# The Answer to All Man's Problems

*Tom Christiansen* – Convex Computer Corporatioin


## ABSTRACT

The UNIX on-line manual system was designed many years ago to suit the needs of systems at the time, but despite the growth in complexity of typical systems and the need for more sophisticated software, few modifications have been made to it since then. This paper presents the results of a complete rewrite of the man system. The three principal goals were to effect substantial gains in functionality, extensibility, and speed. The secondary goal was to rewrite a basic UNIX utility in the perl programming language to observe how perl affected development time, execution time, and design decisions.

Extensions to the original man system include storing the whatis database in DBM format for quicker access, intelligent handling of entries with multiple names (via **.so** inclusion, links, or the NAME section), embedded tbl and eqn directives, multiple man trees, extensible section naming possibilities, user-definable section and sub-section search ordering, an indexing mechanism for long man pages, typesetting of man pages, text-previewer support for bit mapped displays, automatic validity checks on the SEE ALSO sections, support for compressed man pages to conserve disk usage, per-tree man macro definitions, and support for man pages for multiple architectures or software versions from the same host.


## Introduction

The UNIX on-line manual system was designed many years ago to suit the needs of the systems at the time. Since then, despite the growth in complexity of typical systems and the need for more sophisticated software to support them, few modifications of major significance have been made to it. This paper describes problems inherent in earlier versions of the *man* program, proposes solutions to these problems, and outlines one implementation of these solutions.

## The Problem

### The Monolithic Approach

One of the most serious problems with the *man* program through the BSD4.2 release was that all man pages on the entire system were expected to reside under a common directory, */usr/man*. There was no notion of separate sets of man pages installed on the same machine in different subdirectories. At large installations, this functionality is desirable. A site may wish to keep vendor-supplied man pages separate from man pages that were developed locally or acquired from some third party. An individual or group may wish to maintain their own set of man pages. Multiple versions of the same software package might be simultaneously installed on the same machine. A heterogeneous environment may want to be able to view man pages for all available architectures from any machine. Given the requirement that all man pages live in the same directory, these scenarios are difficult to impossible to support.

The *man* program distributed in the BSD4.3 release included the concept of a MANPATH, a colon-delimited list of complete man trees taken either from the user's environment or supplied on the command line. While this was a vast improvement over the previous monolithic approach, several significant problems remained. For one thing, the program still had to use the *access*(2) system call on all possible paths to find out where the man page for a particular topic existed. When the user has a MANPATH containing multiple components, the time needed for the *man* program to locate a man page is often noticeable, particularly when the target man page does not exist.

### Hard-coded Section Names

Another problem with the *man* program unresolved by the BSD4.3 release was that all possible sections in which a man page could reside were hard-coded into the program. This means that while a **manp** section would be recognized, a **manq** directory would not be, and while a **man3f** directory would be recognized, a **man3x11** directory would not be.

Likewise, the possible subsections for a man page were also embedded in the source code, so a man page named something like

*/usr/man/man3/XmLabel.3x11*

would not be found because **3x11** was not in the hard-coded list of viable subsections. Some systems install all man pages stripped of subsection components in the file name. This situation is less than optimal because it proves useful to be able to supply both a *getc*(3f) and a *getc*(3s). Distinguishing between subsections is particularly convenient with the "intro" man pages; a vendor could supply *intro*(3) *intro*(3a), *intro*(3c), *intro*(3f), *intro*(3m), *intro*(3n), *intro*(3r), *intro*(3s), and *intro*(3x) as introductory man pages for the various libraries. However, the task of running *access*(2) on all possible subsections is slow and tedious, requiring recompilation whenever a new subsection is invented.

## References in the Filesystem

The existing man system had no elegant way to handle man pages containing more than one entry. For example, *string* (3) contains references to *strcat* (3) and *strcpy* (3), amongst others. Because the *man* program looks for entries only in the file system, these extra references must be represented as files that reference the base man page. The most common practice is to have a file consisting of a single line telling *troff* to source the other man page. This file would read something like:

```
.so man3/string.3
```

Occasionally, extra references are created with a link in the file system (either a hard link or a symbolic one). Except when using hard links, this method wastes disk blocks and inodes. In any case, the directory gains more entries, slowing down accesses to files in those directories. Logic must be built into the *man* program to detect these extra references. If not, when man pages are reformatted into their cat directories, separate formatted man pages are stored on disk, wasting substantial amounts of disk space on duplicate information. On systems with numerous man pages, the directories can grow so large that all man pages for a given section cannot be listed on the command line at one time because of kernel restrictions on the total length of the arguments to *exec* (2). Because of the need to store reference information in the file system, the problem is only made worse. This often happens in section 3 after the man pages for the X library have been installed, but can occur in other sections as well.

