separate file system. In modern systems, the solution is simple enough: place the directory where you want it, and create a symbolic link */opt* that points to it. This works only if your system has symbolic links, of course, so I have come to a compromise: I use */opt* on systems with symbolic links, and */usr/local* on systems without symbolic links.

Many packages compile pathnames into the code, either because it's faster that way, or because it's easier. As a result, you should set the path names before compilation—don't put off this task until you're ready to install, or you may run into problems where the packages are all nicely installed in the correct place and look for data in the wrong directories.

Preferred tools

Many of the most popular software packages are alternative tools. Free software such as *gcc*, *emacs* and *perl* have become so popular that they are frequently supplied with proprietary system releases, and many other systems have ported them and use them as standard tools. If you want to use such programs, you need to tell the configuration routines about them.

Depending on the tools you use, you may also need to change the flags that you pass to them. For example, if you compile with *gcc*, you may choose to include additional compiler flags such as `-fstrength-reduce`, which is specific to *gcc*.

Conveying configuration information

The goal of configuration is to supply the configuration information to the program sources. A good configuration mechanism will hide this from you, but it's helpful to understand what it's doing. In this section, we'll look under the covers—you can skip it if it looks too technical.

There are a number of possible ways to use configuration information: for example, the package may have separate communication modules for STREAMS and sockets, and the configuration routines may decide which of the two modules to compile. More typically, however, the configuration routines convey configuration information to the package by defining preprocessor variables indicating the presence or absence of a specific feature. Many packages provide this information in the *make* variable `CFLAGS`—for example, when you make *bash*, the GNU Bourne Again Shell, you see things like

```
$ make
gcc  -DOS_NAME='"FreeBSD"' -DProgram=bash           -DSYSTEM_NAME='"i386"'  \
-DMAINTAINER='"bug-bash@prep.ai.mit.edu"'  -O -g -DHAVE_SETLINEBUF -DHAVE_VFPRINTF \
-DHAVE_UNISTD_H -DHAVE_STDLIB_H -DHAVE_LIMITS_H        -DHAVE_GETGROUPS \
-DHAVE_RESOURCE -DHAVE_SYS_PARAM -DVOID_SIGHANDLER -DOPENDIR_NOT_ROBUST \
-DINT_GROUPS_ARRAY -DHAVE_WAIT_H        -DHAVE_GETWD -DHAVE_DUP2 -DHAVE_STRERROR \
-DHAVE_DIRENT -DHAVE_DIRENT_H -DHAVE_STRING_H        -DHAVE_VARARGS_H -DHAVE_STRCHR \
-DHAVE_STRCASECMP -DHAVE_DEV_FD        -D"i386" -D"FreeBSD" -DSHELL -DHAVE_ALLOCA \
-I. -I. -I././lib/ -c shell.c
```

The `-D` arguments pass preprocessor variables that define the configuration information.

An alternative method is to put this information in a file with a name like *config.h*. Taylor *uucp* does it this way: in *config.h* you will find things like:

```
/* If your compiler supports prototypes, set HAVE_PROTOTYPES to 1.  */
#define HAVE_PROTOTYPES 1

/* Set ECHO_PROGRAM to a program which echoes its arguments; if echo
   is a shell builtin you can just use "echo".  */
#define ECHO_PROGRAM "echo"

/* The following macros indicate what header files you have.  Set the
   macro to 1 if you have the corresponding header file, or 0 if you
   do not.  */
#define HAVE_STDDEF_H 1 /* <stddef.h> */
#define HAVE_STDARG_H 1 /* <stdarg.h> */
#define HAVE_STRING_H 1 /* <string.h> */
```

I prefer this approach: you have all the configuration information in one place, it is documented, and it's more reliable. Assuming that the *Makefile* dependencies are correct, any change to *config.h* will cause the programs to be recompiled on the next *make*. As we will see in Chapter 5, *Building the package*, page 68, this usually doesn't happen if you modify the *Makefile*.

