2

Unpacking the goodies

Before you can start porting, you need to put the sources on disk. We use the term *source tree* to refer to the directory or hierarchy of directories in which the package is stored. Unpacking the archives may not be as trivial as it seems: software packages are supplied in many different formats, and it is not always easy to recognize the format. In this chapter, we'll look at how to extract the sources to create the source tree. In Chapter 3, *Care and feeding of source trees*, we'll see how the source tree changes in the course of a port, and what you can do to keep it in good shape.

Getting the sources

The standard way to get software sources is on some form of storage medium, such as CD-ROM or tape. Many packages are also available online via the Internet. The choice is not as simple as it seems:

Software from the Internet

If you have an Internet connection, and if the software is available on the net, it's tempting to just copy it across the net with *ftp*. This may not be the best choice, however. Some packages are very big. The compressed sources of the GNU C compiler, for example, occupy about 6 MB. You can't rely on a typical 56 kb/s line to transfer more than about 2 kilobytes per second.* At this speed, it will take nearly an hour to copy the archives. If you're connected via a SLIP line, it could take several hours.

Gaining access to the archive sites is not always trivial: many sites have a maximum number of users. In particular, `prep.ai.mit.edu`, the prime archive site for *gcc*, is frequently overloaded, and you may need several attempts to get in.

In addition, copying software over the net is not free. It may not cost you money, but somebody has to pay for it, and once you have the software, you need somewhere to store it, so you don't really save on archive media.

* Of course, it *should* approach 7 kilobytes per second, but network congestion can pull this figure down to a trickle.

Choice of archive medium

If you do choose to get your software on some other medium, you have the choice between CD-ROM and tape. Many archive sites will send you tapes if you ask for them. This may seem like a slow and old-fashioned way to get the software, but the bandwidth is high:* DAT and Exabyte tapes can store 2 GB per tape, so a single tape could easily contain as much software as you can duplicate in a week. In addition, you don't need to make a backup before you start.

Software on CD-ROM is not as up-to-date as a freshly copied tape, but it's easy to store and reasonably cheap. Many companies make frequent CD editions of the more widely known archive sites—for example, Walnut Creek CD-ROM has editions of most commonly known software, frequently pre-ported, and Prime Time Freeware issues a pair of CD-ROMs twice a year with 5 GB of compressed software including lesser-known packages. This can be worth it just to be able to find packages that you would otherwise not even have known about.

If you have already ported a previous version of the package, another alternative is to use *diff*s to bring the archive up to date. We'll look at this on page 29.

Archives

You frequently get pure source trees on CD-ROM, but other media, and also many CD-ROMs, transform the source tree several times:

- A source tree is usually supplied in an *archive*, a file containing a number of other files. Like a paper bag around groceries, an archive puts a wrapper around the files so that you can handle them more easily. It does not save any space—in fact, the wrapper makes it slightly larger than the sum of its files.
- Archives make it easier to handle files, but they don't do anything to save space. Much of the information in files is redundant: each byte can have 256 different values, but typically 99% of an archive of text or program sources will consist of the 96 printable ASCII characters, and a large proportion of these characters will be blanks. It makes sense to encode them in a more efficient manner to save space. This is the purpose of *compression* programs. Modern compression programs such as *gzip* reduce the size of an archive by up to 90%.
- If you want to transfer archives by electronic mail, you may also need to *encode* them to comply with the allowable email character set.
- Large archives can become very unwieldy. We have already seen that it can take several hours to transfer *gcc*. If the line drops in this time, you may find that you have to start the file again. As a result, archives are frequently *split* into more manageable chunks.

The most common form of archive you'll find on the Internet or on CD-ROM is *gzipped tar*, a *tar* archive that has been compressed with *gzip*. A close second is *compressed tar*, a *tar*

* To quote a fortune from the *fortune* program: *Never underestimate the bandwidth of a station wagon full of tapes.*.

archive that has been compressed with *compress*. From time to time, you'll find a number of others. In the following sections we'll take a brief look at the programs that perform these tasks and recover the data.