The *makewhatis* (8) program is a Bourne shell script that generates the */usr/lib/whatis* index, and is used by *apropos* (1) and *whatis* (1) to provide one-line summaries of man pages. These programs are part of the *man* system and are often links to each other and sometimes to *man* itself. If any of the man subdirectories contain more files than the shell can successfully expand on the command line, the *makewhatis* script fails and no index is generated. When this occurs, *whatis* and *apropos* stop working. The *catman* (8) program, used to pre-format raw man pages, suffers from the same problem.

Of course, *makewhatis* wasn't working all that well, anyway. It was a wrapper around many calls to little programs that each did a small piece of the work, making it run slowly. It, too, had a hard-coded pathname for where man pages resided and which sections were permitted. *Makewhatis* didn't always extract the proper information from the man page's NAME section. When it did, this information was sometimes garbled due to embedded *troff* formatting information. But even garbled information was better than none at all. Even so, these programs left some things to be desired. *Apropos* didn't understand regular expression searches, and both it and *whatis* preferred to do their own lookups using basic, unoptimized C functions like *index* (3) rather than using a general-purpose optimized string search program like *egrep* (1).

## The Solution

### A Real Database

The problem in all these cases appeared to be that the filesystem was being used as a database, and that this paradigm did not hold up well to expansion. Therefore the solution was to move this information into a database for more rapid access. Using this database, *man* and *whatis* need no longer call *access* (2) to test all possible locations for the desired man page. To solve the other problems, *makewhatis* (8) would be recoded so it didn't rely on the shell for looking at directories.

### Coding in Perl

When the project was first contemplated, the perl programming language by Larry Wall was rapidly gaining popularity as an alternative to C for tasks that were either too slow when written as shell scripts, or simply exceeded the shells' somewhat limited capabilities. Since perl was optimized for parsing text, had convenient *dbm* (3x) support built in to it, and the task really didn't seem complex enough to merit a full-blown treatment in C or C++, perl was selected as the language of choice. Having all code written in perl would also help support heterogeneous environments because the resulting scripts could be copied and run on any hardware or software platform supporting perl. No recompilation would be required.

Some concern existed about choosing an interpreted language when one of the issues to address was that of speed. It was decided to do the prototype in perl and, if necessary, translate this into C should performance prove unacceptable.

The first task was to recode *makewhatis* (8) to generate the new *whatis* database using *dbm*. The *directory* (3) routines were used rather than shell globbing to circumvent the problem of large directories breaking shell wildcard expansions. Perl proved to be an appropriate choice for this type of text processing (see Figure 1).

### Database Format

The database entries themselves are conveniently accessed as arrays from perl. To save space and accommodate man pages with multiple references, two kinds of database entries exist: direct and indirect. Indirect entries are simply references to direct entries. For example, indirect entries for *getc* (3s), *getchar* (3s), *fgetc* (3s), and *getw* (3s) all point to the real entry, which is *getc* (3s). Indirect entries are created for multiple entries in the NAME section, for symbolic and hard links, and for **.so** references. Using the NAME section is the preferred method; the others are supported for backwards compatibility.

Assuming that the WHATIS array has been bound to the appropriate *dbm* file, storing indirect entries is trivial:

```
$WHATIS{'fgetc'} = 'getc.3s';
```

When a program encounters an indirect entry, such as for *fgetc*, it must make another lookup based on the return value of first lookup (stripped of its trailing extension) until it finds a direct entry. The trailing extension is kept so that an indirect reference to *gtty* (3c) doesn't accidentally pull out *stty* (1) when it really wanted *stty* (3c).

The format of a direct entry is more complicated, because it needs to encode the description to be used by *whatis* (1) as well as the section and subsection information. It can be distinguished from an indirect entry because it contains four fields delimited by control-A's (ASCII 001), which are themselves prohibited from being in any of the fields. The fields are as follows:

1. List of references that point to this man page; this is usually everything to the left of the hyphen in the NAME section.
2. Relative pathname of the file the man page is kept in; this is stored for the indirect entries.
3. Trailing component of the directory in which the man page can be found, such as **3** for **man3**.
4. Description of the man page for use by the *whatis* and *apropos* programs; basically everything to the right of the hyphen in the NAME section.

