Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'tcsh.1' command

$ man tcsh.1

TCSH(1) General Commands Manual TCSH(1)

NAME

tcsh - C shell with file name completion and command line editing

SYNOPSIS

tcsh [-bcdefFimnqstvVxX] [-Dname[=value]] [arg ...]

tcsh -l

DESCRIPTION

tcsh is an enhanced but completely compatible version of the Berkeley UNIX C shell, csh(1). It is a command language interpreter usable both as an interactive login shell and a shell script command processor. It includes a command-line editor (see The command-line editor), program? mable word completion (see Completion and listing), spelling correction (see Spelling correction), a history mechanism (see History substitu? tion), job control (see Jobs) and a C-like syntax. The NEW FEATURES section describes major enhancements of tcsh over csh(1). Throughout this manual, features of tcsh not found in most csh(1) implementations (specifically, the 4.4BSD csh) are labeled with `(+)', and features which are present in csh(1) but not usually documented are labeled with `(u)'.

Argument list processing

If the first argument (argument 0) to the shell is `-' then it is a lo? gin shell. A login shell can be also specified by invoking the shell with the -l flag as the only argument.

The rest of the flag arguments are interpreted as follows:

-b Forces a ``break'' from option processing, causing any further shell arguments to be treated as non-option arguments. The remaining arguments will not be interpreted as shell options. This may be used to pass options to a shell script without confusion or possible subterfuge. The shell will not run a set-user ID script without this option.

-c Commands are read from the following argument (which must be present, and must be a single argument), stored in the command shell variable for reference, and executed. Any remaining arguments are placed in the argv shell variable.

-d The shell loads the directory stack from ~/.cshdirs as described under Startup and shutdown, whether or not it is a login shell. (+)

-Dname[=value]

Sets the environment variable name to value. (Domain/OS only) (+)

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell does not load any resource or startup files, or perform any command hashing, and thus starts faster.

-F The shell uses fork(2) instead of vfork(2) to spawn processes. (+)

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-l The shell is a login shell. Applicable only if -l is the only flag specified.

-m The shell loads ~/.tcshrc even if it does not belong to the effective user. Newer versions of su(1) can pass -m to the shell. (+)

-n The shell parses commands but does not execute them. This aids in debugging shell scripts.

-q The shell accepts SIGQUIT (see Signal handling) and behaves when it is used under a debugger. Job control is disabled. (u)

-s Command input is taken from the standard input.

-t The shell reads and executes a single line of input. A `\' may be

used to escape the newline at the end of this line and continue onto another line.

-v Sets the verbose shell variable, so that command input is echoed after history substitution.

-x Sets the echo shell variable, so that commands are echoed immedi? ately before execution.

-V Sets the verbose shell variable even before executing ~/.tcshrc.

-X Is to -x as -V is to -v.

--help

Print a help message on the standard output and exit. (+)

--version

Print the version/platform/compilation options on the standard out? put and exit. This information is also contained in the version shell variable. (+)

After processing of flag arguments, if arguments remain but none of the -c, -i, -s, or -t options were given, the first argument is taken as the name of a file of commands, or ``script'', to be executed. The shell opens this file and saves its name for possible resubstitution by `$0'. Because many systems use either the standard version 6 or ver? sion 7 shells whose shell scripts are not compatible with this shell, the shell uses such a `standard' shell to execute a script whose first character is not a `#', i.e., that does not start with a comment. Remaining arguments are placed in the argv shell variable.

Startup and shutdown

A login shell begins by executing commands from the system files /etc/csh.cshrc and /etc/csh.login. It then executes commands from files in the user's home directory: first ~/.tcshrc (+) or, if ~/.tc? shrc is not found, ~/.cshrc, then the contents of ~/.history (or the value of the histfile shell variable) are loaded into memory, then ~/.login, and finally ~/.cshdirs (or the value of the dirsfile shell variable) (+). The shell may read /etc/csh.login before instead of af? ter /etc/csh.cshrc, and ~/.login before instead of after ~/.tcshrc or ~/.cshrc and ~/.history, if so compiled; see the version shell vari?

able. (+)

Non-login shells read only /etc/csh.cshrc and ~/.tcshrc or ~/.cshrc on startup.

For examples of startup files, please consult http://tcshrc.source?forge.net.

Commands like stty(1) and tset(1), which need be run only once per lo?gin, usually go in one's ~/.login file. Users who need to use the same set of files with both csh(1) and tcsh can have only a ~/.cshrc which checks for the existence of the tcsh shell variable (q.v.) before using tcsh-specific commands, or can have both a ~/.cshrc and a ~/.tcshrc which sources (see the builtin command) ~/.cshrc. The rest of this manual uses `~/.tcshrc' to mean `~/.tcshrc or, if ~/.tcshrc is not found, ~/.cshrc'.

In the normal case, the shell begins reading commands from the termi?nal, prompting with `> '. (Processing of arguments and the use of the shell to process files containing command scripts are described later.)

The shell repeatedly reads a line of command input, breaks it into words, places it on the command history list, parses it and executes each command in the line.

One can log out by typing `^D' on an empty line, `logout' or `login' or via the shell's autologout mechanism (see the autologout shell vari?able). When a login shell terminates it sets the logout shell variable to `normal' or `automatic' as appropriate, then executes commands from the files /etc/csh.logout and ~/.logout. The shell may drop DTR on lo?gout if so compiled; see the version shell variable.

The names of the system login and logout files vary from system to sys?tem for compatibility with different csh(1) variants; see FILES.

Editing

We first describe The command-line editor. The Completion and listing and Spelling correction sections describe two sets of functionality that are implemented as editor commands but which deserve their own treatment. Finally, Editor commands lists and describes the editor commands specific to the shell and their default bindings.

The command-line editor (+)

Command-line input can be edited using key sequences much like those used in emacs(1) or vi(1). The editor is active only when the edit shell variable is set, which it is by default in interactive shells.

The bindkey builtin can display and change key bindings. emacs(1)-style key bindings are used by default (unless the shell was compiled otherwise; see the version shell variable), but bindkey can change the key bindings to vi(1)-style bindings en masse.

The shell always binds the arrow keys (as defined in the TERMCAP envi? ronment variable) to

down down-history

up up-history

left backward-char

right forward-char

unless doing so would alter another single-character binding. One can set the arrow key escape sequences to the empty string with settc to prevent these bindings. The ANSI/VT100 sequences for arrow keys are always bound.

Other key bindings are, for the most part, what emacs(1) and vi(1) users would expect and can easily be displayed by bindkey, so there is no need to list them here. Likewise, bindkey can list the editor com? mands with a short description of each. Certain key bindings have dif? ferent behavior depending if emacs(1) or vi(1) style bindings are being used; see vimode for more information.

Note that editor commands do not have the same notion of a ``word'' as does the shell. The editor delimits words with any non-alphanumeric characters not in the shell variable wordchars, while the shell recog? nizes only whitespace and some of the characters with special meanings to it, listed under Lexical structure.

Completion and listing (+)

The shell is often able to complete words when given a unique abbrevia? tion. Type part of a word (for example `ls /usr/lost') and hit the tab key to run the complete-word editor command. The shell completes the

filename `/usr/lost' to `/usr/lost+found/', replacing the incomplete word with the complete word in the input buffer. (Note the terminal `/'; completion adds a `/' to the end of completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion. The addsuffix shell vari? able can be unset to prevent this.) If no match is found (perhaps `/usr/lost+found' doesn't exist), the terminal bell rings. If the word is already complete (perhaps there is a `/usr/lost' on your system, or perhaps you were thinking too far ahead and typed the whole thing) a `/' or space is added to the end if it isn't already there.

Completion works anywhere in the line, not at just the end; completed text pushes the rest of the line to the right. Completion in the mid? dle of a word often results in leftover characters to the right of the cursor that need to be deleted.

Commands and variables can be completed in much the same way. For ex? ample, typing `em[tab]' would complete `em' to `emacs' if emacs were the only command on your system beginning with `em'. Completion can find a command in any directory in path or if given a full pathname. Typing `echo $ar[tab]' would complete `$ar' to `$argv' if no other variable began with `ar'.

The shell parses the input buffer to determine whether the word you want to complete should be completed as a filename, command or vari? able. The first word in the buffer and the first word following `;', `|', `|&', `&&' or `||' is considered to be a command. A word begin? ning with `$' is considered to be a variable. Anything else is a file? name. An empty line is `completed' as a filename.

You can list the possible completions of a word at any time by typing `^D' to run the delete-char-or-list-or-eof editor command. The shell lists the possible completions using the ls-F builtin (q.v.) and re? prints the prompt and unfinished command line, for example:

```
> ls /usr/l[^D]
lbin/       lib/        local/      lost+found/
> ls /usr/l
```

If the autolist shell variable is set, the shell lists the remaining choices (if any) whenever completion fails:

```
> set autolist
> nm /usr/lib/libt[tab]
libtermcap.a@ libtermlib.a@
> nm /usr/lib/libterm
```

If autolist is set to `ambiguous', choices are listed only when comple? tion fails and adds no new characters to the word being completed.