Typically, configuration information is based on the kind of operating system you run and the kind of hardware you use. For example, if you compile for a Sparc II running SunOS 4.1.3, you might define `sparc` to indicate the processor architecture used and `sunos4` to indicate the operating system. Since SunOS 4 is basically UNIX, you might also need to define `unix`. On an Intel 486 running UnixWare you might need to define `i386` for the processor architecture,* and `SVR4` to indicate the operating system. This information is then used in the source files as arguments to preprocessor `#ifdef` commands. For example, the beginning of each source file, or a general configuration file, might contain:

```
#ifdef i386
#include "m/i386.h"
#endif
#ifdef sparc
#include "m/sparc.h"
#endif
```

* Why not `i486`? The processor is an Intel 486, but the architecture is called the i386 architecture. You also use `i386` when compiling for a Pentium.

```
#ifdef sunos4
#include "s/sunos4.h"
#endif
#ifdef SVR4
#include "s/usg-4.0.h"
#endif
```

You can get yourself into real trouble if you define more than one machine architecture or more than one operating system. Since configuration is usually automated to some extent, the likelihood of this is not very great, but if you end up with lots of double definitions when compiling, this is a possible reason.

Configuration through the preprocessor works nicely if the hardware and software both exactly match the expectations of the person who wrote the code. In many cases, this is not the case: looking at the example above, note that the file included for SVR4 is *s/usg-4.0.h*, which suggests that it is intended for UNIX System V release 4.0. UnixWare is System V release 4.2. Will this work? Maybe. It could be that the configuration mechanism was last revised before System V.4.2 came out. If you find a file *s/usg-4.2.h*, it's a good idea to use it instead, but otherwise it's a matter of trial and error.

Most software uses this approach, although it has a number of significant drawbacks:

- The choices are not very detailed: for example, most packages don't distinguish between Intel 386 and Intel 486, although the latter has a fbating point coprocessor and the former doesn't.
- There is no general consensus on what abbreviations to use. For UnixWare, you may find that the correct operating system information is determined by `USG` (USG is the Unix Systems Group, which, with some interruption,* is responsible for System V), `SYSV`, `SVR4`, `SYSV_4`, `SYSV_4_2` or even `SVR3`. This last can happen when the configuration needed to be updated from System V.2 to System V.3, but not again for System V.4.
- The choice of operating system is usually determined by just a couple of differences. For example, base System V.3 does not have the system call `rename`, but most versions of System V.3 that you will find today have it. System V.4 does have `rename`. A software writer may use `#ifdef SVR4` only to determine whether the system has the `rename` system call or not. If you are porting this package to a version of System V.3.2 with `rename`, it might be a better idea to define `SVR4`, and not `SVR3`.
- Many aspects attributed to the kernel are in fact properties of the system library. As we will see in the introduction to Part 2 of this book, there is a big difference between kernel functionality and library functionality. The assumption is that a specific kernel uses the library with which it is supplied. The situation is changing, though: many companies sell systems without software development tools, and alternative libraries such as the GNU C library are becoming available. Making assumptions about the library based on the kernel was never a good idea—now it's completely untenable. For example, the GNU C

* The first USG was part of AT&T, and was superseded by UNIX Systems Laboratories (USL). After the sale of USL to Novell, USL became Novell's UNIX Systems Group.

library supplies a function `rename` where needed, so our previous example would fail even on a System V.3 kernel without a `rename` system call if it uses the GNU C library. As you can imagine, many packages break when compiled with the GNU C library, through their own fault, not that of the library.

In the example above, it would make a whole lot more sense to define a macro `HAS_RENAME` which can be set if the `rename` function is present. Some packages use this method, and the GNU project is gradually working towards it, but the majority of packages base their decisions primarily on the combination of machine architecture and operating system.

The results of incorrect configuration can be far-reaching and subtle. In many cases, it looks as if there is a bug in the package, and instead of reconfiguring, you can find yourself making significant changes to the source. This can cause it to work for the environment in which it is compiled, but to break it for anything else.

What do I need to change?