At first glance, the third field would seem redundant. It would appear that you could derive it from the character after the dot in the second field. However, to support arbitrary subdirectories like **man3f** or **man3x11**, you must also know the name of the directory so you don't look in **man3** instead. Additionally, a long-standing tradition exists of using the **mano** section to store old man pages from arbitrary sections. Furthermore, man pages are sometimes installed in the wrong section. To support these scenarios, restrictions regarding the format of filenames used for man pages were relaxed in *man*, *makewhatis*, and *catman*, but warnings would be issued by *makewhatis* for man pages installed in directories that don't have the same suffix as the man pages.

### Multiple References to the Same Topic

A problem arises from the fact that the same topic may exist in more than one section of the manual. When a lookup is performed on a topic, you want to retrieve all possible man page locations for that topic. The *whatis* program wants to display them all to the user, while the *man* program will either show all the man pages (if the **−a** flag is given) or sort what it has retrieved according to a particular section and subsection precedence, by default showing entries from section 1 before those from section 2, and so forth. Therefore, each lookup may actually return a list of direct and indirect lookups. This list is delimited by control-B's (ASCII 002), which are stripped from the data fields, should they somehow contain any. The code for storing a direct entry in the *whatis* database is featured in Figure 2.

Notice the check the new datum's length against the value of MAXDATUM. This is because of the inherent limitations in the implementation of the *dbm* (3x) routines. This is 1k for *dbm* and 4k for *ndbm*. This restriction will be relaxed if a *dbm*-compatible set of routines is written without these size limitations. The GNU *gdbm* routines hold promise,

```
s/\\f([PBIR]|\(..)//g;       # kill font changes
s/\\s[+-]?\d+//g;            # kill point changes
s/\\&//g;                    # and \&
s/\\\((ru|ul)/_/g;           # xlate to '_'
s/\\\((mi|hy|em)/-/g;        # xlate to '-'
s/\\\*\(..//g  &&            # no troff strings
    print STDERR "trimmed troff string macro in NAME section of $FILE\n";
s/\\//g;                     # kill all remaining backslashes
s/^\.\\"\s*//;               # kill comments
if (!/\s+-+\s+/) {
    #   ^ otherwise L-devices would be L
    print STDERR "$FILE: no separated dash in $_\n";
    $needcmdlist = 1;        # forgive their braindamage
    s/.*-//;
    $desc = $_;
} else {
    ($cmdlist, $desc) = ( $`, $' );
    $cmdlist =~ s/^\s+//;
}
```

**Figure 1**: *makewhatis* excerpt #1

but they were released after the writing of these programs and haven't been investigated yet. In practice, these limits are seldom if ever reached, especially when *ndbm* is used.

### Other Problems, Other Solutions

The rewrite of *makewhatis*, *catman*, and *man* to understand multiple man trees and to use a database for topic-to-pathname mapping did much to alleviate the most important problems in the existing man system, but several minor problems remained. Since this was a complete rewrite of the entire system, it seemed an appropriate time to address these as well.

#### Indexing Long Pages

Several of the most frequently consulted man pages on the system have grown beyond the scope of a quick reference guide, instead filling the function of a detailed user manual. Man pages of this sort include those for shells, window managers, general purpose utilities such as awk and perl, and the X11 man pages. Although these man pages are internally organized into sections and subsections that are easily visible on a hard-copy printout, the on-line man system could not recognize these internal sections. Instead, the user was forced to search through pages of output looking for the section of the man page containing the desired information.

To alleviate this time-consuming tedium, the man program was taught to parse the *nroff* source for man pages in order to build up an index of these sections and present them to the user on demand. See Figure 3 for an excerpt from the *ksh*(1) index page, displayable via the new **–i** switch.