Archive programs

A number of archive programs are available:

- *tar*, the tape archive program, is the all-time favourite. The chances are about 95% that your archive will be in *tar* format, even if it has nothing to do with tape.
- *cpio* is a newer file format that once, years ago, was intended to replace *tar*. *cpio* archives suffer from compatibility problems, however, and you don't see them very often.
- *ar* is a disk archive program. It is occasionally used for source archives, though nowadays it is almost only used for object file archives. The *ar* archive format has never been completely standardized, so you get an *ar* archive from a different machine, you might have a lot of trouble extracting it. We'll look at *ar* formats again in , on page 383.
- *shar* is the shell archive program. It is unique amongst archive programs in never using non-printing characters, so *shar* archives can be sent by mail. You can extract *shar* archives simply by feeding them to a (Bourne) shell, though it is safer to use a program like *unshar*.

Living with tar

tar is a relatively easy program to use, but the consequences of mistakes can be far-reaching. In the following sections, we'll look at how to use *tar* and how to avoid trouble.

Basic use

When it comes to unpacking software, one or two tar commands can meet all your needs. First, you often want to look at the contents before unpacking. Assuming that the archive is named *et1.3.tar*, the following command lists the files in the archive:

```
$ tar tf et1.3.tar
et1.3/
et1.3/bell.c
pet1.3/bltgraph.c
et1.3/BLURB
```

The `t` option stands for table of contents, and the `f` option means "use the next parameter in the command (*et1.3.tar*) as the name of the archive to list."

To read in the files that were listed, use the command:

```
$ tar xfv et1.3.tar
et1.3/
et1.3/bell.c
pet1.3/bltgraph.c
et1.3/BLURB
```

The list looks the same, but this time the command actually creates the directory *et1.3* if necessary, and then creates the contents. The `x` option stands for *extract*, and the `f` option has the same meaning as before. The `v` option means "verbose" and is responsible for generating the list, which gives you the assurance that the command is actually doing something.

To bundle some files into an archive, use a command like:

```
$ tar cvf et1.3.tar et1.3
```

This command packs everything in the *et1.3* directory into an archive named *et1.3.tar* (which is where we started). The `c` option stands for "create" and the `v` option for "verbose." This time, the `f` means "use the next parameter in the command (*et1.3.tar*) as the archive to create."

Absolute pathnames

Many versions of *tar* have difficulties with absolute pathnames. If you back up a directory */usr/foo*, they will only be able to restore it to exactly this directory. If the directory is */usr/bin*, and you're trying to restore programs like *sh*, this could give you serious problems. Some versions of *tar* have an option to ignore the leading */*, and others, such as GNU *tar*, ignore it unless you tell them otherwise.

Symbolic links

Many versions of *tar* will only back up a symbolic link, not the file or directory to which it points. This can be very embarrassing if you send somebody a tape with what should be a complete software package, and it arrives with only a single symbolic link.

Tape block size

Many DDS (DAT) drives work better with high blocking factors, such as 65536 bytes per block (128 "tape blocks"). You can do this with the option `b` (block size):

```
$ tar cvfb /dev/tape 128 foo-dir
```

Unfortunately, this can cause problems too. Some DDS drives cannot read tapes with block sizes of more than 32768 bytes, and some versions of *tar*, such as SGI IRIS 5.x, cannot handle tapes blocked larger than 20 tape blocks (10240 bytes). This is a show-stopper if you have a tape which is really blocked at more than this size: you just won't be able to read it directly. You can solve this problem by installing GNU *tar* or piping the archive through *dd*:

```
$ dd if=/dev/rmt/ctape0 ibs=128b obs=2b | tar xvf -
```

File names

Most versions of *tar* perform filename matching based on the exact text as it appears on the tape. If you want to extract specific files, you must use the names by which they are known in the archive. For example, some versions of *tar* may end up writing absolute names with two leading slashes (like *//usr/bin/sh*, for example). This doesn't worry the operating system, which treats multiple leading slashes the same as a single leading slash, but if you want to

extract this file, you need to write:

```
$ tar x //usr/bin/sh
```

File name sorting

A *tar* archive listing with *tar tv* deliberately looks very much like a listing done with *ls -l*. There is one big difference, however: *ls -l* sorts the file names by name before displaying them, whereas *tar*, being a serial archive program, displays the names in the order in which they occur in the archive. The list may look somewhat sorted, depending on how the archive was created, but you can't rely on it. This means that if you are looking for a file name in an archive, you should not be misled if it's not where you expect to find it: use tools like *grep* or *sort* to be sure.