A filename to be completed can contain variables, your own or others' home directories abbreviated with `~' (see Filename substitution) and directory stack entries abbreviated with `=' (see Directory stack sub? stitution). For example,

```
> ls ~k[^D]
kahn    kas     kellogg
> ls ~ke[tab]
> ls ~kellogg/
```

or

```
> set local = /usr/local
> ls $lo[tab]
> ls $local/[^D]
bin/ etc/ lib/ man/ src/
> ls $local/
```

Note that variables can also be expanded explicitly with the expand-variables editor command.

delete-char-or-list-or-eof lists at only the end of the line; in the middle of a line it deletes the character under the cursor and on an empty line it logs one out or, if ignoreeof is set, does nothing.

`M-^D', bound to the editor command list-choices, lists completion pos? sibilities anywhere on a line, and list-choices (or any one of the re? lated editor commands that do or don't delete, list and/or log out, listed under delete-char-or-list-or-eof) can be bound to `^D' with the bindkey builtin command if so desired.

The complete-word-fwd and complete-word-back editor commands (not bound

to any keys by default) can be used to cycle up and down through the list of possible completions, replacing the current word with the next or previous word in the list.

The shell variable fignore can be set to a list of suffixes to be ig? nored by completion. Consider the following:

```
> ls
Makefile        condiments.h~   main.o          side.c
README          main.c          meal            side.o
condiments.h    main.c~
> set fignore = (.o \~)
> emacs ma[^D]
main.c   main.c~  main.o
> emacs ma[tab]
> emacs main.c
```

`main.c~' and `main.o' are ignored by completion (but not listing), be? cause they end in suffixes in fignore. Note that a `\' was needed in front of `~' to prevent it from being expanded to home as described un? der Filename substitution. fignore is ignored if only one completion is possible.

If the complete shell variable is set to `enhance', completion 1) ig? nores case and 2) considers periods, hyphens and underscores (`.', `-' and `_') to be word separators and hyphens and underscores to be equiv? alent. If you had the following files

```
comp.lang.c      comp.lang.perl   comp.std.c++
comp.lang.c++    comp.std.c
```

and typed `mail -f c.l.c[tab]', it would be completed to `mail -f comp.lang.c', and ^D would list `comp.lang.c' and `comp.lang.c++'. `mail -f c..c++[^D]' would list `comp.lang.c++' and `comp.std.c++'.

Typing `rm a--file[^D]' in the following directory

```
A_silly_file    a-hyphenated-file    another_silly_file
```

would list all three files, because case is ignored and hyphens and un? derscores are equivalent. Periods, however, are not equivalent to hy? phens or underscores.

If the complete shell variable is set to `Enhance', completion ignores case and differences between a hyphen and an underscore word separator only when the user types a lowercase character or a hyphen. Entering an uppercase character or an underscore will not match the correspond? ing lowercase character or hyphen word separator. Typing `rm a--file[^D]' in the directory of the previous example would still list all three files, but typing `rm A--file' would match only `A_silly_file' and typing `rm a__file[^D]' would match just `A_silly_file' and `another_silly_file' because the user explicitly used an uppercase or an underscore character.

Completion and listing are affected by several other shell variables: recexact can be set to complete on the shortest possible unique match, even if more typing might result in a longer match:

```
> ls
fodder  foo     food    foonly
> set recexact
> rm fo[tab]
```

just beeps, because `fo' could expand to `fod' or `foo', but if we type another `o',

```
> rm foo[tab]
> rm foo
```

the completion completes on `foo', even though `food' and `foonly' also match. autoexpand can be set to run the expand-history editor command before each completion attempt, autocorrect can be set to spelling-cor? rect the word to be completed (see Spelling correction) before each completion attempt and correct can be set to complete commands automat? ically after one hits `return'. matchbeep can be set to make comple? tion beep or not beep in a variety of situations, and nobeep can be set to never beep at all. nostat can be set to a list of directories and/or patterns that match directories to prevent the completion mecha? nism from stat(2)ing those directories. listmax and listmaxrows can be set to limit the number of items and rows (respectively) that are listed without asking first. recognize_only_executables can be set to

make the shell list only executables when listing commands, but it is quite slow.

Finally, the complete builtin command can be used to tell the shell how to complete words other than filenames, commands and variables. Com? pletion and listing do not work on glob-patterns (see Filename substi? tution), but the list-glob and expand-glob editor commands perform equivalent functions for glob-patterns.

Spelling correction (+)

The shell can sometimes correct the spelling of filenames, commands and variable names as well as completing and listing them.

Individual words can be spelling-corrected with the spell-word editor command (usually bound to M-s and M-S) and the entire input buffer with spell-line (usually bound to M-$). The correct shell variable can be set to `cmd' to correct the command name or `all' to correct the entire line each time return is typed, and autocorrect can be set to correct the word to be completed before each completion attempt.

When spelling correction is invoked in any of these ways and the shell thinks that any part of the command line is misspelled, it prompts with the corrected line:

```
> set correct = cmd
> lz /usr/bin
CORRECT>ls /usr/bin (y|n|e|a)?
```

One can answer `y' or space to execute the corrected line, `e' to leave the uncorrected command in the input buffer, `a' to abort the command as if `^C' had been hit, and anything else to execute the original line unchanged.

Spelling correction recognizes user-defined completions (see the com? plete builtin command). If an input word in a position for which a completion is defined resembles a word in the completion list, spelling correction registers a misspelling and suggests the latter word as a correction. However, if the input word does not match any of the pos? sible completions for that position, spelling correction does not reg? ister a misspelling.

Like completion, spelling correction works anywhere in the line, push? ing the rest of the line to the right and possibly leaving extra char? acters to the right of the cursor.

Editor commands (+)

`bindkey' lists key bindings and `bindkey -l' lists and briefly de? scribes editor commands. Only new or especially interesting editor commands are described here. See emacs(1) and vi(1) for descriptions of each editor's key bindings.

The character or characters to which each command is bound by default is given in parentheses. `^character' means a control character and `M-character' a meta character, typed as escape-character on terminals without a meta key. Case counts, but commands that are bound to let? ters by default are bound to both lower- and uppercase letters for con? venience.

backward-char (^B, left)

Move back a character. Cursor behavior modified by vimode.

backward-delete-word (M-^H, M-^?)

Cut from beginning of current word to cursor - saved in cut buffer. Word boundary behavior modified by vimode.

backward-word (M-b, M-B)

Move to beginning of current word. Word boundary and cursor behavior modified by vimode.

beginning-of-line (^A, home)

Move to beginning of line. Cursor behavior modified by vimode.

capitalize-word (M-c, M-C)

Capitalize the characters from cursor to end of current word. Word boundary behavior modified by vimode.

complete-word (tab)

Completes a word as described under Completion and listing.

complete-word-back (not bound)

Like complete-word-fwd, but steps up from the end of the list.

complete-word-fwd (not bound)

Replaces the current word with the first word in the list of

possible completions. May be repeated to step down through the list. At the end of the list, beeps and reverts to the incom? plete word.

complete-word-raw (^X-tab)

Like complete-word, but ignores user-defined completions.

copy-prev-word (M-^_)

Copies the previous word in the current line into the input buffer. See also insert-last-word. Word boundary behavior modified by vimode.

dabbrev-expand (M-/)

Expands the current word to the most recent preceding one for which the current is a leading substring, wrapping around the history list (once) if necessary. Repeating dabbrev-expand without any intervening typing changes to the next previous word etc., skipping identical matches much like history-search-backward does.

delete-char (not bound)

Deletes the character under the cursor. See also delete-char-or-list-or-eof. Cursor behavior modified by vimode.

delete-char-or-eof (not bound)

Does delete-char if there is a character under the cursor or end-of-file on an empty line. See also delete-char-or-list-or-eof. Cursor behavior modified by vimode.

delete-char-or-list (not bound)

Does delete-char if there is a character under the cursor or list-choices at the end of the line. See also delete-char-or-list-or-eof.

delete-char-or-list-or-eof (^D)

Does delete-char if there is a character under the cursor, list-choices at the end of the line or end-of-file on an empty line. See also those three commands, each of which does only a single action, and delete-char-or-eof, delete-char-or-list and list-or-eof, each of which does a different two out of the

three.

delete-word (M-d, M-D)

Cut from cursor to end of current word - save in cut buffer.

Word boundary behavior modified by vimode.

down-history (down-arrow, ^N)

Like up-history, but steps down, stopping at the original input line.

downcase-word (M-l, M-L)

Lowercase the characters from cursor to end of current word.