The */usr/man/idx*/* directories serve the same function for saved indices as */usr/man/cat*/* directories do for saved formatted man pages. These are regenerated as needed according the the same criteria used to regenerate the cat pages. They can be used to index into a given man page or to list a man page's subsections. To begin at a given subsection, the user appends the desired subsection to the name of the man page on the command line, using a forward slash as a delimiter. Alternatively, the user can just supply a trailing slash on the man page name, in which case they are presented with the index listing like the one the **–i** switch provides, then prompted for the section in which they are interested. A double slash indicates an arbitrary regular expression, not a section name. This is merely a short-hand notation for first running man and then typing `/expr` from within the user's pager. See Figure 4 for example usages of the indexing features.

```
Idx  Subsections in ksh.1    Lines
 1   NAME                        3
 2   SYNOPSIS                   22
 3   DESCRIPTION                15
 4   Definitions.               43
 5   Commands.                 338
 6   Comments.                   6
 7   Aliasing.                 107
 8   Tilde Substitution.        47
 9   Command Substitution.      28
10   Process Substitution.      49
11   Parameter Substitution.   645
12   Blank Interpretation.      15
13   File Name Generation.      87
```

**Figure 3**: *ksh* index excerpt

This indexing scheme is implemented by searching the index stored in */usr/man/idx1/ksh.1* if it exists, or generated dynamically otherwise, for the requested subsection. A numeric subsection is easily handled. For strings, a case-insensitive pattern match is first made anchored to the front of the string, then — failing that — anywhere in the section description. This way the user doesn't need to type the full section title. The *man* program starts up the pager with a leading argument to begin at that section. Both *more*(1) and

```
sub store_direct {
    local($cmd, $list, $page, $section, $desc) = @_; # args
    local($datum);

    $datum = join("\001", $list, $page, $section, $desc);

    if (defined $WHATIS{$cmd}) {
        if (length($WHATIS{$cmd}) + length($datum) + 1 > $MAXDATUM) {
            print STDERR "can't store $page -- would break DBM\n";
            return;
        }
        $WHATIS{$cmd} .= "\002";  # append separator
    }
    $WHATIS{$cmd} .= $datum;  # append entry
}
```

**Figure 2**: *makewhatis* excerpt #2

```
man -i ksh       # show sections
man ksh/         # show sections,
                 # prompt for which one
man ksh/tilde
man ksh/8        # equivalent to
                 # preceding line
man ksh/file
man ksh/generat # == preceding line
man ksh/13       # so is this

man ksh//hangup # start at this string
```

**Figure 4**: Index Examples

*less* (1) understand this particular notation. In the first example given above, this would be

```
less '+/^[ \t]*Tilde Substitution'\
                 /usr/man/cat1/ksh.1
```

Once again, perl proved useful for coding this algorithm concisely. The subroutine for doing this is given in Figure 5. Given an expression such as "5" or "tilde" or "file" and a pathname of the man page, *man* loads an array of subsection index titles and quickly retrieves the proper header to pass on to the pager. Perl's built-in **grep** routine for selecting from arrays those elements conforming to certain criteria made the coding easy.

### Conditional Tbl and Eqn Inclusion

Several other relatively minor enhancements were made to the man system in the course of its rewrite. One of these was to include calls to *eqn* (1) and *tbl* (1) where appropriate. For instance, the X11 man pages use *tbl* directives to construct a number of tables. It was not sufficient to supply these extra filters for all man pages. Besides the slight performance degradation this would incur, a more serious problem exists: some systems have man pages that contain embedded **.TS** and **.TE** directives; however, the data between them was not *tbl* input, but rather its output. They have already been pre-processed in the unformatted versions. To do so again causes *tbl* to complain bitterly, so heuristics to check for this condition were built in to the function that determines which filters are needed.

To support tables and equations in man pages when viewed on-line, the output must be run through *col* (1) to be legible. Unfortunately, this strips the man pages of any bold font changes, which is undesirable because it is often important to distinguish between bold and italics for clarity. Therefore, before the formatted man page is fed to *col*, all text in bold (between escape sequences) is converted to character-backspace-character combinations. These combinations can be recognized by the user's pager as a character in a bold font, just as underbar-backspace-character is recognized as an italic (or underlined) one. Unfortunately, while *less* does recognize this convention, *more* does not. By storing the formatted versions with all escape-sequences removed, the user's pager can be invoked without a pipe to *ul* or *col* to fix the reverse line motion directives. This provides the pager with a handle on the pathname of the cat page, allowing users to back up to the start of man pages, even exceptionally long ones, without exiting the *man* program. This would not be feasible if the pager were being fed from a pipe.