tar: dir - cannot create

With System V systems, you may see things like:

```
$ tar xvf shellutils-1.9.4.tar
tar: shellutils-1.9.4/ - cannot create
x shellutils-1.9.4/COPYING, 17982 bytes, 36 tape blocks
x shellutils-1.9.4/COPYING.LIB, 25263 bytes, 50 tape blocks
tar: shellutils-1.9.4/lib/ - cannot create
x shellutils-1.9.4/lib/Makefile.in, 2868 bytes, 6 tape blocks
x shellutils-1.9.4/lib/getopt.h, 4412 bytes, 9 tape blocks
```

This "bug" has been around so long that you might suspect that it is an insider joke. In fact, it is a benign compatibility problem. The POSIX.2 standard *tar* format allows archives to contain both directory and file names, although the directory names are not really necessary: assuming it has permission, *tar* creates all directories necessary to extract a file. The only use of the directory names is to specify the modification time and permissions of the directory. Older versions of *tar*, including System V *tar*, do not include the directory names in the archive, and don't understand them when they find them. In this example, we have extracted a POSIX.2 *tar* archive on a System V system, and it doesn't understand (or need) the directory information. The only effect is that the directories will not have the correct modification timestamps and possibly not the correct permissions.

Losing access to your files

Some versions of *tar*, notably System V versions, have another trick in store: they restore the original *owner* of the files, even if that owner does not exist. That way you can lose access to your files completely if they happen to have permissions like `rw-------`. You can avoid this by using the `o` flag (restore ownership to current user).

It would be nice to be able to say "make a rule of always using the `o` flag". Unfortunately, other versions of *tar* define this flag differently—check your man pages for details.

Multivolume archives

tar can also handle multi-volume archives, in other words archives that go over more than one tape. The methods used are not completely portable: one version of *tar* may not be able to read multivolume archives written by a different version. Some versions of *tar* just stop writing data at the end of one tape and continue where they left off at the beginning of the next reel, whereas others write header information on the second tape to indicate that it is a continuation volume. If possible, you should avoid writing multivolume archives unless you are sure that the destination system can read them. If you run into problems with multivolume archives you can't read, you might save the day with something like:

```
$ (dd if=$TAPE
++ echo 1>&2 Change tapes and press RET
++ read confirmation      the name of the variable isn't important
++ dd if=$TAPE
++ echo 1>&2 Change tapes and press RET
++ read confirmation
++ dd if=$TAPE) | tar xvf -
```

This uses *dd* to copy the first tape to *stdout*, then prints a message and waits for you to press the enter key, copies a second tape, prompts and waits again, and then copies a third tape. Since all the commands are in parentheses, the standard output of all three *dd* commands is piped into the *tar* waiting outside. The *echo* commands need to go to *stderr* (that's the 1>&2) to get displayed on the terminal—otherwise they would be piped into the *tar*, which would not appreciate it.

This only works if the version of *tar* you use doesn't put any header information (like reel number and a repeat of the file header) at the beginning of the subsequent reels. If it does, and you can't find a compatible *tar* to extract it again, the following method may help. Assuming a user of an SCO system has given you a large program *foo* spread over 3 *diskettes*, each of which contains header information that your *tar* doesn't understand, you might enter

```
$ tar x foo             extract first part from first floppy
$ mv foo foo.0          save the first part
$ tar x foo             extract second part from second floppy
$ mv foo foo.1          save the second part
$ tar x foo             extract third part from third floppy
$ mv foo foo.2          save the third part
$ cat foo.* >foo        concatenate them
$ rm foo.*              and remove the intermediate files
```

Extracting an archive with tar

Using *tar* to extract a file is normally pretty straightforward. You can cause a lot of confusion, however, if you extract into the wrong directory and it already contains other files you want to keep. Most archives contain the contents of a single directory as viewed from the parent directory—in other words, the name of the directory is the first part of all file names. All GNU software follows this rule:

```
$ tar tvf groff-1.09.tar
drwxr-xr-x jjc/staff        0 Feb 19 14:15 1994 groff-1.09/
drwxr-xr-x jjc/staff        0 Feb 19 14:13 1994 groff-1.09/include/
-rw-r--r-- jjc/staff      607 Sep 21 12:03 1992 groff-1.09/include/Makefile.sub
-rw-r--r-- jjc/staff     1157 Oct 30 07:38 1993 groff-1.09/include/assert.h
-rw-r--r-- jjc/staff     1377 Aug  3 12:34 1992 groff-1.09/include/cmap.h
-rw-r--r-- jjc/staff     1769 Aug 10 15:48 1992 groff-1.09/include/cset.h
```