A good configuration mechanism should be able to decide the hardware and software dependencies that interest the package, but only you can tell it about the local preferences. For example, which compiler do you use? Where do you want to install the executables? If you don't know the answers to these questions, there's a good chance that you'll be happy with the defaults chosen by the configuration routines. On the other hand, you may want to use *gcc* to compile the package, and to install the package in the */opt* hierarchy. In all probability, you'll have to tell the configuration routines about this. Some configuration routines will look for *gcc* explicitly, and will take it if they find it. In this case, you may have a reason to tell the configuration routines *not* to use *gcc*.

Some packages have a number of local preferences: for example, do you want the package to run with X11 (and possibly fail if X isn't running)? This sort of information *should* be in the *README* file.

Creating configuration information

A number of configuration methods exist, none of them perfect. In most cases you don't get a choice: you use the method that the author of the package decided upon. The first significant problem can arise at this point: *what* method does he use? This is not always easy to figure out—it *should* be described in a file called *README* or *INSTALL* or some such, but occasionally you just find cryptic comments in the *Makefile*.

In the rest of this chapter we'll look at configuration via multiple *Makefile* targets, manual configuration, shell scripts, and *imake*, the X11 configuration mechanism. In addition, the new BSD *make* system includes a system of automatic configuration: once it is set up, you don't have to do anything, assuming you already have a suitable *Makefile*. We'll look at this method in more detail in Chapter 19, *Make*, page 323.

Multiple Makefile targets

Some packages anticipate every possibility for you and supply a customized Makefile. For example, when building *unzip*, a free uncompression utility compatible with the DOS package *PK-ZIP*, you would find:

```
$ make
If you're not sure about the characteristics of your system, try typing "make
generic".  If the compiler barfs and says something unpleasant about "timezone
redefined," try typing "make clean" followed by "make generic2".  One of these
actions should produce a working copy of unzip on most Unix systems.  If you
know a bit more about the machine on which you work, you might try "make list"
for a list of the specific systems supported herein.  And as a last resort, feel
free to read the numerous comments within the Makefile itself.  Note that to
compile the decryption version of UnZip, you must obtain the full versions of
crypt.c and crypt.h (see the "Where" file for ftp and mail-server sites).  Have
an excruciatingly pleasant day.
```

As the comments suggest, typing *make generic* should work most of the time. If it doesn't, looking at the *Makefile* reveals a whole host of targets for a number of combined hardware/software platforms. If one of them works for you, and you can find which one, then this might be an easy way to go. If none does, you might find yourself faced with some serious Makefile rewriting. This method has an additional disadvantage that it might compile with no problems and run into subtle problems when you try to execute it—for example, if the program expects System V `sigpause` and your system supplies BSD `sigpause`,* the build process may complete without detecting any problems, but the program will not run correctly, and you might have a lot of trouble finding out why.

Manual configuration

Modifying the *Makefile* or *config.h* manually is a better approach than multiple *Makefile* targets. This seemingly arduous method has a number of advantages:

- You get to see what is being changed. If you have problems with the resultant build, it's usually relatively easy to pin-point them.
- Assuming that the meanings of the parameters are well documented, it can be easier to modify them manually than run an automated procedure that hides much of what it is doing.
- If you find you *do* need to change something, you can usually do it fairly quickly. With an automated script, you may need to go through the whole script to change a single minor parameter.

On the down side, manual configuration requires that you understand the issues involved: you can't do it if you don't understand the build process. In addition, you may need to repeat it every time you get an update of the package, and it is susceptible to error.

* See Chapter 13, *Signals*, pages 190 and 192 for further information.