### Troffing and Previewing Man Pages

Now that many sites have high-quality laser printers and bit-mapped displays, it seemed desirable for *man* to understand how to direct *troff* output to these. A new option, **-t**, was added to mean that *troff* should be used instead of *nroff*. This way users can easily get pretty-printed versions of their man pages.

```
sub find_index {
    local($expr, $path) = @_;  # subroutine args
    local(@matches, @ssindex);
    @ssindex = &load_index($path);

    if ($expr > 0) {            # test for numeric section
        return $ssindex[$expr];
    } else {
        if (@matches = grep (/^$expr/i, @ssindex)) {
            return $matches[0];
        } elsif (@matches = grep (/$expr/i, @ssindex)) {
            return $matches[0];
        } else {
            return '';
        }
    }
}
```

**Figure 5**: Locate Subsection by Index

For workstation or X-terminal users, *man* will recognize a TROFF environment variable or command line argument to indicate an alternate program to use for typesetting. (This presumes that the program recognizes *troff* options.) This method often produces more legible output than *nroff* would, allows the user to stay in their office, and saves trees as well.

### Section Ordering

The same topic can occur in more than one section of the manual, but not all users on the system want the same default section ordering that *man* uses to sort these possible pages. For instance, C programmers who want to look up the man page for *sleep*(3) or *stty*(3) find that by default, *man* gives them *sleep*(1) and *stty*(1) instead. A FORTRAN programmer may want to see *system*(3f), but instead gets *system*(3). To accommodate these needs, the *man* program will honor a MANSECT environment variable (or a **–S** command line switch) containing a list of section suffixes. If subsection or multi-character section ordering is desired, this string should be colon-delimited. The default ordering is "ln16823457po". A C programmer might set his MANSECT to be "231" instead to access subroutines and system calls before commands of the same name. A FORTRAN programmer might prefer "3f:2:3:1" to get at the FORTRAN versions of subroutines before the standard C versions. Sections absent from the MANSECT have a sorting priority lower than any that are present.

### Compressed Man Pages

Because man pages are ASCII text files, they stand to benefit from being run through the *compress*(1) program. Compressing man pages typically yields disk space savings of around 60%. The start-up time for decompressing the man page when viewing is not enough to be bothersome. However, running *makewhatis* across compressed man pages takes significantly longer than running it over uncompressed ones, so some sites may wish to keep only the formatted pages compressed, not the unformatted ones.

Two different ways of indicating compressed man pages seem to exist today. One is where the man page itself has an attached **.Z** suffix, yielding pathnames like */usr/man/man1/who.1.Z.* The other way is to have the section directory contain the **.Z** suffix and have the files named normally, as in */usr/man/man1.Z/who.1*. Either strategy is supported to ease porting the program to other systems. All programs dealing with man pages have been updated to understand man pages stored in compressed form.

### Automated Consistency Checking

After receiving a half-dozen or so bug reports regarding non-existent man pages referenced in SEE ALSO sections, it became apparent that the only way to verify that all bugs of this nature had really been expurgated would be to automate the process. The *cfman* program was verifies that man pages are mutually consistent in their SEE ALSO references. It also reports man pages whose **.TH** line claims the man page is in a different place than *cfman* found it. *Cfman* can locate man pages that are improperly referenced rather than merely missing. It can be run on an entire man tree, or on individual files as an aid to developers writing new man pages.

```
at.1: cron(8) really in cron(1)
binmail.1: xsend(1) missing
dbadd.1: dbm(3) really in dbm(3x)
ksh.1: exec(2) missing
ksh.1: signal(2) missing
ksh.1: ulimit(2) missing
ksh.1: rand(3) really in rand(3c)
ksh.1: profile(5) missing
ld.1: fc(1) really in fc(1f)
sccstorcs.1: thinks it's in ci(1)
uuencode.1c: atob(n) missing
yppasswd.1: mkpasswd(5) missing
fstream.3: thinks it's in fstream(3c++)
ftpd.8c: syslog(8) missing
nfmail.8: delivermail(8) missing
versatec.8: vpr(1) missing
```

**Figure 6**: Sample *cfman* run

The amount of output produced by *cfman* is startling. A portion of the output of a sample run is seen in Figure 6. Some of its complaints are relatively harmless, such as *dbm* being in section **3x** rather than section **3**, because the *man* program can find entries with the subsection left off. Having inconsistent **.TH** headers is also harmless, although the printed man pages will have headers that do not reflect their filenames on the disk. However, entries that refer to pages that are truly absent, like *exec*(2) or *delivermail*(8), merit closer attention.