Others, however, show the files from the viewpoint of the directory itself—the directory name is missing in the archive:

```
$ tar tvf blaster.tar
-rw-r--r-- 400/1         5666 Feb 14 01:44 1993 README
-rw-r--r-- 400/1         3638 Feb 14 01:44 1993 INSTALL
-r--r--r-- 400/1         2117 Feb 14 01:44 1993 LICENSE
-rw-r--r-- 400/1         2420 Feb 14 15:17 1993 Makefile
-rw-r--r-- 400/1         3408 Feb 14 01:44 1993 sb_asm.s
-rw------- 400/1        10247 Feb 14 01:44 1993 stream.c
-rw-r--r-- 400/1         1722 Feb 14 04:10 1993 apps/Makefile
```

If you have an archive like the first example, you want to be in the parent directory when you extract the archive; in the second case you need to first create the directory and then *cd* to it. If you extract the second archive while in the parent directory, you will face a lot of cleaning up. In addition, there is a good chance that files with names like *README*, *INSTALL* and *LICENSE* may already be present in that directory, and extracting this archive would overwrite them. There are a couple of ways to avoid these problems:

- Always look at the archive contents with *tar t* before extracting it. Once you have looked at the archive contents, you can change to the correct directory into which to extract it. In the case of *groff* above, you might choose a directory name like *~/mysources*[*]. In the case of *blaster*, you could create a directory *~/mysources/blaster* and extract into that directory.
- Alternatively, you can *always* create a subdirectory and extract there, and then rename the directory. In the first example, you might create a directory *~/mysources/temp*. After extraction, you might find that the files were in a directory *~/mysources/temp/groff-1.09*, so you could move them with

  ```
  $ mv groff-1.09 ..
  ```

  If they extract directly into *temp*, you can rename the directory:

  ```
  $ cd ..
  $ mv temp groff-1.09
  ```

  This method may seem easier, but in fact there are a couple of problems with it:

  - You need to choose a directory name that doesn't clash with the real name. That's why we used the name *temp* in this example: otherwise it won't be possible to rename the directory in the first example, since you would be trying to overwrite the directory with one of its own subdirectories.

* A number of shells use the shorthand notation `~/` to refer to your home directory.

- Not all flavours of UNIX allow you to move directories.

The command to extract is almost identical to the command to list the archive—a clear case for a shell with command line editing:

```
$ tar tvf groff-1.09.tar  list the archive
$ tar xvf groff-1.09.tar  extract the archive
```

Frequently your *tar* archive will be compressed in some way. There are methods for extracting files directly from compressed archives. We'll examine these when we look at compression programs on page .

Compression programs

If the archive is compressed, you will need to uncompress it before you can extract files from it. UNIX systems almost invariably use one of three compression formats:

- *compressed* files are created with the *compress* program and extracted with *uncompress*. They can be up to 70% smaller than the original file. The *zcat* program will uncompress a compressed file to the standard output.
- *gzipped* files are created by *gzip* and extracted by *gunzip*. They can be up to 90% smaller than the original file. *gunzip* will also uncompress compressed or packed files.
- *packed* files are obsolete, though you still occasionally see packed *man* pages. They are created by the *pack* program and uncompressed by the *unpack* program. The *pcat* program will uncompress a packed file to the standard output.

Each of these programs is installed with three different names. The name determines the behavior. For example, *gzip* is also known as *gunzip* and *zcat*:

```
$ ls -li /opt/bin/gzip /opt/bin/gunzip /opt/bin/zcat
 13982 -rwxr-xr-x   3 grog     wheel       77824 Nov  5  1993 /opt/bin/gunzip
 13982 -rwxr-xr-x   3 grog     wheel       77824 Nov  5  1993 /opt/bin/gzip
 13982 -rwxr-xr-x   3 grog     wheel       77824 Nov  5  1993 /opt/bin/zcat
```

The `-i` option to *ls* tells it to list the inode number, which uniquely identifies the file. In this case, you will see that all three names are linked to the same file (and that the link count field is 3 as a result). You will notice that *gzip* has also been installed under then name *zcat*, replacing the name used by *compress*. This is not a problem, since *gzcat* can do everything that *zcat* can do, but it can lead to confusion if you rely on it and one day try to extract a *gzipped* file with the real *zcat*.