Word boundary behavior modified by vimode.

end-of-file (not bound)

Signals an end of file, causing the shell to exit unless the ignoreeof shell variable (q.v.) is set to prevent this. See also delete-char-or-list-or-eof.

end-of-line (^E, end)

Move cursor to end of line. Cursor behavior modified by vi? mode.

expand-history (M-space)

Expands history substitutions in the current word. See History substitution. See also magic-space, toggle-literal-history and the autoexpand shell variable.

expand-glob (^X-*)

Expands the glob-pattern to the left of the cursor. See File? name substitution.

expand-line (not bound)

Like expand-history, but expands history substitutions in each word in the input buffer.

expand-variables (^X-$)

Expands the variable to the left of the cursor. See Variable substitution.

forward-char (^F, right)

Move forward one character. Cursor behavior modified by vi? mode.

forward-word (M-f, M-F)

Move forward to end of current word. Word boundary and cursor behavior modified by vimode.

history-search-backward (M-p, M-P)

Searches backwards through the history list for a command be? ginning with the current contents of the input buffer up to the cursor and copies it into the input buffer. The search string may be a glob-pattern (see Filename substitution) containing `*', `?', `[]' or `{}'. up-history and down-history will pro? ceed from the appropriate point in the history list. Emacs mode only. See also history-search-forward and i-search-back.

history-search-forward (M-n, M-N)

Like history-search-backward, but searches forward.

i-search-back (not bound)

Searches backward like history-search-backward, copies the first match into the input buffer with the cursor positioned at the end of the pattern, and prompts with `bck: ' and the first match. Additional characters may be typed to extend the search, i-search-back may be typed to continue searching with the same pattern, wrapping around the history list if neces? sary, (i-search-back must be bound to a single character for this to work) or one of the following special characters may be typed:

^W Appends the rest of the word under the cursor to the search pattern.

delete (or any character bound to backward-delete-char)
Undoes the effect of the last character typed and deletes a character from the search pattern if ap? propriate.

^G If the previous search was successful, aborts the entire search. If not, goes back to the last suc? cessful search.

escape Ends the search, leaving the current line in the

input buffer.

Any other character not bound to self-insert-command terminates the search, leaving the current line in the input buffer, and is then interpreted as normal input. In particular, a carriage return causes the current line to be executed. See also i-search-fwd and history-search-backward. Word boundary behavior modified by vimode.

i-search-fwd (not bound)

Like i-search-back, but searches forward. Word boundary behav? ior modified by vimode.

insert-last-word (M-_)

Inserts the last word of the previous input line (`!$') into the input buffer. See also copy-prev-word.

list-choices (M-^D)

Lists completion possibilities as described under Completion and listing. See also delete-char-or-list-or-eof and list-choices-raw.

list-choices-raw (^X-^D)

Like list-choices, but ignores user-defined completions.

list-glob (^X-g, ^X-G)

Lists (via the ls-F builtin) matches to the glob-pattern (see Filename substitution) to the left of the cursor.

list-or-eof (not bound)

Does list-choices or end-of-file on an empty line. See also delete-char-or-list-or-eof.

magic-space (not bound)

Expands history substitutions in the current line, like expand-history, and inserts a space. magic-space is designed to be bound to the space bar, but is not bound by default.

normalize-command (^X-?)

Searches for the current word in PATH and, if it is found, re? places it with the full path to the executable. Special char? acters are quoted. Aliases are expanded and quoted but com?

mands within aliases are not. This command is useful with com?
mands that take commands as arguments, e.g., `dbx' and `sh -x'.

normalize-path (^X-n, ^X-N)

Expands the current word as described under the `expand' set?
ting of the symlinks shell variable.

overwrite-mode (unbound)

Toggles between input and overwrite modes.

run-fg-editor (M-^Z)

Saves the current input line and looks for a stopped job where
the file name portion of its first word is found in the editors
shell variable. If editors is not set, then the file name por?
tion of the EDITOR environment variable (`ed' if unset) and the
VISUAL environment variable (`vi' if unset) will be used. If
such a job is found, it is restarted as if `fg %job' had been
typed. This is used to toggle back and forth between an editor
and the shell easily. Some people bind this command to `^Z' so
they can do this even more easily.

run-help (M-h, M-H)

Searches for documentation on the current command, using the
same notion of `current command' as the completion routines,
and prints it. There is no way to use a pager; run-help is de?
signed for short help files. If the special alias helpcommand
is defined, it is run with the command name as a sole argument.
Else, documentation should be in a file named command.help,
command.1, command.6, command.8 or command, which should be in
one of the directories listed in the HPATH environment vari?
able. If there is more than one help file only the first is
printed.

self-insert-command (text characters)

In insert mode (the default), inserts the typed character into
the input line after the character under the cursor. In over?
write mode, replaces the character under the cursor with the
typed character. The input mode is normally preserved between

lines, but the inputmode shell variable can be set to `insert' or `overwrite' to put the editor in that mode at the beginning of each line. See also overwrite-mode.

sequence-lead-in (arrow prefix, meta prefix, ^X)

Indicates that the following characters are part of a multi-key sequence. Binding a command to a multi-key sequence really creates two bindings: the first character to sequence-lead-in and the whole sequence to the command. All sequences beginning with a character bound to sequence-lead-in are effectively bound to undefined-key unless bound to another command.

spell-line (M-$)

Attempts to correct the spelling of each word in the input buf? fer, like spell-word, but ignores words whose first character is one of `-', `!', `^' or `%', or which contain `\', `*' or `?', to avoid problems with switches, substitutions and the like. See Spelling correction.

spell-word (M-s, M-S)

Attempts to correct the spelling of the current word as de? scribed under Spelling correction. Checks each component of a word which appears to be a pathname.

toggle-literal-history (M-r, M-R)

Expands or `unexpands' history substitutions in the input buf? fer. See also expand-history and the autoexpand shell vari? able.

undefined-key (any unbound key)

Beeps.

up-history (up-arrow, ^P)

Copies the previous entry in the history list into the input buffer. If histlit is set, uses the literal form of the entry. May be repeated to step up through the history list, stopping at the top.

upcase-word (M-u, M-U)

Uppercase the characters from cursor to end of current word.

Word boundary behavior modified by vimode.

vi-beginning-of-next-word (not bound)

Vi goto the beginning of next word. Word boundary and cursor behavior modified by vimode.

vi-eword (not bound)

Vi move to the end of the current word. Word boundary behavior modified by vimode.

vi-search-back (?)

Prompts with `?' for a search string (which may be a glob-pat? tern, as with history-search-backward), searches for it and copies it into the input buffer. The bell rings if no match is found. Hitting return ends the search and leaves the last match in the input buffer. Hitting escape ends the search and executes the match. vi mode only.

vi-search-fwd (/)

Like vi-search-back, but searches forward.

which-command (M-?)

Does a which (see the description of the builtin command) on the first word of the input buffer.

yank-pop (M-y)

When executed immediately after a yank or another yank-pop, re? places the yanked string with the next previous string from the killring. This also has the effect of rotating the killring, such that this string will be considered the most recently killed by a later yank command. Repeating yank-pop will cycle through the killring any number of times.

Lexical structure

The shell splits input lines into words at blanks and tabs. The spe? cial characters `&', `|', `;', `<', `>', `(', and `)' and the doubled characters `&&', `||', `<<' and `>>' are always separate words, whether or not they are surrounded by whitespace.

When the shell's input is not a terminal, the character `#' is taken to begin a comment. Each `#' and the rest of the input line on which it

appears is discarded before further parsing.

A special character (including a blank or tab) may be prevented from having its special meaning, and possibly made part of another word, by preceding it with a backslash (`\') or enclosing it in single (`'), double (`"') or backward (``') quotes. When not otherwise quoted a newline preceded by a `\' is equivalent to a blank, but inside quotes this sequence results in a newline.

Furthermore, all Substitutions (see below) except History substitution can be prevented by enclosing the strings (or parts of strings) in which they appear with single quotes or by quoting the crucial charac? ter(s) (e.g., `$' or ``' for Variable substitution or Command substitu? tion respectively) with `\'. (Alias substitution is no exception: quoting in any way any character of a word for which an alias has been defined prevents substitution of the alias. The usual way of quoting an alias is to precede it with a backslash.) History substitution is prevented by backslashes but not by single quotes. Strings quoted with double or backward quotes undergo Variable substitution and Command substitution, but other substitutions are prevented.

Text inside single or double quotes becomes a single word (or part of one). Metacharacters in these strings, including blanks and tabs, do not form separate words. Only in one special case (see Command substi? tution below) can a double-quoted string yield parts of more than one word; single-quoted strings never do. Backward quotes are special: they signal Command substitution (q.v.), which may result in more than one word.

Quoting complex strings, particularly strings which themselves contain quoting characters, can be confusing. Remember that quotes need not be used as they are in human writing! It may be easier to quote not an entire string, but only those parts of the string which need quoting, using different types of quoting to do so if appropriate.

The backslash_quote shell variable (q.v.) can be set to make back? slashes always quote `\', `', and `"'. (+) This may make complex quoting tasks easier, but it can cause syntax errors in csh(1) scripts.

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur. We note in passing the data structures involved and the commands and variables which affect them. Remember that substitutions can be prevented by quoting as described under Lexical structure.