### Multiple Architecture Support

As mentioned in the discussion of the need for a MANPATH, a site may for various reasons wish to maintain several complete sets of man pages on the same machine. Of course, a user could know to specify the full pathname of the alternate tree on the command line or set up their environment appropriately, but this is inconvenient. Instead, it is preferable to specify the machine type on the command line and let the system worry about pathnames. Consider these examples:

```
man vax csh
apropos sun rpc
whatis tahoe man
```

To implement this, when presented with more than one argument, *man* (in any of its three guises) checks to see whether the first non-switch argument is a directory beneath */usr/man*. If so, it automatically adjusts its MANPATH to that subdirectory.

Not all vendors use precisely the same set of *man*(7) macros for formatting their man pages.

Furthermore, it's helpful to see in the header of the man page which manual it came from. The *man* program therefore looks for a local *tmac.an* file in the root of the current man tree for alternate macro definitions. If this file exists, it will be used rather than the system defaults for passing to *nroff* or *troff* when reformatting.

### Performance Analysis

The *man* program is one that is often used on the system, so users are sensitive to any significant degradation in response time. Because it is written in perl (an interpreted language) this was cause for concern. On a CONVEX C2, the C version runs faster when only one element is present in the MANPATH. However, when the MANPATH contains four elements, the C version bogs down considerably because of the large number of *access* (2) calls it must make.

The start-up time on the parsing of the script, now just over 1300 lines long, is around 0.6 seconds. This time can be reduced by dumping the parse tree that perl generates to disk and executing that instead. The expense of this action is disk space, as the current implementation requires that the whole perl interpreter be included in the new executable, not just the parse tree. This method yields performance superior to that of the C version, irrespective of the number of components in the user's MANPATH, except occasionally on the initial run. This is because the program needs to be loaded into memory the first time. If perl itself is installed "sticky" so it is memory resident, start-up time improves considerably. In any case, the total variance (on a CONVEX) is less than two seconds in the worst case (and often under one second), so it was deemed acceptable, particularly considering the additional functionality the perl version offers.

Nothing in the algorithms employed in the *man* program require that it be written in perl; it was just easier this way. It could be rewritten in C using *dbm* (3x) routines, although the development time would probably be much longer.

The *makewhatis* program was originally a conglomeration of man calls to various individual utilities such as *sed*, *expand*, *sort*, and others. The perl rewrite runs in less than half the time of the original, and does a much better job. There are two reasons for the speed increase. The first is the cost of the numerous *exec* (2) calls made via the shell script used by the old version of *makewhatis*. The second is that perl is optimized for text processing, which is most of what *makewhatis* is doing.

Total development time was only a few weeks, which was much shorter than originally anticipated. The short development cycle was chiefly attributable to the ease of text processing in perl, the many built-in routines for doing things that in C would have required extensive library development, and, last but not at all least, the omission of the compilation stage in the normal edit-compile-test cycle of development when working with non-interpreted languages.

### Conclusions

The system described above has been in operation for the last six months on a large local network consisting of three dozen CONVEX machines, a token VAX, quite a few HP workstations and servers, and innumerable Sun workstations, all running different flavors of UNIX. Despite this heterogeneity, the same code runs on all systems without alterations. Few problems have been seen, and those that did arise were quickly fixed in the scripts, which could be immediately redistributed to the network. The principal project goals of improved functionality, extensibility, and execution time were adequately met, and the experience of rewriting a set of standard UNIX utilities in perl was an educational one. Man pages stand a much better chance of being internally consistent with each other. Response from the user and development community has been favorable. They have been relieved by the many bug fixes and pleasantly surprised by the new functionality. The suite of man programs will replace the old man system in the next release of CONVEX utilities.

### Author Information

Tom Christiansen left the University of Wisconsin with an MS-CS in 1987 where he had been a system administrator for 6 years to join CONVEX Computer Corporation in Richardson, Texas. He is a software development engineer in the Internal Tools Group there, designing software tools to streamline software development and systems administration and to improve overall system security. Reach him via U. S. Mail at CONVEX Computer Corporation; POB 833851; 3000 Waterview Parkway; Richardson, TX 75083-3851. Reach him electronically at uunet!convex!tchrist or tchrist@convex.com .