Encoded files

Most archive programs and all compression programs produce output containing non-printable characters. This can be a problem if you want to transfer the archive via electronic mail, which cannot handle all binary combinations. To solve this problem, the files can be *encoded*: they are transformed into a representation that contains only printable characters. This has the disadvantage that it makes the file significantly larger, so it is used only when absolutely

necessary. Two programs are in common use:

- *uuencode* is by far the most common format. The companion program *uudecode* will extract from standard input.
- *btoa* format is used to some extent in Europe. It does not expand the file as much as *uuencode* (25% compared to 33% with *uuencode*), and is more resistant to errors. You decode the file with the *atob* program.

Split archives

Many *ftp* sites split large archives into equal-sized chunks, typically between 256 kB and 1.44 MB (a fbppy disk image). It's trivial to combine them back to the original archive: *cat* will do just that. For example, if you have a set of files *base09.000* through *base09.013* representing a gzipped tar archive, you can combine them with:

```
$ cat base09.* > base09.tar.gz
```

This will, of course, require twice the amount of storage, and it takes time. It's easier to extract them directly:

```
$ cat base09.* | gunzip | tar xvf -
drwxr-xr-x root/wheel        0 Aug 23 06:22 1993 ./sbin/
-r-xr-xr-x bin/bin      106496 Aug 23 06:21 1993 ./sbin/chown
-r-xr-xr-x bin/bin       53248 Aug 23 06:21 1993 ./sbin/mount_mfs
...  etc
```

cat pipes all archives in alphabetical file name order to *gunzip*. *gunzip* uncompresses it and pipes the uncompressed data to *tar*, which extracts the files.

Extracting a linked file

tar is clever enough to notice when it is backing up multiple copies of a file under different names, in other words so-called *hard links*. When backing up, the first time it encounters a file, it copies it to the archive, but if it encounters it again under another name, it simply creates an entry pointing to the first file. This saves space, but if you just try to extract the second file, *tar* will fail: in order to extract the second name, you also need to extract the file under the first name that *tar* found. Most versions of *tar* will tell you what the name was, but if you are creating archives, it helps to back up the most-used name first.

What's that archive?

All the preceding discussion assumes that you know the format of the archive. The fun begins when you don't. How do you extract it?

Your primary indication of the nature of the file is its filename. When archives are created, compressed and encoded, they usually receive a *file name suffix* to indicate the nature of the file. You may also have come across the term *extension*, which comes from the MS-DOS world. These suffixes accumulate as various steps proceed. A distribution of *gcc* might come in a file called *gcc-2.5.8.tar.gz.uue*. This name gives you the following information:

- The name of the package: *gcc*.
- The revision level: *-2.5.8*. You would expect the name of the root directory for this package to be *gcc-2.5.8*.
- The archive format: *.tar*. Since this is a GNU package, you can expect the name of the uncompressed archive to be *gcc-2.5.8.tar*.
- The compression format: *.gz* (*gzip* format). The name of the compressed archive would be *gcc-2.5.8.tar.gz*.
- The encoding format: *.uue* (encoded with *uuencode*).

Some operating systems, notably System V.3 and Linux, still provide file systems which restrict file names to 14 characters. This can lead to several problems.* Archives distributed for these systems frequently use variants on these names designed to make them shorter; *gcc-2.5.8.tzue* might be an alternate name for the same package.

The following table gives you an overview of archive file suffixes you might encounter. We'll look at source file suffixes in Chapter 20, *Compilers*, page

Table 2−1: Common file name suffixes

Name suffix	Format
#	Alternate *patch* reject file name.
~	*emacs* backup files, also used by some versions of *patch*.
,v	RCS file. Created by *ci*, extracted by *co*.
.a	*ar* format. Created by and extracted with *ar*.
.arc	Created by and extracted with *arc*.
.arj	DOS *arj* format
.cpio	Created by and extracted with *cpio*.
.diff	Difference file, created by *diff*, can be applied by *patch*.
.gif	Graphics Interchange Format
.gz	*gzip* format. Created by *gzip*, extracted with *gunzip*.
.hqx	HQX (Apple Macintosh)
.jpg	JPEG (graphics format)
.lzh	LHa, LHarc, Larc
.orig	Original file after processing by *patch*.
.rej	*patch* reject file.
.shar	Shell archive: created by *shar*, extracted with any Bourne-compatible shell.
.sit	Stuff-It (Apple Macintosh)
.tar	*tar* format. Created by and extracted with *tar*.
.uu	*uuencode*d file. Created by *uuencode*, decoded with *uudecode*.