History substitution

Each command, or ``event'', input from the terminal is saved in the history list. The previous command is always saved, and the history shell variable can be set to a number to save that many commands. The histdup shell variable can be set to not save duplicate events or con? secutive duplicate events.

Saved commands are numbered sequentially from 1 and stamped with the time. It is not usually necessary to use event numbers, but the cur? rent event number can be made part of the prompt by placing an `!' in the prompt shell variable.

By default history entries are displayed by printing each parsed token separated by space; thus the redirection operator `>&!' will be dis? played as `> & !'.

The shell actually saves history in expanded and literal (unexpanded) forms. If the histlit shell variable is set, commands that display and store history use the literal form.

The history builtin command can print, store in a file, restore and clear the history list at any time, and the savehist and histfile shell variables can be set to store the history list automatically on logout and restore it on login.

History substitutions introduce words from the history list into the input stream, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confi? dence.

History substitutions begin with the character `!'. They may begin anywhere in the input stream, but they do not nest. The `!' may be

preceded by a `\' to prevent its special meaning; for convenience, a `!' is passed unchanged when it is followed by a blank, tab, newline, `=' or `('. History substitutions also occur when an input line begins with `^'. This special abbreviation will be described later. The characters used to signal history substitution (`!' and `^') can be changed by setting the histchars shell variable. Any input line which contains a history substitution is printed before it is executed.

A history substitution may have an ``event specification'', which indi? cates the event from which words are to be taken, a ``word designa? tor'', which selects particular words from the chosen event, and/or a ``modifier'', which manipulates the selected words.

An event specification can be

n A number, referring to a particular event

-n An offset, referring to the event n before the current event

The current event. This should be used carefully in csh(1), where there is no check for recursion. tcsh allows 10 levels of recursion. (+)

! The previous event (equivalent to `-1')

s The most recent event whose first word begins with the string s

?s? The most recent event which contains the string s. The second `?' can be omitted if it is immediately followed by a newline.

For example, consider this bit of someone's history list:

```
 9  8:30    nroff -man wumpus.man
10  8:31    cp wumpus.man wumpus.man.old
11  8:36    vi wumpus.man
12  8:37    diff wumpus.man.old wumpus.man
```

The commands are shown with their event numbers and time stamps. The current event, which we haven't typed in yet, is event 13. `!11' and `!-2' refer to event 11. `!!' refers to the previous event, 12. `!!' can be abbreviated `!' if it is followed by `:' (`:' is described be?

low). `!n' refers to event 9, which begins with `n'. `!?old?' also refers to event 12, which contains `old'. Without word designators or modifiers history references simply expand to the entire event, so we might type `!cp' to redo the copy command or `!!|more' if the `diff' output scrolled off the top of the screen.

History references may be insulated from the surrounding text with braces if necessary. For example, `!vdoc' would look for a command be? ginning with `vdoc', and, in this example, not find one, but `!{v}doc' would expand unambiguously to `vi wumpus.mandoc'. Even in braces, his? tory substitutions do not nest.

(+) While csh(1) expands, for example, `!3d' to event 3 with the letter `d' appended to it, tcsh expands it to the last event beginning with `3d'; only completely numeric arguments are treated as event numbers. This makes it possible to recall events beginning with numbers. To ex? pand `!3d' as in csh(1) say `!{3}d'.

To select words from an event we can follow the event specification by a `:' and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	The first (command) word
n	The nth argument
^	The first argument, equivalent to `1'
$	The last argument
%	The word matched by an ?s? search
x-y	A range of words
-y	Equivalent to `0-y'
*	Equivalent to `^-$', but returns nothing if the event con? tains only 1 word
x*	Equivalent to `x-$'
x-	Equivalent to `x*', but omitting the last word (`$')

Selected words are inserted into the command line separated by single blanks. For example, the `diff' command in the previous example might

have been typed as `diff !!:1.old !!:1' (using `:1' to select the first argument from the previous event) or `diff !-2:2 !-2:1' to select and swap the arguments from the `cp' command. If we didn't care about the order of the `diff' we might have said `diff !-2:1-2' or simply `diff !-2:*'. The `cp' command might have been written `cp wumpus.man !#:1.old', using `#' to refer to the current event. `!n:- hurkle.man' would reuse the first two words from the `nroff' command to say `nroff -man hurkle.man'.

The `:' separating the event specification from the word designator can be omitted if the argument selector begins with a `^', `$', `*', `%' or `-'. For example, our `diff' command might have been `diff !!^.old !!^' or, equivalently, `diff !!$.old !!$'. However, if `!!' is abbre? viated `!', an argument selector beginning with `-' will be interpreted as an event specification.

A history reference may have a word designator but no event specifica? tion. It then references the previous command. Continuing our `diff' example, we could have said simply `diff !^.old !^' or, to get the ar? guments in the opposite order, just `diff !*'.

The word or words in a history reference can be edited, or ``modi? fied'', by following it with one or more modifiers, each preceded by a `:':

- h Remove a trailing pathname component, leaving the head.
- t Remove all leading pathname components, leaving the tail.
- r Remove a filename extension `.xxx', leaving the root name.
- e Remove all but the extension.
- u Uppercase the first lowercase letter.
- l Lowercase the first uppercase letter.
- s/l/r/ Substitute l for r. l is simply a string like r, not a regular expression as in the eponymous ed(1) command. Any character may be used as the delimiter in place of `/'; a `\' can be used to quote the delimiter inside l and r. The character `&' in the r is replaced by l; `\' also quotes `&'. If l is empty (``''), the l from a previous substitu?

tion or the s from a previous search or event number in event specification is used. The trailing delimiter may be omitted if it is immediately followed by a newline.

& Repeat the previous substitution.

g Apply the following modifier once to each word.

a (+) Apply the following modifier as many times as possible to a single word. `a' and `g' can be used together to apply a modifier globally. With the `s' modifier, only the pat? terns contained in the original word are substituted, not patterns that contain any substitution result.

p Print the new command line but do not execute it.

q Quote the substituted words, preventing further substitu? tions.

Q Same as q but in addition preserve empty variables as a string containing a NUL. This is useful to preserve posi? tional arguments for example:

```
> set args=('arg 1' '' 'arg 3')
> tcsh -f -c 'echo ${#argv}' $args:gQ
3
```

x Like q, but break into words at blanks, tabs and newlines.

Modifiers are applied to only the first modifiable word (unless `g' is used). It is an error for no word to be modifiable.

For example, the `diff' command might have been written as `diff wum? pus.man.old !#^:r', using `:r' to remove `.old' from the first argument on the same line (`!#^'). We could say `echo hello out there', then `echo !*:u' to capitalize `hello', `echo !*:au' to say it out loud, or `echo !*:agu' to really shout. We might follow `mail -s "I forgot my password" rot' with `!:s/rot/root' to correct the spelling of `root' (but see Spelling correction for a different approach).

There is a special abbreviation for substitutions. `^', when it is the first character on an input line, is equivalent to `!:s^'. Thus we might have said `^rot^root' to make the spelling correction in the pre? vious example. This is the only history substitution which does not

explicitly begin with `!'.

(+) In csh as such, only one modifier may be applied to each history or variable expansion. In tcsh, more than one may be used, for example

```
% mv wumpus.man /usr/man/man1/wumpus.1
% man !$:t:r
man wumpus
```

In csh, the result would be `wumpus.1:r'. A substitution followed by a colon may need to be insulated from it with braces:

```
> mv a.out /usr/games/wumpus
> setenv PATH !$:h:$PATH
Bad ! modifier: $.
> setenv PATH !{-2$:h}:$PATH
setenv PATH /usr/games:/bin:/usr/bin:.
```

The first attempt would succeed in csh but fails in tcsh, because tcsh expects another modifier after the second colon rather than `$'.

Finally, history can be accessed through the editor as well as through the substitutions just described. The up- and down-history, history-search-backward and -forward, i-search-back and -fwd, vi-search-back and -fwd, copy-prev-word and insert-last-word editor commands search for events in the history list and copy them into the input buffer.

The toggle-literal-history editor command switches between the expanded and literal forms of history lines in the input buffer. expand-history and expand-line expand history substitutions in the current word and in the entire input buffer respectively.

Alias substitution

The shell maintains a list of aliases which can be set, unset and printed by the alias and unalias commands. After a command line is parsed into simple commands (see Commands) the first word of each com? mand, left-to-right, is checked to see if it has an alias. If so, the first word is replaced by the alias. If the alias contains a history reference, it undergoes History substitution (q.v.) as though the orig? inal command were the previous input line. If the alias does not con? tain a history reference, the argument list is left untouched.

Thus if the alias for `ls' were `ls -l' the command `ls /usr' would be? come `ls -l /usr', the argument list here being undisturbed. If the alias for `lookup' were `grep !^ /etc/passwd' then `lookup bill' would become `grep bill /etc/passwd'. Aliases can be used to introduce parser metasyntax. For example, `alias print 'pr \!* | lpr'' defines a ``command'' (`print') which pr(1)s its arguments to the line printer.

