Two Years with TkMan: Lessons and Innovations
Or, Everything I Needed to Know about Tcl/Tk I Learned from TkMan

Thomas A. Phelps
Computer Science Division
University of California, Berkeley

Two Years with TkMan: Lessons and Innovations
Or, Everything I Needed to Know about Tcl/Tk I Learned from TkMan

Thomas A. Phelps
Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley
`phelps@CS.Berkeley.EDU`

Abstract

Among Tcl/Tk applications, TkMan is unusual. Whereas Tcl was written to glue together C functions, TkMan is written entirely in Tcl. And TkMan is the beneficiary of years of battle testing by 1000s of users on every flavor of UNIX. The extreme position created by the demands of this large, diverse audience and the (self-imposed) limitation of remaining strictly within Tcl brought to the fore a severe set of implementation issues, and provoked a variety of solutions ranging from general methods to a low-level bag of tricks with regard to speeding up Tcl scripts, exploiting Tcl as its own scripting language, configuring applications and interoperating with other tools. Although developed to meet these particular, extreme requirements, most of the resulting solutions can be broadly applied, and this paper shares this lore with the aim of helping other authors develop elegant, efficient and robust Tcl/Tk-based applications.

The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

—F. Brooks, *The Mythical Man Month*

1. Introduction

TkMan is a graphical, hypertext browser for UNIX's online documentation, the manual pages. It was borne of frustration with `xman` [Shei]—why aren't words in the SEE ALSO section hypertext links? Such a feature was straightforward to implement with Tk's text widget, and a host of other dissatisfactions were satiated along the way to a full-featured system.

In the world of applications written with Tcl and Tk, TkMan is unusual. Although Tcl is meant to glue together selected routines written in C that have exposed a protocol-compliant interface, TkMan is written entirely in Tcl, apart from an external filter. Why doesn't TkMan employ C? The original version worked reasonably well without C (users combatted the lengthy startup time by invoking TkMan at login and keeping it running throughout). More importantly for an author who doubles as tech support, a script-only implementation eased installation: there was no patching of the Tcl core, no worries about ANSI vs. K&R C compilers, no need to create a large binary to add only a few functions to the basic functionality provided by the wish windowing shell, and the same copy of the TkMan code can be shared across machine architectures. As the code was developed, urges to take the plunge into C were resisted as no isolated feature was significant enough to outweigh the advantages of Tcl-only, and reasons were not allowed to accumulate as solutions were continuously devised to mitigate the bottlenecks.

Since the first version of TkMan was posted to the `comp.lang.tcl` newsgroup on April 1, 1993, the size of the distribution has grown by an order of magnitude, from 38K to 340K bytes (1373 to 7950 lines). Version 1.5 of the software has been `ftp`'ed by almost 1400 people from the Berkeley home site alone (many of them system administrators, which suggests a larger user population), and by unknown numbers of others from the main Tcl archive and geographically distributed sources in Germany, Japan and elsewhere.

During the main development period from its release until November 1993, a continuous flow of bug reports and feature requests peppered my mailbox. Except for one major improvement described below, each bug was simple enough to squash, each request straightforward enough to appease, each port just one more case to check for (periodically, enough similar patches would accumulate to suggest a general simplification).

Throughout development, a key goal was to reduce time spent dealing with individual problems, which translated into a focus on simplifying installation and making the code more robust and dynamically adaptable to the variety of UNIX environments. If we judge by the subsequent great reduction in bug- and feature-related e-mail, this work produced code that is robust, portable, and feature-rich.

This paper shares lessons learned and solutions devised during the implementation of TkMan. The following sections describe general solutions and various low-level tricks that address various categories of problems. The restriction to Tcl-only provoked a variety of strategies to achieve acceptable speed in the face of a considerable amount of string processing. But the interpreted aspect of Tcl paid off in other ways, as Tcl could be used as its own scripting language, which in one case supplied a nice solution to a portability crisis.