* If you have one of these systems, and you have a choice of file systems, you can save yourself a lot of trouble by installing one that allows long file names.

Table 2-1: Common file name suffixes (continued)

Name suffix	Format
.uue	Alternative for *.uu*
.Z	Compressed with *compress*, uncompressed with *uncompress*, *zcat* or *gunzip*.
.z	Two different formats: either *pack* format, compressed by *pack*, extracted with *pcat*, or old *gzip* format, compressed by *gzip*, extracted with *gunzip*.
.zip	Zip (either PKZip or Zip/Unzip)
.zoo	Zoo

Identifying archives

Occasionally you'll get an archive whose name gives you no indication of the format. Under these circumstances, finding the kind of archive can be a matter of trial and error, particularly if it is compressed. Here are a couple of ideas that might help:

file

The UNIX *file* command recognizes a lot of standard file types and prints a brief description of the format. Unfortunately, the file really needs to be a file: *file* performs some file system checks, so it can't read from standard input. For example,

```
$ file *
0install.txt:                English text
base09.000:                gzip compressed data - deflate method ,   original
file name ,   last modified: Mon Aug 23 07:53:21 1993 ,   max compression os:
Unix
base09.001:                  data
...more of same
base09.011:                DOS executable (COM)
man-1.0.cpio:              cpio archive
tcl7.3.tar.gz:             empty
tex:                       directory
tk3.6.tar:                 POSIX tar archive
```

The information for *base09.000* was one output line that wrapped around onto 3 output lines.

Most files have certain special values, so-called *magic numbers*, in specific locations in their headers. *file* uses a file, usually */etc/magic*, which describes these formats. Occasionally it makes a mistake—we can be reasonably sure that the file *base09.011* is not a DOS executable, but it has the right number in the right place, and thus fools *file*.

This version of *file* (from BSD/OS) recognizes *base09.000*—and none of the following pieces of the archive—as a *gzip* archive file, and even extracts a lot of information. Not all versions of *file* do this. Frequently, it just tells you that the archive is data—in this case, the first assumption should be that the archive is compressed in a format that your version of *file* doesn't recognize. If the file is *pack*ed, *compress*ed or *gzip*ped, *gzip* expands it, and otherwise it prints an error message, so the next step might look something like:

```
$ gunzip < mystery > /tmp/junk
$                       aha! it didn't complain
$ file /tmp/junk
/tmp/junk:       POSIX tar archive
```

In this case, we have established that the file *mystery* is, in fact, a compressed *tar* archive, though we don't know what kind of compression, since *gzip* doesn't tell.

If *file* tells you that the file is ASCII or English text, then you can safely look at it with *more* or *less*:

```
$ more strange-file
Newsgroups: comp.sources.unix
From: clewis@ferret.ocunix.on.ca (Chris Lewis)
Subject: v26i014: psroff 3.0, Patch09
Sender: unix-sources-moderator@pa.dec.com
Approved: vixie@pa.dec.com

Submitted-By: clewis@ferret.ocunix.on.ca (Chris Lewis)
Posting-Number: Volume 26, Issue 14
Archive-Name: psroff3.0/patch9

    This is official patch 09 for Psroff 3.0.
...  intervening lines skipped
        clewis@ferret.ocunix.on.ca (Chris Lewis)


Patchwrapped: 920128230528

Index: ./lib/lj3.fonts
*** /tmp/PATCHold/./lib/lj3.fonts Tue Jan 28 23:03:45 1992
--- ./lib/lj3.fonts      Tue Jan 28 23:03:46 1992
```

This is a plain text patch file: you can pass it straight through the *patch* program, since *patch* doesn't worry about junk at the beginning or the end of the file. We'll look at *patch* in depth in Chapter 3, *Care and feeding of source trees*, page 30.

```
Newsgroups: comp.sources.unix From: lm@Sunburn.Stanford.EDU (Larry McVoy)
Subject: v26i020: perfmon - interface to rstatd(8)
Sender: unix-sources-moderator@pa.dec.com
Approved: vixie@pa.dec.com ...  more stuff omitted
#! /bin/sh
# This is a shell archive.  Remove anything before this line,
# then unpack it by saving it into a file and typing "sh file".
```