Alias substitution is repeated until the first word of the command has no alias. If an alias substitution does not change the first word (as in the previous example) it is flagged to prevent a loop. Other loops are detected and cause an error.

Some aliases are referred to by the shell; see Special aliases.

Variable substitution

The shell maintains a list of variables, each of which has as value a list of zero or more words. The values of shell variables can be dis? played and changed with the set and unset commands. The system main? tains its own list of ``environment'' variables. These can be dis? played and changed with printenv, setenv and unsetenv.

(+) Variables may be made read-only with `set -r' (q.v.). Read-only variables may not be modified or unset; attempting to do so will cause an error. Once made read-only, a variable cannot be made writable, so `set -r' should be used with caution. Environment variables cannot be made read-only.

Some variables are set by the shell or referred to by it. For in? stance, the argv variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways. Some of the variables referred to by the shell are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the verbose variable is a toggle which causes command input to be echoed. The -v command line option sets this variable. Special shell variables lists all variables which are referred to by the shell.

Other operations treat variables numerically. The `@' command permits numeric calculations to be performed and the result assigned to a vari? able. Variable values are, however, always represented as (zero or

more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multi-word values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by `$' characters. This expansion can be prevented by preceding the `$' with a `\' except within `"'s where it always occurs, and within `'"s where it never oc? curs. Strings quoted by ``' are interpreted later (see Command substi? tution below) so `$' substitution does not occur there until later, if at all. A `$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and en? tire argument list are expanded together. It is thus possible for the first (command) word (to this point) to generate more than one word, the first of which becomes the command name, and the rest of which be? come arguments.

Unless enclosed in `"' or given the `:q' modifier the results of vari? able substitution may eventually be command and filename substituted. Within `"', a variable whose value consists of multiple words expands to a (portion of a) single word, with the words of the variable's value separated by blanks. When the `:q' modifier is applied to a substitu? tion the variable will expand to multiple words with each word sepa? rated by a blank and quoted to prevent later command or filename sub? stitution.

The following metasequences are provided for introducing variable val? ues into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name

${name} Substitutes the words of the value of variable name, each sepa? rated by a blank. Braces insulate name from following charac? ters which would otherwise be part of it. Shell variables have names consisting of letters and digits starting with a letter.

The underscore character is considered a letter. If name is not a shell variable, but is set in the environment, then that value is returned (but some of the other forms given below are not available in this case).

$name[selector]

${name[selector]}

Substitutes only the selected words from the value of name. The selector is subjected to `$' substitution and may consist of a single number or two numbers separated by a `-'. The first word of a variable's value is numbered `1'. If the first number of a range is omitted it defaults to `1'. If the last member of a range is omitted it defaults to `$#name'. The se? lector `*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$0 Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number

${number}

Equivalent to `$argv[number]'.

$* Equivalent to `$argv', which is equivalent to `$argv[*]'.

The `:' modifiers described under History substitution, except for `:p', can be applied to the substitutions above. More than one may be used. (+) Braces may be needed to insulate a variable substitution from a literal colon just as with History substitution (q.v.); any mod? ifiers must appear within the braces.

The following substitutions can not be modified with `:' modifiers.

$?name

${?name}

Substitutes the string `1' if name is set, `0' if it is not.

$?0 Substitutes `1' if the current input filename is known, `0' if it is not. Always `0' in interactive shells.

$#name

${#name}

Substitutes the number of words in name.

$# Equivalent to `$#argv'. (+)

$%name

${%name}

Substitutes the number of characters in name. (+)

$%number

${%number}

Substitutes the number of characters in $argv[number]. (+)

$? Equivalent to `$status'. (+)

$$ Substitutes the (decimal) process number of the (parent) shell.

$! Substitutes the (decimal) process number of the last background process started by this shell. (+)

$_ Substitutes the command line of the last command executed. (+)

$< Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script. (+) While csh always quotes $<, as if it were equivalent to `$<:q', tcsh does not. Furthermore, when tcsh is waiting for a line to be typed the user may type an interrupt to interrupt the sequence into which the line is to be substituted, but csh does not allow this.

The editor command expand-variables, normally bound to `^X-$', can be used to interactively expand individual variables.

Command, filename and directory stack substitution

The remaining substitutions are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in ``'. The output from such a command is broken into separate words at blanks, tabs and newlines, and null words are discarded. The output is vari?

able and command substituted and put in place of the original string. Command substitutions inside double quotes (`"') retain blanks and tabs; only newlines force new words. The single final newline does not force a new word in any case. It is thus possible for a command sub? stitution to yield only part of a word, even if the command outputs a complete line.

By default, the shell since version 6.12 replaces all newline and car? riage return characters in the command by spaces. If this is switched off by unsetting csubstnonl, newlines separate commands as usual.

Filename substitution

If a word contains any of the characters `*', `?', `[' or `{' or begins with the character `~' it is a candidate for filename substitution, also known as ``globbing''. This word is then regarded as a pattern (``glob-pattern''), and replaced with an alphabetically sorted list of file names which match the pattern.

In matching filenames, the character `.' at the beginning of a filename or immediately following a `/', as well as the character `/' must be matched explicitly (unless either globdot or globstar or both are set(+)). The character `*' matches any string of characters, including the null string. The character `?' matches any single character. The sequence `[...]' matches any one of the characters enclosed. Within `[...]', a pair of characters separated by `-' matches any character lexically between the two.

(+) Some glob-patterns can be negated: The sequence `[^...]' matches any single character not specified by the characters and/or ranges of characters in the braces.

An entire glob-pattern can also be negated with `^':

```
> echo *
bang crash crunch ouch
> echo ^cr*
bang ouch
```

Glob-patterns which do not use `?', `*', or `[]' or which use `{}' or `~' (below) are not negated correctly.

The metanotation `a{b,c,d}e' is a shorthand for `abe ace ade'. Left-to-right order is preserved: `/usr/source/s1/{oldls,ls}.c' expands to `/usr/source/s1/oldls.c /usr/source/s1/ls.c'. The results of matches are sorted separately at a low level to preserve this order: `../{memo,*box}' might expand to `../memo ../box ../mbox'. (Note that `memo' was not sorted with the results of matching `*box'.) It is not an error when this construct expands to files which do not exist, but it is possible to get an error from a command to which the expanded list is passed. This construct may be nested. As a special case the words `{', `}' and `{}' are passed undisturbed.

The character `~' at the beginning of a filename refers to home direc? tories. Standing alone, i.e., `~', it expands to the invoker's home directory as reflected in the value of the home shell variable. When followed by a name consisting of letters, digits and `-' characters the shell searches for a user with that name and substitutes their home di? rectory; thus `~ken' might expand to `/usr/ken' and `~ken/chmach' to `/usr/ken/chmach'. If the character `~' is followed by a character other than a letter or `/' or appears elsewhere than at the beginning of a word, it is left undisturbed. A command like `setenv MANPATH /usr/man:/usr/local/man:~/lib/man' does not, therefore, do home direc? tory substitution as one might hope.

It is an error for a glob-pattern containing `*', `?', `[' or `~', with or without `^', not to match any files. However, only one pattern in a list of glob-patterns must match a file (so that, e.g., `rm *.a *.c *.o' would fail only if there were no files in the current directory ending in `.a', `.c', or `.o'), and if the nonomatch shell variable is set a pattern (or list of patterns) which matches nothing is left un? changed rather than causing an error.

The globstar shell variable can be set to allow `**' or `***' as a file glob pattern that matches any string of characters including `/', re? cursively traversing any existing sub-directories. For example, `ls **.c' will list all the .c files in the current directory tree. If used by itself, it will match zero or more sub-directories (e.g. `ls

/usr/include/**/time.h' will list any file named `time.h' in the /usr/include directory tree; `ls /usr/include/**time.h' will match any file in the /usr/include directory tree ending in `time.h'; and `ls /usr/include/**time**.h' will match any .h file with `time' either in a subdirectory name or in the filename itself). To prevent problems with recursion, the `**' glob-pattern will not descend into a symbolic link containing a directory. To override this, use `***' (+)

The noglob shell variable can be set to prevent filename substitution, and the expand-glob editor command, normally bound to `^X-*', can be used to interactively expand individual filename substitutions.

Directory stack substitution (+)

The directory stack is a list of directories, numbered from zero, used by the pushd, popd and dirs builtin commands (q.v.). dirs can print, store in a file, restore and clear the directory stack at any time, and the savedirs and dirsfile shell variables can be set to store the di? rectory stack automatically on logout and restore it on login. The dirstack shell variable can be examined to see the directory stack and set to put arbitrary directories into the directory stack.

The character `=' followed by one or more digits expands to an entry in the directory stack. The special case `=-' expands to the last direc? tory in the stack. For example,

```
> dirs -v
0       /usr/bin
1       /usr/spool/uucp
2       /usr/accts/sys
> echo =1
/usr/spool/uucp
> echo =0/calendar
/usr/bin/calendar
> echo =-
/usr/accts/sys
```

The noglob and nonomatch shell variables and the expand-glob editor command apply to directory stack as well as filename substitutions.