As mentioned, the large user population promoted work on simplifying installation and making the code robust. A significant percentage of this user population used another graphical man page browser, `xman`, whereas the rest used a variety of text-based viewers. Satisfying the expectations of both groups of users led to new insights into the requirements for the interoperability of text with graphical, and graphical with graphical tools.

2. Speeding Up Scripts

With earlier generations of compiler technology, if the C code was too slow and more efficient algorithms were not obvious, one considered coding the time-critical pieces in assembly language. Similarly today, if the interpreted Tcl code is too slow, the standard advice is to code that portion in C and expose an interface back to the Tcl level.

For reasons already described, this option was resisted in TkMan. Moreover, sometimes the backdoor to C does not relieve the bottleneck. One such case arises when there is considerable state to communicate from Tcl to C and then back to Tcl: in moving to C there may be a great deal of string copying and type conversion; and in returning to Tcl, there are the inverse data conversions and possibly numerous variable settings.

Herewith are suggestions for speeding up scripts, principles made concrete with specific references to experiences in building TkMan.

2.1 Exploit External Processes

Character-by-character processing

Usually manual pages are distributed as `[tn]roff` source code and formatted by `nroff` for viewing on the screen. Since formatting takes a noticable amount of time, even on current workstations, formatted versions are cached on disk. Betraying the days when man pages were printed on line printers, formatted man pages simulate boldfacing with *character*-`backspace`-*character* overstriking and italics with *character*-`backspace-underscore`. Clearly this format is unsuitable for modern documentation systems, which must step through the text character by character to reconstruct bold and italics passages. In addition, most man page macros still format the page for printing as pages, inserting page headers and footers that are unwanted when the information is viewed online.

In the initial development of TkMan, Tcl code was written to perform this character-level analysis of the text. Deadly slow. With all the string copying that `string index` needed for *every* character, even pages that occupied only a screenful or two took *minutes* to format.

The solution? Code page analysis in C, call as an external process, and make it produce valid Tcl commands that can simply be `eval`'ed. The control over filtering parameters TkMan wants to exercise is passed along in command line options.

If dividing code between the Tcl level and the C level promotes discipline in architecture design, dividing it between a script and an external process enforces absolute hygiene. This paid off when the filter was extended to also generate other types of text source markups (including HTML, LaTeX and even [tn]roff source), as there was no need to extract the relevant code from a tangle of TkMan support: The filter was already isolated and hence immediately available for use in a variety of other environments.

Large-scale string searching and filtering

The other chief performance concern involved searching for requested manual page names from those available in the system. Most text-based man pagers dynamically scour the contents of the directory hierarchies along every component of the user's MANPATH search path, a process which can involve several hundred directories and many thousands of file names. This causes a noticable delay, especially over remotely-mounted file systems.

TkMan has always built a database of valid manual page names to speed the search. There are two potential bottlenecks in using the database: in its construction and in its use during searches. A number of database designs were implemented before arriving at one that performs acceptably at both points.

Until the current version, the database was constructed at startup time and stored in memory (as list variables) where it was thought searching would be fast. This design suffered a number of maladies. Startup took a nontrivial amount of time, minutes on slower machines. Although the total size in characters of the text names was typically in the low 100K's bytes, the memory image ballooned to more than a megabyte—on top of a megabyte for `wish` and more for the application code (on a 32-bit RISC machine). And searching was not all that quick. The built-in list search command was not suitable because it returns only the first match, whereas a correct result may include multiple elements from a given list, all of which are desireable. Thus each list had to be iterated through and examined element by element. One could use Tcl's associative arrays as a hash table keyed on the page name for instantaneous search, but the list representation was needed to search by patterns and to construct lists of all pages in various “volumes” like User Commands and File Formats.

In an attempt to reduce startup time, the database was cached on disk and only rebuilt when invalidated due to changes in the man page directories. Compressed, this file consumed only 50K bytes. However, this strategy saved almost no time at startup. The time needed to read directory contents paled next to the time needed by Tcl to parse the results, allocate memory, and store the information. Since the cached database improved nothing and cost disk space, it was abandoned for the original approach.