As the text tells you, this is a shell archive. To extract it, you can remove all text up to the line starting with *#!/bin/sh* and extract it with the Bourne shell, or pass it through *unshar* as it is.

```
begin 666 magic.gz
M'XL("'_!NRT''V5A<W1E<@!-4KV.VS',WO,4W'('N';:\9:B+3)T.*1HT*DH
M<+3$V+I(HB'*2?/V)14W=YMED-\OGW8HE0K0.#[![V/A!'4B<(M4_>1C>ZTS
MNW&$:<D5>!'J9_(0\@:@C?SJ#SU@]I'P7V'&4L6V=TOAF?Y'[N%C#U\@D0B.
M!%/PGK+NV[)A\/!*KH)C3[:',!<>"R9'T<<KGZC3Z4K9*VUE&'B.O"C?H&Q4
MA+,8C'^"(I2&&/((7&'H?!'[;JX4O0?X]$Y)!\HR3\%U.FT(TE#I>#0YE$*M
```

```
MU$C>%#UPT>&L?WY\ZQKNU^_['_S</S^N@1226061"15.!'K);DF4#4RHFD7'
M2;/R8BI/=)5:U*1TMG\W>C=O0PJF]N:(U[L45\B'*NIIGPDN%..'4^9+$T%8
MXA7>ZEWS"B;<\3+'%O3^0'(.%[%8)TK&<I/O6[6\!M>TPDM"U1+Y3%NXA#K!
M2^8*%RR?MZKA6:NWI5L?&&UM7I1>8,(S05K<!(D+'44<N&'E$R;OKD%#7!-P
M<?'66PQR.R73X>E,D0U_"QFUP@YFCJ$&IVST=^)2L0:-OH%(QNHF:MMI$>O8
I3#PH#VM<#H4>_]<O$)*>PYU)JPJE7>;*:>5!)4S]9O,/(PQ?IS4#''!I
'
end
```

This is a *uuencoded* file. The first line contains the word `begin`, the default security (which you can't change) and the name of the archive (*magic.gz*). The following lines usually have the same length and begin with the same letter (usually `M`)—this is the encoded length specification for the line. If they don't, something has probably gone wrong in the transmission. The last data line is usually shorter, and thus has a different first character. Finally, the archive contains two end lines: the first is usually the single character `'`, and the second is the word `end` on a line by itself.

To extract the file, first pass it through *uudecode*, which will create the file *magic.gz*, then *gunzip* it to create the file `magic`. *Then* you might need to use *file* to find out what it is.

```
$ uudecode < magic.uue
$ gunzip magic.gz
$ file magic
magic:  English text
```

Don't confuse *uuencode* format with this:

```
xbtoa5 78 puzzle.gz Begin
+,^C1(V%L;!!?e@F*(u6!)69ODSn.:h/s&KF-$KGlWA8mP,0BTe$'Y<$qSODDdUZO:_0iqn&P/S%8H
[AX_&!0:k0$N^5WjWlkG?U*XLRJ6"1S^E;mJ.k'Ea#$EL9q3*Bb.c9J@t/K/'N>62BM=7Ujbp7$YHN
,m"%IZ93t15j%OV"_S#NMI4;GC_N'=%+k5LX,A*uli>IBE@i0T4cP/A#coB""'a]![8jgS1L=p6Kit
X9EU5N%+(>-N=YU4(aeoGoFH9SqM6#c1(r;;K<'aBE/aZRX/^:.cbh&9[r.^f3bpQJQ&fW:*S_7DW9
6No0QkC7@A0?=YtSYlAc@01eeX;bF/9%&4E627AA6GR!u]3?Zhke.l4*T=U@TF9@1Gs4\jQPjbBm\H
K24N:$HKre7#7#jG"KFme^djs!<<*"N
xbtoa End N 331 14b E 5c S 75b7 R b506b514
```

This is a *btoa* encoded file, probably also *gzip*ped like the previous example. Extract it with *btoa -a* and then proceed as with *uuencode*d files.

What's in that archive?

Now you have discovered the format of the archive and can extract the files from it. There's a possibility, though, that you don't know what the archive is good for. This is frequently the case if you have a tape or a CD-ROM of an *ftp* server, and it contains some cryptic names that suggest the files might possibly be of interest. How do you find out what the package does?

README

By convention, many authors include a file *README* in the main directory of the package. *README* should tell you at least:

- The name of the package, and what it is intended to do.
- The conditions under which you may use it.