Other substitutions (+)

There are several more transformations involving filenames, not strictly related to the above but mentioned here for completeness. Any filename may be expanded to a full path when the symlinks variable (q.v.) is set to `expand'. Quoting prevents this expansion, and the normalize-path editor command does it on demand. The normalize-command editor command expands commands in PATH into full paths on demand. Fi? nally, cd and pushd interpret `-' as the old working directory (equiva? lent to the shell variable owd). This is not a substitution at all, but an abbreviation recognized by only those commands. Nonetheless, it too can be prevented by quoting.

Commands

The next three sections describe how the shell executes commands and deals with their input and output.

Simple commands, pipelines and sequences

A simple command is a sequence of words, the first of which specifies the command to be executed. A series of simple commands joined by `|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next.

Simple commands and pipelines may be joined into sequences with `;', and will be executed sequentially. Commands and pipelines can also be joined into sequences with `||' or `&&', indicating, as in the C lan? guage, that the second is to be executed only if the first fails or succeeds respectively.

A simple command, pipeline or sequence may be placed in parentheses, `()', to form a simple command, which may in turn be a component of a pipeline or sequence. A command, pipeline or sequence can be executed without waiting for it to terminate by following it with an `&'.

Builtin and non-builtin command execution

Builtin commands are executed within the shell. If any component of a pipeline except the last is a builtin command, the pipeline is executed in a subshell.

Parenthesized commands are always executed in a subshell.

(cd; pwd); pwd

thus prints the home directory, leaving you where you were (printing this after the home directory), while

cd; pwd

leaves you in the home directory. Parenthesized commands are most of? ten used to prevent cd from affecting the current shell.

When a command to be executed is found not to be a builtin command the shell attempts to execute the command via execve(2). Each word in the variable path names a directory in which the shell will look for the command. If the shell is not given a -f option, the shell hashes the names in these directories into an internal table so that it will try an execve(2) in only a directory where there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. This hash? ing mechanism is not used:

1. If hashing is turned explicitly off via unhash.
2. If the shell was given a -f argument.
3. For each directory component of path which does not begin with a `/'.
4. If the command contains a `/'.

In the above four cases the shell concatenates each component of the path vector with the given command name to form a path name of a file which it then attempts to execute it. If execution is successful, the search stops.

If the file has execute permissions but is not an executable to the system (i.e., it is neither an executable binary nor a script that specifies its interpreter), then it is assumed to be a file containing shell commands and a new shell is spawned to read it. The shell spe? cial alias may be set to specify an interpreter other than the shell itself.

On systems which do not understand the `#!' script interpreter conven? tion the shell may be compiled to emulate it; see the version shell variable. If so, the shell checks the first line of the file to see if

it is of the form `#!interpreter arg ...'. If it is, the shell starts interpreter with the given args and feeds the file to it on standard input.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name Open file name (which is first variable, command and filename expanded) as the standard input.

<< word Read the shell input up to a line which is identical to word. word is not subjected to variable, filename or command substi? tution, and each input line is compared to word before any sub? stitutions are done on this input line. Unless a quoting `\', `"', `' or ``' appears in word variable and command substitu? tion is performed on the intervening lines, allowing `\' to quote `$', `\' and ``'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as stan? dard input.

> name

>! name

>& name

>&! name

The file name is used as standard output. If the file does not exist then it is created; if the file exists, it is truncated, its previous contents being lost.

If the shell variable noclobber is set, then the file must not exist or be a character special file (e.g., a terminal or `/dev/null') or an error results. This helps prevent acciden? tal destruction of files. In this case the `!' forms can be used to suppress this check. If notempty is given in noclob? ber, `>' is allowed on empty files; if ask is set, an in? teracive confirmation is presented, rather than an error.

The forms involving `&' route the diagnostic output into the specified file as well as the standard output. name is ex? panded in the same way as `<' input filenames are.

>> name

>>& name

>>! name

>>&! name

Like `>', but appends output to the end of name. If the shell variable noclobber is set, then it is an error for the file not to exist, unless one of the `!' forms is given.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The `<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. Note that the default standard in? put for a command run detached is not the empty file /dev/null, but the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see Jobs).

Diagnostic output may be directed through a pipe with the standard out? put. Simply use the form `|&' rather than just `|'.

The shell cannot presently redirect diagnostic output without also redirecting standard output, but `(command > output-file) >& error-file' is often an acceptable workaround. Either output-file or error-file may be `/dev/tty' to send output to the terminal.

Features

Having described how the shell accepts, parses and executes command lines, we now turn to a variety of its useful features.

Control flow

The shell contains a number of commands which can be used to regulate

the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the imple? mentation, restrict the placement of some of the commands.

The foreach, switch, and while statements, as well as the if-then-else form of the if statement, require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input when? ever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward gotos will succeed on non-seekable inputs.)

Expressions

The if, while and exit builtin commands use expressions with a common syntax. The expressions can include any of the operators described in the next three sections. Note that the @ builtin command (q.v.) has its own separate syntax.

Logical, arithmetical and comparison operators

These operators are similar to those of C and have the same precedence. They include

|| && | ^ & == != =~ !~ <= >=

< > << >> + - * / % ! ~ ( )

Here the precedence increases to the right, `==' `!=' `=~' and `!~', `<=' `>=' `<' and `>', `<<' and `>>', `+' and `-', `*' `/' and `%' be? ing, in groups, at the same level. The `==' `!=' `=~' and `!~' opera? tors compare their arguments as strings; all others operate on numbers. The operators `=~' and `!~' are like `!=' and `==' except that the right hand side is a glob-pattern (see Filename substitution) against which the left hand operand is matched. This reduces the need for use of the switch builtin command in shell scripts when all that is really needed is pattern matching.

Null or missing arguments are considered `0'. The results of all ex? pressions are strings, which represent decimal numbers. It is impor? tant to note that no two components of an expression can appear in the

same word; except when adjacent to components of expressions which are syntactically significant to the parser (`&' `|' `<' `>' `(' `)') they should be surrounded by spaces.

Command exit status

Commands can be executed in expressions and their exit status returned by enclosing them in braces (`{}'). Remember that the braces should be separated from the words of the command by spaces. Command executions succeed, returning true, i.e., `1', if the command exits with status 0, otherwise they fail, returning false, i.e., `0'. If more detailed sta? tus information is required then the command should be executed outside of an expression and the status shell variable examined.

File inquiry operators

Some of these operators perform true/false tests on files and related objects. They are of the form -op file, where op is one of

- r Read access
- w Write access
- x Execute access
- X Executable in the path or shell builtin, e.g., `-X ls' and `-X ls-F' are generally true, but `-X /bin/ls' is not (+)
- e Existence
- o Ownership
- z Zero size
- s Non-zero size (+)
- f Plain file
- d Directory
- l Symbolic link (+) *
- b Block special file (+)
- c Character special file (+)
- p Named pipe (fifo) (+) *
- S Socket special file (+) *
- u Set-user-ID bit is set (+)
- g Set-group-ID bit is set (+)
- k Sticky bit is set (+)

t file (which must be a digit) is an open file descriptor for a terminal device (+)

R Has been migrated (Convex only) (+)

L Applies subsequent operators in a multiple-operator test to a symbolic link rather than to the file to which the link points (+) *

file is command and filename expanded and then tested to see if it has the specified relationship to the real user. If file does not exist or is inaccessible or, for the operators indicated by `*', if the speci? fied file type does not exist on the current system, then all inquiries return false, i.e., `0'.

These operators may be combined for conciseness: `-xy file' is equiva? lent to `-x file && -y file'. (+) For example, `-fx' is true (returns `1') for plain executable files, but not for directories.

L may be used in a multiple-operator test to apply subsequent operators to a symbolic link rather than to the file to which the link points. For example, `-lLo' is true for links owned by the invoking user. Lr, Lw and Lx are always true for links and false for non-links. L has a different meaning when it is the last operator in a multiple-operator test; see below.

It is possible but not useful, and sometimes misleading, to combine op? erators which expect file to be a file with operators which do not (e.g., X and t). Following L with a non-file operator can lead to par? ticularly strange results.

Other operators return other information, i.e., not just `0' or `1'. (+) They have the same format as before; op may be one of

A Last file access time, as the number of seconds since the epoch

A: Like A, but in timestamp format, e.g., `Fri May 14 16:36:10 1993'

M Last file modification time

M: Like M, but in timestamp format

C Last inode modification time

C: Like C, but in timestamp format

D Device number

I Inode number

F Composite file identifier, in the form device:inode

L The name of the file pointed to by a symbolic link

N Number of (hard) links

P Permissions, in octal, without leading zero

P: Like P, with leading zero

Pmode Equivalent to `-P file & mode', e.g., `-P22 file' returns `22' if file is writable by group and other, `20' if by group only, and `0' if by neither

Pmode: Like Pmode, with leading zero

U Numeric userid

U: Username, or the numeric userid if the username is unknown

G Numeric groupid

G: Groupname, or the numeric groupid if the groupname is un? known

Z Size, in bytes

Only one of these operators may appear in a multiple-operator test, and it must be the last. Note that L has a different meaning at the end of and elsewhere in a multiple-operator test. Because `0' is a valid re? turn value for many of these operators, they do not return `0' when they fail: most return `-1', and F returns `:'.