The current version of TkMan, 1.7, is successful in giving both quick startup and quick searches. It caches the database on disk as described immediately above, but spawns the standard UNIX searching and filtering utilities `grep` and `sed` to search for man page names. It was surprising to discover that it is faster to read the data from disk and spawn three processes to decompress, filter and search it, than it is to loop through the equivalent lists in Tcl.

In short, the lesson of this section is that if a Tcl script runs too slowly at some point, an alternative to implementing that portion as a C funtion in the executable is to spawn processes that do run fast for the heavy processing, and then collect the results in Tcl. That is, Tcl is a good glue languages for processes as well.

2.2 Process fewer characters

Tcl is interpreted, not compiled to native code or even to a “byte code” that in effect translates the human-readable text into an efficient machine-manipulable form. That is, in Tcl every character in the human-readable source text is seen by the interpreter during dynamic execution every time that line is executed—even code that was “commented out” must be reparsed. Until Tcl compilers relieve this situation, a pair of Tcl-specific tricks can maximize performance within these bounds.

Postprocess to reduce code size

Unlike comments in compiled programs, comments in Tcl and even the whitespace used to format the code reduce performance, especially if included within loops. Rather than move all comments outside of loops and start all lines at the left margin, one can postprocess the code with `sed` or `awk` as part of a Makefile to remove lines starting with the comment character (“#”) and strip initial spaces. In certain cases this can introduce errors into the program, but in my experience these cases almost never arise, and in fact rather than imposing a mental overhead to avoid these cases, it makes programming more straightforward for one accustomed to the C preprocessor, for when one knows that lines starting with a pound sign are comments, not possibly elements of a list. If one does not with to change Tcl semantics in this way, it is preferable to use the semantics-preserving `tcl_cruncher` [Dema], which strips comments and reduces whitespace (wherever it appears) to a minimum.

Reduce input size and command count

TkMan's external manual page filter (written in C) produces Tcl commands that insert the text, set tags for fonts and other presentation attributes, and set marks at page section heads. Changes in the Tcl generated by C, by the Tcl that executed this code and by a change in Tk 4.0's text widget syntax all contributed to reducing the time needed to read in a page by about 40% from version 1.5 to 1.6. Most changes were appeared trivially small at first glance, but they produced disproportionately large impact.

The filter formerly generated Tcl command strings like the following, at least one for each line:

```
$w.show insert end “NAME\n”
$w.show mark set m1 1.0
$w.show insert end “ls \[-aAbR\]\n”
$w.show tag add b 3.2 3.4
```

To read in this code and `eval` it, TkMan used the following code:

```
while {[gets $fid line]!=-1} {eval $line}
```

The simple change of not setting a variable yielded a significant speed up:

```
while {![eof $fid]} {eval [gets $fid]}
```

The fastest version, however, read in the entire file at once:

```
while {![eof $fid]} {eval [read $fid]}
```

But this had the unfortunate side effect ballooning the memory image when long pages were viewed. Tcl would correctly free the memory but UNIX would not shrink its allocation.

Since it is repeated for every line—over 400 times for the Perl 4 manual page—a trivial abbreviation of `$w.show` to `$t` significantly reduced input size.

Most importantly, however, was a change in Tk 4.0 text widget insert syntax that led to man page translations that required about 35% fewer characters and 98% fewer function dispatches through `eval`. The new syntax (the extended version of which was suggested by this author to enable this very optimization), adds tags to the text in the same command and can process multiple (*text, tag*) tuples. The text widget can then step through longer passages of text in the same function call and can to an extent cache internal position markers used in setting tags and marks. Thus the four lines above needed to process two lines of the ls manual page can be reduced to these two:

```
$t insert end “NAME\n” h2 “\n   “ {} “ls” b “ - Lists and generates statistics for files\n\n”
$t mark set m1 1.0
```

Together, these changes noticably reduced the time needed to process man page input text.