For example, the *README* file for GNU *termcap* reads:

```
This is the GNU termcap library -- a library of C functions that enable programs
to send control strings to terminals in a way independent of the terminal type.
Most of this package is also distributed with GNU Emacs, but it is available in
this separate distribution to make it easier to install as -ltermcap.

The GNU termcap library does not place an arbitrary limit on the size of termcap
entries, unlike most other termcap libraries.

See the file INSTALL for compilation and installation instructions.

Please report any bugs in this library to bug-gnu-emacs@prep.ai.mit.edu.  You
can check which version of the library you have by using the RCS 'ident' command
on libtermcap.a.
```

In some cases, however, there doesn't seem to be any file to tell you what the package does. Sometimes you may be lucky and find a good man page or even documentation intended to be printed as hardcopy—see Chapter 7, *Documentation* for more information. In many cases, though, you might be justified in deciding that the package is so badly documented that you give up.

There may also be files with names like *README.BSD*, *README.SYSV*, *README.X11* and such. If present, these will usually give specific advice to people using these platforms.

INSTALL file

There may be a separate *INSTALL* file, or the information it should contain might be included in the *README* file. It should tell you:

- A list of the platforms on which the package has been ported. This list may or may not include your system, but either way it should give you a first inkling of the effort that lies in store. If you're running System V.4, for example, and it has already been ported to your hardware platform running System V.3, then it should be easy. If it has been ported to V.4, and you're running V.3, this can be a completely different matter.
- A description of how to configure the package (we'll look at this in Chapter 4, *Package configuration*).
- A description of how to build the package (see Chapter 4, *Package configuration* and Chapter 19, *Make* for more details on this subject).

It may, in addition, offer suggestions on how to port to other platforms or architectures.

Other files

The package may include other information files as well. By convention, the names are written in upper case or with an initial capital letter, so that they will be stand out in a directory listing. The GNU project software may include some or all of the following files:

- *ABOUT* is an alternative name used instead of *README* by some authors.
- *COPYING* and *COPYING.LIB* are legal texts describing the constraints under which you may use the software.
- *ChangeLog* is a list of changes to the software. This name is hard-coded into the *emacs* editor macros, so it's a good chance that a file with this name will really be an emacs-style change log.
- *MANIFEST* may give you a list of the files intended to be in the package.
- *PROBLEMS* may help you if you run into problems.
- *SERVICE* is supplied by the Free Software Foundation to point you to companies and individuals who can help you if you run into trouble.

A good example of these files is the root directory of Taylor *uucp*:

```
$ gunzip </cd0/gnu/uucp/uucp-1.05.tar.gz |tar tvf -
drwxrwxr-x 269/15           0 May  6 06:10 1994 uucp-1.05/
-r--r--r-- 269/15       17976 May  6 05:23 1994 uucp-1.05/COPYING
-r--r--r-- 269/15      163997 May  6 05:24 1994 uucp-1.05/ChangeLog
^C$
```

This archive adheres to the GNU convention of including the name of the top-level directory in the archive. When we extract the archive, *tar* will create a new directory *uucp-1.05* and put all the files in it. So we continue:

```
$ cd /porting/src  the directory in which I do my porting
$ gunzip </cd0/gnu/uucp/uucp-1.05.tar.gz |tar xf -
$
```

After extraction, the resultant directory contains most of the "standard" files that we discussed above:

```
$ cd uucp-1.05
$ ls -l
  total 1724
  drwxrwxr-x   7 grog     wheel        1536 May  6 06:10 .
  drwxrwxrwx  44 grog     wheel        3584 Aug 19 14:34 ..
  -r--r--r--   1 grog     wheel       17976 May  6 05:23 COPYING
  -r--r--r--   1 grog     wheel      163997 May  6 05:24 ChangeLog
  -r--r--r--   1 grog     wheel         499 May  6 05:24 MANIFEST
  -rw-r--r--   1 grog     wheel       14452 May  6 06:09 Makefile.in
  -r--r--r--   1 grog     wheel        4283 May  6 05:24 NEWS
  -r--r--r--   1 grog     wheel        7744 May  6 05:24 README
  -r--r--r--   1 grog     wheel       23563 May  6 05:24 TODO
  -r--r--r--   1 grog     wheel       32866 May  6 05:24 chat.c
```

```
  -r--r--r--   1 grog     wheel       19032 May  6 05:24 config.h.in
  -rwxrwxr-x   1 grog     wheel       87203 May  6 05:27 configure
  -r--r--r--   1 grog     wheel       11359 May  6 05:24 configure.in
...etc
```