If the shell is compiled with POSIX defined (see the version shell variable), the result of a file inquiry is based on the permission bits of the file and not on the result of the access(2) system call. For example, if one tests a file with -w whose permissions would ordinarily allow writing but which is on a file system mounted read-only, the test will succeed in a POSIX shell but fail in a non-POSIX shell.

File inquiry operators can also be evaluated with the filetest builtin command (q.v.) (+).

Jobs

The shell associates a job with each pipeline. It keeps a table of

current jobs, printed by the jobs command, and assigns them small inte?
ger numbers. When a job is started asynchronously with `&', the shell
prints a line which looks like

[1] 1234

indicating that the job which was started asynchronously was job number
1 and had one (top-level) process, whose process id was 1234.
If you are running a job and wish to do something else you may hit the
suspend key (usually `^Z'), which sends a STOP signal to the current
job. The shell will then normally indicate that the job has been `Sus?
pended' and print another prompt. If the listjobs shell variable is
set, all jobs will be listed like the jobs builtin command; if it is
set to `long' the listing will be in long format, like `jobs -l'. You
can then manipulate the state of the suspended job. You can put it in
the ``background'' with the bg command or run some other commands and
eventually bring the job back into the ``foreground'' with fg. (See
also the run-fg-editor editor command.) A `^Z' takes effect immedi?
ately and is like an interrupt in that pending output and unread input
are discarded when it is typed. The wait builtin command causes the
shell to wait for all background jobs to complete.
The `^]' key sends a delayed suspend signal, which does not generate a
STOP signal until a program attempts to read(2) it, to the current job.
This can usefully be typed ahead when you have prepared some commands
for a job which you wish to stop after it has read them. The `^Y' key
performs this function in csh(1); in tcsh, `^Y' is an editing command.
(+)
A job being run in the background stops if it tries to read from the
terminal. Background jobs are normally allowed to produce output, but
this can be disabled by giving the command `stty tostop'. If you set
this tty option, then background jobs will stop when they try to pro?
duce output like they do when they try to read input.
There are several ways to refer to jobs in the shell. The character
`%' introduces a job name. If you wish to refer to job number 1, you
can name it as `%1'. Just naming a job brings it to the foreground;

thus `%1' is a synonym for `fg %1', bringing job 1 back into the fore?
ground. Similarly, saying `%1 &' resumes job 1 in the background, just
like `bg %1'. A job can also be named by an unambiguous prefix of the
string typed in to start it: `%ex' would normally restart a suspended
ex(1) job, if there were only one suspended job whose name began with
the string `ex'. It is also possible to say `%?string' to specify a
job whose text contains string, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In out?
put pertaining to jobs, the current job is marked with a `+' and the
previous job with a `-'. The abbreviations `%+', `%', and (by analogy
with the syntax of the history mechanism) `%%' all refer to the current
job, and `%-' refers to the previous job.

The job control mechanism requires that the stty(1) option `new' be set
on some systems. It is an artifact from a `new' implementation of the
tty driver which allows generation of interrupt characters from the
keyboard to tell jobs to stop. See stty(1) and the setty builtin com?
mand for details on setting options in the new tty driver.

Status reporting

The shell learns immediately whenever a process changes state. It nor?
mally informs you whenever a job becomes blocked so that no further
progress is possible, but only right before it prints a prompt. This
is done so that it does not otherwise disturb your work. If, however,
you set the shell variable notify, the shell will notify you immedi?
ately of changes of status in background jobs. There is also a shell
command notify which marks a single process so that its status changes
will be immediately reported. By default notify marks the current
process; simply say `notify' after starting a background job to mark
it.

When you try to leave the shell while jobs are stopped, you will be
warned that `There are suspended jobs.' You may use the jobs command to
see what they are. If you do this or immediately try to exit again,
the shell will not warn you a second time, and the suspended jobs will
be terminated.

Automatic, periodic and timed events (+)

There are various ways to run commands and take other actions automati? cally at various times in the ``life cycle'' of the shell. They are summarized here, and described in detail under the appropriate Builtin commands, Special shell variables and Special aliases.

The sched builtin command puts commands in a scheduled-event list, to be executed by the shell at a given time.

The beepcmd, cwdcmd, periodic, precmd, postcmd, and jobcmd Special aliases can be set, respectively, to execute commands when the shell wants to ring the bell, when the working directory changes, every tpe? riod minutes, before each prompt, before each command gets executed, after each command gets executed, and when a job is started or is brought into the foreground.

The autologout shell variable can be set to log out or lock the shell after a given number of minutes of inactivity.

The mail shell variable can be set to check for new mail periodically.

The printexitvalue shell variable can be set to print the exit status of commands which exit with a status other than zero.

The rmstar shell variable can be set to ask the user, when `rm *' is typed, if that is really what was meant.

The time shell variable can be set to execute the time builtin command after the completion of any process that takes more than a given number of CPU seconds.

The watch and who shell variables can be set to report when selected users log in or out, and the log builtin command reports on those users at any time.

Native Language System support (+)

The shell is eight bit clean (if so compiled; see the version shell variable) and thus supports character sets needing this capability.

NLS support differs depending on whether or not the shell was compiled to use the system's NLS (again, see version). In either case, 7-bit ASCII is the default character code (e.g., the classification of which characters are printable) and sorting, and changing the LANG or

LC_CTYPE environment variables causes a check for possible changes in these respects.

When using the system's NLS, the setlocale(3) function is called to de? termine appropriate character code/classification and sorting (e.g., a 'en_CA.UTF-8' would yield "UTF-8" as a character code). This function typically examines the LANG and LC_CTYPE environment variables; refer to the system documentation for further details. When not using the system's NLS, the shell simulates it by assuming that the ISO 8859-1 character set is used whenever either of the LANG and LC_CTYPE vari? ables are set, regardless of their values. Sorting is not affected for the simulated NLS.

In addition, with both real and simulated NLS, all printable characters in the range \200-\377, i.e., those that have M-char bindings, are au? tomatically rebound to self-insert-command. The corresponding binding for the escape-char sequence, if any, is left alone. These characters are not rebound if the NOREBIND environment variable is set. This may be useful for the simulated NLS or a primitive real NLS which assumes full ISO 8859-1. Otherwise, all M-char bindings in the range \240-\377 are effectively undone. Explicitly rebinding the relevant keys with bindkey is of course still possible.

Unknown characters (i.e., those that are neither printable nor control characters) are printed in the format \nnn. If the tty is not in 8 bit mode, other 8 bit characters are printed by converting them to ASCII and using standout mode. The shell never changes the 7/8 bit mode of the tty and tracks user-initiated changes of 7/8 bit mode. NLS users (or, for that matter, those who want to use a meta key) may need to ex? plicitly set the tty in 8 bit mode through the appropriate stty(1) com? mand in, e.g., the ~/.login file.

OS variant support (+)

A number of new builtin commands are provided to support features in particular operating systems. All are described in detail in the Builtin commands section.

On systems that support TCF (aix-ibm370, aix-ps2), getspath and

setspath get and set the system execution path, getxvers and setxvers get and set the experimental version prefix and migrate migrates pro? cesses between sites. The jobs builtin prints the site on which each job is executing.

Under BS2000, bs2cmd executes commands of the underlying BS2000/OSD op? erating system.

Under Domain/OS, inlib adds shared libraries to the current environ? ment, rootnode changes the rootnode and ver changes the systype.

Under Mach, setpath is equivalent to Mach's setpath(1).

Under Masscomp/RTU and Harris CX/UX, universe sets the universe.

Under Harris CX/UX, ucb or att runs a command under the specified uni? verse.

Under Convex/OS, warp prints or sets the universe.

The VENDOR, OSTYPE and MACHTYPE environment variables indicate respec? tively the vendor, operating system and machine type (microprocessor class or machine model) of the system on which the shell thinks it is running. These are particularly useful when sharing one's home direc? tory between several types of machines; one can, for example,

```
set path = (~/bin.$MACHTYPE /usr/ucb /bin /usr/bin .)
```

in one's ~/.login and put executables compiled for each machine in the appropriate directory.

The version shell variable indicates what options were chosen when the shell was compiled.

Note also the newgrp builtin, the afsuser and echo_style shell vari? ables and the system-dependent locations of the shell's input files (see FILES).

Signal handling

Login shells ignore interrupts when reading the file ~/.logout. The shell ignores quit signals unless started with -q. Login shells catch the terminate signal, but non-login shells inherit the terminate behav? ior from their parents. Other signals have the values which the shell inherited from its parent.

In shell scripts, the shell's handling of interrupt and terminate sig?

nals can be controlled with onintr, and its handling of hangups can be controlled with hup and nohup.