3. Exploiting Tcl as its Own Scripting Language

Well known are many advantages of working in an interpreted scripting language, such as rapid edit-compile-test cycles and high level control of functionality that is implemented more efficiently in a lower-level language. In the three cases described below, Tcl served as its own scripting language. Although the extension code was executed in the same environment as the implementation code and could directly access important data structures, it was important to operate through an interface layer in every case in order to make extension code portable across changes to the main application code and to provide a higher level, tailored interface to the application.

3.1 tkmandesc commands

It is occasionally desirable to see a list of manual page names in a particular category either to see what's available or to choose a name from a set of options rather than typing it in from memory. The man page directory hierarchy categorizes pages into 11 major groupings, and a browser could present lists of all pages in a particular grouping, but most groupings contain 100s and some groupings (“User Commands and “Subroutines”) 1000s of pages, thus mitigating the meaningfulness of the categories.

With `xman` one can create “pseudosections” and (re)group directories of pages into arbitrary sections. For instance, one could collect together all Tcl/Tk- or TeX-related pages in their own volume. Unfortunately, besides being verbose, the specifications file that describes the regroupings is mandatorily shared by all who share the same man pages. One user cannot regroup without forcing all users to see the same rearrangements.

In contrast, TkMan's set of tkmandesc commands read the specifications from a user's individual startup file. Control of volume reorganizations can be changed on a user-by-user basis. (Actually, since tkmandesc commands are ordinary Tcl code, one could `source` a common file of tkmandesc commands to share regroupings.) Also, some combination of shared and individual commands is possible.

Although user code can directly manipulate internal data structures, it is better to operate through the tkmandesc abstraction layer. The commands map better to objects in the application ontology, and the implementation of the intensional tkmandesc commands can change without affecting existing customization code.

The tkmandesc layer made possible a user-transparent solution to a problem introduced with the full-text searching package Glimpse [Manb94]. Usually pages are indexed in their hierarchy so that Glimpse indexes, though small, can be shared and thereby ammortized across users. The question was where to index isolated directories not part of any hierarchy. For performance reasons, it would be better to index them as a group. Because these isolated directories are added with tkmandesc commands, it was trivial to amass a list of all such isolated directories for Glimpse without any changes required of existing tkmandesc code.

Manual page organization is slightly different on every flavor of UNIX. Some vendors, for instance, ship only preformatted versions of pages, and HP-UX stores the pages in compressed files, but it is not the files that are named with a compression suffix, it is the directory. In most cases, it was possible to accomodate these variations without excessive stress on the code.

The exception was UNIX System V's organization, most prominently SGI's IRIX, which uses the common organization for user added manual pages, but stores built-in pages under one and sometimes two levels of indirection. To give one illustration, the audio-related subroutines, which in usual organizations would be placed in the directory `/usr/man/man3` and mixed with other subroutines, were placed in `/usr/catman/p_man/cat3/audio`. Supporting this required fairly extensive patching and led to a separate code path maintained by the author of the patches, Paul Raines. Unfortunately, now each new release of the software meant adjustment of and reapplication of the patches and a delay for SGI users.

With the introduction of tkmandesc customization commands, however, Paul could make a one-time translation of the patches into tkmandesc extension code. With its well defined and supported interface to TkMan entry points, this same extension code has, without modification, served to customize TkMan for SGI machines on all subsequent releases. Using Tcl as its own extension language solved this portability crisis.

3.2 Saving state

By now it is common practice to save application state as a sequence of Tcl commands (most of them set statements, probably) that can be `source`'d back in to restore this state. TkMan has refined this method in several ways.