Configuration scripts

Neither multiple Makefile targets nor manual modification of the Makefile leave you with the warm, fuzzy feeling that everything is going to work correctly. It would be nice to have a more mechanized method to ensure that the package gets the correct information about the environment in which it is to be built. One way to do this is to condense the decisions you need to make in manual configuration into a shell script. Some of these scripts work very well. A whole family of configuration scripts has grown up in the area of electronic mail and news. Here's part of the configuration script for C news, which for some reason is called *build*:

```
$ cd conf
$ build
This interactive command will build shell files named doit.root,
doit.bin, doit.news, and again.root to do all the work.  It will not
actually do anything itself, so feel free to abort and start again.

C News wants to keep most of its files under a uid which preferably
should be all its own.  Its programs, however, can and probably should
be owned by another user, typically the same one who owns most of the
rest of the system.  (Note that on a system running NFS, any program
not owned by "root" is a gaping security hole.)
What user id should be used for news files [news]? RETURN pressed
What group id should be used for news files [news]? RETURN pressed
What user id should be used for news programs [bin]? RETURN pressed
What group id should be used for news programs [bin]? RETURN pressed
Do the C News sources belong to bin [yes]? no
You may need to do some of the installation procedures by hand
after the software is built; doit.bin assumes that it has the
power to create files in the source directories and to update
the news programs.

It would appear that your system is among the victims of the
4.4BSD / SVR4 directory reorganization, with (e.g.) shared
data in /usr/share.  Is this correct [yes]? RETURN pressed
This will affect where C News directories go.  We recommend
making the directories wherever they have to go and then making
symbolic links to them under the standard names that are used
as defaults in the following questions.  Should such links
be made [yes]? no
```

We chose not to use the symbolic links: the script doesn't say why this method is recommended, they don't buy us anything, and symbolic links mean increased access time.

The configuration script continues with many more questions like this. We'll pick it up at various places in the book.

The flexibility of a shell script is an advantage when checking for system features which are immediately apparent, but most of them require that you go through the whole process from start to finish if you need to modify anything. This can take up to 10 minutes on each occasion, and they are often interactive, so you can't just go away and let it do its thing.

GNU package configuration

Most GNU project packages supply another variety of configuration script. For more details, see *Programming with GNU Software*, by Mike Loukides. GNU configuration scripts sometimes expect you to know the machine architecture and the operating system, but they often attempt to guess if you don't tell them. The main intention of the configuration utility is to figure out which features are present in your particular operating system port, thus avoiding the problems with functions like rename discussed on page 51. Taylor *uucp* uses this method:

```
$ sh configure
checking how to run the C preprocessor
checking whether -traditional is needed see page 351
checking for install                    the install program, page 128
checking for ranlib                     see page
checking for POSIXized ISC              Interactive POSIX extensions?
checking for minix/config.h             MINIX specific
checking for AIX                        IBM UNIX
checking for -lseq                      libseq.a needed?
checking for -lsun                      libsun.a?
checking whether cross-compiling
checking for lack of working const      see page 339
checking for prototypes                 does the compiler understand function prototypes?
checking if '#!' works in shell scripts
checking for echo program               is echo a program or a builtin?
checking for ln -s                      do we have symbolic links? (page 218)
```

This method makes life a whole lot easier if the package has already been ported to your particular platform, and if you are prepared to accept the default assumptions that it makes, but can be a real pain if not:

- You may end up having to modify the configuration scripts, which are not trivial.
- It's not always easy to configure things you want. In the example above, we accepted the default compiler flags. If you want maximum optimization, and the executables should be installed in */opt/bin* instead of the default */usr/local/bin*, running *configure* becomes significantly more complicated:*

  ```
  $ CFLAGS="-O3 -g" sh configure --prefix=/opt
  ```

- The scripts aren't perfect. You should really check the resultant *Makefile*s, and you will often find that you need to modify them. For example, the configuration scripts of many packages, including the GNU debugger, *gdb*, do not allow you to override the preset value of CFLAGS. In other cases, you can run into a lot of trouble if you do things that the script didn't expect. I once spent a couple of hours trying to figure out the behaviour of the GNU *make* configuration script when porting to Solaris 2.4:

* This example uses the feature of modern shells of specifying environment variables at the beginning of the command. The program being run is *sh*, and the definition of CFLAGS is exported only to the program being started.

```
$ CFLAGS="O3 -g" configure --prefix=/opt
creating cache ./config.cache
checking for gcc... gcc
checking whether we are using GNU C... yes
checking how to run the C preprocessor... gcc -E
checking whether cross-compiling... yes
```

Although this was a normal port, it claimed I was trying to cross-compile. After a lot of experimentation, I discovered that the configuration script checks for cross-compilation by compiling a simple program. If this compilation fails for any reason, the script assumes that it should set up a cross-compilation environment. In this case, I had mistakenly set my `CFLAGS` to `O3 -g`—of course, I had meant to write `-O3 -g`. The compiler looked for a file *O3* and couldn't find it, so it failed. The configuration script saw this failure and assumed I was cross-compiling.

- In most cases, you need to re-run the configuration script every time a package is updated. If the script runs correctly, this is not a problem, but if you need to modify the Makefile manually, it can be a pain. For example, *gdb* creates 12 *Makefile*s. If you want to change the `CFLAGS`, you will need to modify each of them every time you run *configure*.
- Like all configuration scripts, the GNU scripts have the disadvantage of only configuring things they know about. If your *man* program requires pre-formatted man pages, you may find that there is no way to configure the package to do what you want, and you end up modifying the Makefile after you have built it.

Modifying automatically build *Makefile*s is a pain. An alternative is to modify *Makefile.in*, the raw *Makefile* used by *configure*. That way, you will not have to redo the modifications after each run of *configure*.

imake

imake is the X11 solution to package configuration. It uses the C preprocessor to convert a number of configuration files into a *Makefile*. Here are the standard files for X11R6:

- *Imake.tmpl* is the main configuration file that is passed to the C preprocessor. It is responsible for including all the other configuration files via the preprocessor `#include` directive.
- *Imake.cf* determines the kind of system upon that *imake* is running. This may be based on preprocessor variables supplied by default to the preprocessor, or on variables compiled in to *imake*.
- *site.def* describes local preferences. This is one of the few files that you should normally consider modifying.
- As its name implies, *<vendor>.cf* has a different name for each platform. *Imake.tmpl* decides which file to include based on the information returned by *Imake.cf*. For example, on BSD/OS the file *bsdi.cf* will be included, whereas under SunOS 4 or Solaris 2 the file *sun.cf* will be included.

- *Imake.rules* contains preprocessor macros used to define the individual *Makefile* targets.
- *Imakefile* is part of the package, not the *imake* configuration, and describes the package to *imake*.

You don't normally run *imake* directly, since it needs a couple of pathname parameters: instead you have two possibilities:

- Run *xmkmf*, which is a one-line script that supplies the parameters to *imake*.
- Run *make Makefile*. This assumes that some kind of functinoal *Makefile* is already present in the package.

Strangely, *make Makefile* is the recommended way to create a new *Makefile*. I don't agree: one of the most frequent reasons to make a new *Makefile* is because the old one doesn't work, or because it just plain isn't there. If your *imake* configuration is messed up, you can easily remove all traces of a functional *Makefile* and have to restore the original version from tape. *xmkmf* always works, and anyway, it's less effort to type.

Once you have a *Makefile*, you may not be finished with configuration. If your package contains subdirectories, you may need to create *Makefile*s in the subdirectories as well. In general, the following sequence will build most packages:

```
$ xmkmf                 run imake against the Imakefile
$ make Makefiles        create subordinate Makefiles
$ make depend           run makedepend against all Makefiles
$ make                  make the packages
$ make install          install the packages
```

These commands include no package-dependent parameters—the whole sequence can be run as a shell script. Well, yes, there are minor variations: *make Makefiles* fails if there are no subordinate *Makefiles* to be made, and sometimes you have targets like a *make World* instead of *make* or *make all*, but in general it's very straightforward.

If your *imake* configuration files are set up correctly, and the package that you are porting contains no obscenities, this is all you need to know about *imake*, which saves a lot of time and is good for your state of mind. Otherwise, check *Software Portability with imake*, by Paul DuBois, for the gory details.