It is also common practice to introduce a namespace into Tcl, though it relies on programmer discipline, by adding a unique prefix to all global variables and procedures in a module. TkMan entends this convention by placing all persistent globals in an array named by the prefix and the others in an array named *prefix*x ("x" for "expire" or "extinguish"). Now, rather than maintaining a list of variables to save, persistent variables are saved by simply querying Tcl for the names of elements in the persistent array and writing them out. There is no list of persistent variables to maintain; a naming convention and Tcl's capacity for introspection generate the correct results, guaranteed.

A similar trick calls each module's save state command. Because not all state is captured in variables, every module that needs to save state contains a proc named *prefix*SaveConfig. At application shutdown and at checkpoints, the Tcl interpreter is dynamically queried for all procs matching the pattern `*SaveConfig` (using the `info proc` command), and each match is executed with `eval`. Again, there is no list to maintain.

TkMan divides the startup file into a program-controlled section that is rewritten at every save and a user-controlled section that is copied verbatim each time. This small, aesthetic improvement avoids cluttering the user's home directory, the traditional location location for startup files, with two more files when one will do.

A final refinement concerns propagation of default values. Although the latest release of TkMan sports a graphical Preferences editor for most options, not all important state that the user might want to change is available through it. For this reason, we write out every interesting setting, each suggestively named as to its function. This could cause problems with new releases that change default values, as those changes would be overridden by the settings in the startup file. The solution is to comment out all values that are identical to the default values, reasoning that it is only those intensionally changed are evidence of user preference. The mechanics are straightforward: after the application's default values have been set in the persistent array, a loop through them sets parallel values in a "defaults" array; at save time, the two values are compared and any not changed by the user, either by the startup file or interactively in Preferences, are preceded by a "#". Now changes in the default values propagate nicely.

3.3 tkchrom, a scriptable clock

TkMan is a sizable application, one significant enough to justify an extension language by historical standards. Now, any application written in Tcl has available a free extension language, since Tcl can serve this purpose for itself. This section describes how a tiny application benefitted from this.

Olaf Heimburger's xchrom graphical clock displays time by a circular disc with a wedge that rotates at the center of a sunburst pattern with twelve rays. At the hour the wedge exactly outlines one ray; the closer the next hour, the more of its wedge is outlined and the less of the current hour. As an exercise, the current author rewrote this Xt/Xmu-based clock in Tcl/Tk. It took a mere 10% of the number of lines. On the other hand, the new clock, called tkchrom [Phel], required one megabyte of supporting libraries to run! For small applications like this, shared libraries would be a great relief.

If you're going to pay for it, you may as well use it. What can one do with a general purpose programing language in a clock? Reminders, unrestricted reminders. The user can place a set of pattern-action triggers in a startup file. Once per minute tkchrom will step through them trying to match the date and time, and for successful matches, execute the action code—which is arbitrary Tcl code. Certain "convenience variables" give access to the clock's internal state so that, for instance, the disc can be yellow during the day, black at night, orange on Halloween; or the clock can be erased in favor of a reminder message. Of course, this unrestricted Tcl can pop up dialog boxes, `exec` sound players, or send man page requests to TkMan....

4. Application Configuration

The UNIX memory allocation command `malloc` returns an error code because, as unlikely as it may seem, it is possible for a program's request for a 10K block of memory to be unfulfillable by a system with 64MB of main memory and 100s of megabytes of virtual memory. The advice below may seem to be asking for guards against conditions that never happen in practice. In fact, every one of them has happened to me personally, and I think that it is advice useful in writing programs that will be used by someone other than its author.

4.1 Installation

If users never read the manual, installers never read the Makefile. Especially if it requires specific knowledge of the system, take this setting out of the Makefile and instead determine the correct value at application runtime if possible. This can also makes scripts portable across architectures. Better still, write a graphical installation tool such as found in exmh [Welc], which can interrogate the environment at installation time. It still may be wise to defer some checks to application runtime, however, in order to dynamically customize the script for different machines and to catch changes in the environment since installation.

In particular, check for minimum required versions of Tcl *and* Tk (mismatched versions do get linked together), and since new major versions introduce incompatibilities, check against maximum compatibile versions. Check for the existence and executability of all supporting binaries and scripts, which requires looping through the PATH environment variable. Assume no software outside of the traditional core UNIX commands, not even the most popular such as Perl. In fact, not even traditional core binaries can be assumed: as a case in point, OSF/1 discards troff. For software under my control, I've found it useful to give each executable a `-v` command line option that reports its version number. That way the dependent code can check that it has sufficiently up to date supporting code, and that those programs are actually executable (compiled for the right machine architecture, et cetera). Although the popular GNU software seems to have adopted a `-V` convention, most standard UNIX utilities do not have version reporting option, unfortunately.

During runtime, do follow the old advice to check for an error wherever one can occur. In my experience, errors have occurred in trying to write a file to the user's home directory (no write permission!), and in another case, after writing the file successfully, an error occurred when trying to compress it (the disk filled up during compression when both the original and the growing partially-compressed version existed simultaneously). Finally, if the script relies on environment variables, check their validity. The one most important to TkMan, MANPATH, which list all man directories to search for pages, is typically built up piecemeal in the user's login scripts, and pieces are often empty, invalid (i.e., they refer to a non-existent directory), or syntactically ill-formed (e.g., a directory ends with a trailing slash, "/").

To paraphrase Ben Bederson at the Tcl/Tk '94 Workshop, "I'll give you one minute to download the software and one minute to install it. If it doesn't work I'm not interested."

4.2 User preferences

Once the software has been installed, configuration is not finished. The user will want to change the fonts, colors, window sizes, icon, scrollbar side, and more. If an author is serious about making this level of customization available to the user, he *must* write a graphical Preferences editor. Experience shows that X resources are simply too obscure to figure out by non-programmers, and even programmers will only pursue the most annoying settings. Furthermore, X font names virtually require a reference manual to specify. If nothing else, an easy means to change the font size is invaluable for demos.

After the changes are made, dynamically show the new look by taking advantages of Tk's introspective nature to step through widget tree resetting fonts and colors, and to repack widgets as necessary. Use these settings with Tk's `option` command during the startup sequence to avoid an unnecessary widget traversal just to set the correct values. Lastly ask Tk (`wm geometry`) for the window size and position and restore these settings at the next application invocation.

5. Interoperating Tools

5.1 GUI-ifying existing text-based tools

TkMan aims to replace the system's text-based `man` command, to ignore `man`'s existence. This replacement is required to provide some functionality, like tkmandesc commands, and eliminates reliance on any particular OS variant of `man`, in effect bringing them all up to the same level (adding automatic page text decompression, for example).

One might think that writing at the Tcl level would automatically make the code portable across all platforms that support the Tcl interpreter. But environment differences thwart this ideal. In writing Expect [Libe94a], its author lamented that he had to implement "over 20 interfaces to handle ptys" [Libe94b] to accomodate the differences among flavors of UNIX.

Although TkMan was largely successful in replacing `man`, in two cases it had to rely on `man` to search for or obtain the text of manual pages: when pages are stored in a proprietary encoding and are therefore available only through the supplied `man`; and when the MANPATH changes frequently after TkMan startup, as when packages are added and removed, thereby invalidating the database. To support these two cases, TkMan drops down to use the system `man`, at the cost of some functionality.

Expect was designed to manage communication with text-based processes, but in some cases if the other tool was not designed for potential use by another program, it is impossible to obtain satisfactory results. In the case of `man`, some implementations provide a command line option (`-w`) that lists the pathnames of all matching names. TkMan can use this to provide a choice among them in a pulldown list. Not considered by other `man` implementations is the fact that not just the page text but also this meta information may of use to other programs. If `man` only supplies page text, TkMan does show that, at the unavoidable loss of the choice among matches.

In short, some text-based tools—which were perhaps implemented before the widespread use of graphical interfaces—impose unnecessary limitations on authors who wish to build on their work for the simple, easily avoided reason that they do not expose essential information with well defined interfaces.

5.2 Hypertools, step two

One should not make the same mistake with graphical tools. If Tk's `send` command is the first step toward interoperating graphical tools, or "hypertools" [Oust93], step two is a set of well defined and abstracted interface functions for use by other programs. Do not assume that the ultimate user of an application with a graphical application is a human.

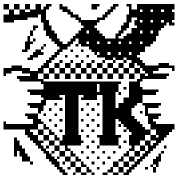
TkMan publishes interfaces for searching for and showing a man page given its name specification (e.g., "text.t" means search for the page named "text" in section "t"); for showing a page given its complete pathname; and for the search proc itself (so that it can be replaced by a call to `man`). These entry points are used by Neil Smithline's small, screen real estate conserving type-in box (which is included in TkMan's `contrib` directory), and interoperability is planned for tkinfo [Whit] and the jhelp/jdoc module of jstools [Seko].

These well defined entry points are useful internally as well. Each of the many ways to specify a manual page—type-in box, double click in text, double click in volumes list, entry in history menu, entry in links menu, entry in multiple matches choice list—calls the main seach-and-show function. In addition, in order to show freshly-added pages or pages not in the MANPATH, the user can type in the full pathname, which of course simply invokes the corresponding public interface point.

More and more commercial software is published with an application programming interface (API). One might be tempted to think that because `send` can execute arbitrary Tcl code in the remote tool, "official" entry points are less important. However, if hypertools are to continue their cooperation across code revisions, semantically meaningful interfaces to all important system information are important.

6. Conclusions

Sometimes reading a list of coding do's and don'ts seems like reading a list of tautologies: "Structure your code. Guard for errors wherever they can happen." I hope that my war stories with one Tcl application bring them to life while sparing the pain of acquiring them firsthand. Furthermore, I hope that solutions I devised within the limitation Tcl as well as the opportunities I exploited given the potentialities of Tcl can be put to use by other Tcl/Tk authors in the production of higher quality software.



7. Acknowledgements

Comments from Michael Schiff, Adam Sah, and David Berger materially improved this paper. I thank them.

In the development of the software, I acknowledge Paul Raines for the SGI port, Rei Shinozuka for witty icon shown above, and the legions from the net whose interest in the software roused me to improve it for the benefit of all, with the side dividend of a rich set of interesting design problems for the author.

8. Availability

The code for TkMan is available from URL `ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/tkman.tar.Z`. It requires the manual page filter RosettaMan, available from `ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/rman.tar.Z`. Releases of these two programs are numbered; the addresses above link to the latest stable versions. Also at that location is a technical report describing TkMan [Phel94a], which is an earlier version of an X Resource article [Phel94b].

9. References

[Dema]	Laurent Demailly. tcl_cruncher. Available from ftp://ftp.aud.alcatel.com/tcl/code.
[Libe94a]	Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., December 1994.
[Libe94b]	Don Libes. Private communication, September 1994.
[Manb94]	U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the Usenix Winter 1994 Technical Conference*, pages 23–32, January 1994.
[Phel]	Thomas A. Phelps. tkchrom, a graphical clock. Available at ftp://ftp.cs.berkeley.edu/ucb/people/phelps/tcltk/tkchrom.
[Phel94a]	Thomas A. Phelps. Domain-specific information browsers for man page, file, and font. Technical Report UCB/CSD 94-802, University of California, Berkeley, 1994.
[Phel94b]	Thomas A. Phelps. TkMan: A man born again. *The X Resource*, 1(10):33–46, 1994.
[Seko]	Jay Sekora. jstools. Available from ftp://ftp.aud.alcatel.com/tcl/code.
[Shei]	Barry Shein and Chris Peterson. xman. Included with the X Window System distribution.
[Welc]	Brent Welch. exmh. Available at ftp://ftp.parc.xerox.com/pub/exmh.
[Whit]	Kennard White. tkinfo. Available at ftp://ftp.aud.alcatel.com/tcl/code/tkinfo-0.6.tar.gz.