The shell exits on a hangup (see also the logout shell variable). By default, the shell's children do too, but the shell does not send them a hangup when it exits. hup arranges for the shell to send a hangup to a child when it exits, and nohup sets a child to ignore hangups.

Terminal management (+)

The shell uses three different sets of terminal (``tty'') modes: `edit', used when editing, `quote', used when quoting literal charac? ters, and `execute', used when executing commands. The shell holds some settings in each mode constant, so commands which leave the tty in a confused state do not interfere with the shell. The shell also matches changes in the speed and padding of the tty. The list of tty modes that are kept constant can be examined and modified with the setty builtin. Note that although the editor uses CBREAK mode (or its equivalent), it takes typed-ahead characters anyway.

The echotc, settc and telltc commands can be used to manipulate and de? bug terminal capabilities from the command line.

On systems that support SIGWINCH or SIGWINDOW, the shell adapts to win? dow resizing automatically and adjusts the environment variables LINES and COLUMNS if set. If the environment variable TERMCAP contains li# and co# fields, the shell adjusts them to reflect the new window size.

REFERENCE

The next sections of this manual describe all of the available Builtin commands, Special aliases and Special shell variables.

Builtin commands

%job A synonym for the fg builtin command.

%job & A synonym for the bg builtin command.

: Does nothing, successfully.

@

@ name = expr

@ name[index] = expr

@ name++|--

@ name[index]++|--

The first form prints the values of all shell variables.

The second form assigns the value of expr to name. The third form assigns the value of expr to the index'th component of name; both name and its index'th component must already exist.

expr may contain the operators `*', `+', etc., as in C. If expr contains `<', `>', `&' or `' then at least that part of expr must be placed within `()'. Note that the syntax of expr has nothing to do with that described under Expressions.

The fourth and fifth forms increment (`++') or decrement (`--') name or its index'th component.

The space between `@' and name is required. The spaces between name and `=' and between `=' and expr are optional. Components of expr must be separated by spaces.

alias [name [wordlist]]

Without arguments, prints all aliases. With name, prints the alias for name. With name and wordlist, assigns wordlist as the alias of name. wordlist is command and filename substi? tuted. name may not be `alias' or `unalias'. See also the un? alias builtin command.

alloc Shows the amount of dynamic memory acquired, broken down into used and free memory. With an argument shows the number of free and used blocks in each size category. The categories start at size 8 and double at each step. This command's output may vary across system types, because systems other than the VAX may use a different memory allocator.

bg [%job ...]

Puts the specified jobs (or, without arguments, the current job) into the background, continuing each if it is stopped. job may be a number, a string, `', `%', `+' or `-' as described under Jobs.

bindkey [-l|-d|-e|-v|-u] (+)

bindkey [-a] [-b] [-k] [-r] [--] key (+)

bindkey [-a] [-b] [-k] [-c|-s] [--] key command (+)

Without options, the first form lists all bound keys and the editor command to which each is bound, the second form lists the editor command to which key is bound and the third form binds the editor command command to key. Options include:

-l Lists all editor commands and a short description of each.

-d Binds all keys to the standard bindings for the default editor, as per -e and -v below.

-e Binds all keys to emacs(1)-style bindings. Unsets vimode.

-v Binds all keys to vi(1)-style bindings. Sets vimode.

-a Lists or changes key-bindings in the alternative key map. This is the key map used in vimode command mode.

-b key is interpreted as a control character written ^character (e.g., `^A') or C-character (e.g., `C-A'), a meta character written M-character (e.g., `M-A'), a function key written F-string (e.g., `F-string'), or an extended prefix key written X-character (e.g., `X-A').

-k key is interpreted as a symbolic arrow key name, which may be one of `down', `up', `left' or `right'.

-r Removes key's binding. Be careful: `bindkey -r' does not bind key to self-insert-command (q.v.), it unbinds key completely.

-c command is interpreted as a builtin or external command instead of an editor command.

-s command is taken as a literal string and treated as terminal input when key is typed. Bound keys in command are themselves reinterpreted, and this continues for ten levels of interpretation.

-- Forces a break from option processing, so the next word is taken as key even if it begins with '-'.

-u (or any invalid option)

Prints a usage message.

key may be a single character or a string. If a command is

bound to a string, the first character of the string is bound to sequence-lead-in and the entire string is bound to the com? mand.

Control characters in key can be literal (they can be typed by preceding them with the editor command quoted-insert, normally bound to `^V') or written caret-character style, e.g., `^A'. Delete is written `^?' (caret-question mark). key and command can contain backslashed escape sequences (in the style of Sys? tem V echo(1)) as follows:

\a Bell

\b Backspace

\e Escape

\f Form feed

\n Newline

\r Carriage return

\t Horizontal tab

\v Vertical tab

\nnn The ASCII character corresponding to the octal num? ber nnn

`\' nullifies the special meaning of the following character, if it has any, notably `\' and `^'.

bs2cmd bs2000-command (+)

Passes bs2000-command to the BS2000 command interpreter for ex? ecution. Only non-interactive commands can be executed, and it is not possible to execute any command that would overlay the image of the current process, like /EXECUTE or /CALL-PROCEDURE. (BS2000 only)

break Causes execution to resume after the end of the nearest enclos? ing foreach or while. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw Causes a break from a switch, resuming after the endsw.

builtins (+)

Prints the names of all builtin commands.

bye (+) A synonym for the logout builtin command. Available only if the shell was so compiled; see the version shell variable.

case label:

A label in a switch statement as discussed below.

cd [-p] [-l] [-n|-v] [l--] [name]

If a directory name is given, changes the shell's working di? rectory to name. If not, changes to home, unless the cdtohome variable is not set, in which case a name is required. If name is `-' it is interpreted as the previous working directory (see Other substitutions). (+) If name is not a subdirectory of the current directory (and does not begin with `/', `./' or `../'), each component of the variable cdpath is checked to see if it has a subdirectory name. Finally, if all else fails but name is a shell variable whose value begins with `/' or '.', then this is tried to see if it is a directory, and the -p option is implied.

With -p, prints the final directory stack, just like dirs. The -l, -n and -v flags have the same effect on cd as on dirs, and they imply -p. (+) Using -- forces a break from option pro? cessing so the next word is taken as the directory name even if it begins with '-'. (+)

See also the implicitcd and cdtohome shell variables.

chdir A synonym for the cd builtin command.

complete [command [word/pattern/list[:select]/[[suffix]/] ...]] (+)

Without arguments, lists all completions. With command, lists completions for command. With command and word etc., defines completions.

command may be a full command name or a glob-pattern (see File? name substitution). It can begin with `-' to indicate that completion should be used only when command is ambiguous.

word specifies which word relative to the current word is to be completed, and may be one of the following:

c Current-word completion. pattern is a glob-pattern which must match the beginning of the current word on the command line. pattern is ignored when completing the current word.

C Like c, but includes pattern when completing the cur? rent word.

n Next-word completion. pattern is a glob-pattern which must match the beginning of the previous word on the command line.

N Like n, but must match the beginning of the word two before the current word.

p Position-dependent completion. pattern is a numeric range, with the same syntax used to index shell vari? ables, which must include the current word.

list, the list of possible completions, may be one of the fol? lowing:

a Aliases

b Bindings (editor commands)

c Commands (builtin or external commands)

C External commands which begin with the supplied path prefix

d Directories

D Directories which begin with the supplied path pre? fix

e Environment variables

f Filenames

F Filenames which begin with the supplied path prefix

g Groupnames

j Jobs

l Limits

n Nothing

s Shell variables

S Signals

t Plain (``text'') files

T Plain (``text'') files which begin with the sup?
plied path prefix

v Any variables

u Usernames

x Like n, but prints select when list-choices is
used.

X Completions

$var Words from the variable var

(...) Words from the given list

`...` Words from the output of command

select is an optional glob-pattern. If given, words from only list that match select are considered and the fignore shell variable is ignored. The last three types of completion may not have a select pattern, and x uses select as an explanatory message when the list-choices editor command is used.

suffix is a single character to be appended to a successful completion. If null, no character is appended. If omitted (in which case the fourth delimiter can also be omitted), a slash is appended to directories and a space to other words.

command invoked from `...` version has additional environment variable set, the variable name is COMMAND_LINE and contains (as its name indicates) contents of the current (already typed in) command line. One can examine and use contents of the COMMAND_LINE variable in her custom script to build more so? phisticated completions (see completion for svn(1) included in this package).

Now for some examples. Some commands take only directories as arguments, so there's no point completing plain files.

> complete cd 'p/1/d/'

completes only the first word following `cd' (`p/1') with a di? rectory. p-type completion can also be used to narrow down command completion:

> co[^D]

complete compress

> complete -co* 'p/0/(compress)/'

> co[^D]

> compress

This completion completes commands (words in position 0, `p/0') which begin with `co' (thus matching `co*') to `compress' (the only word in the list). The leading `-' indicates that this completion is to be used with only ambiguous commands.

> complete find 'n/-user/u/'

is an example of n-type completion. Any word following `find' and immediately following `-user' is completed from the list of users.

> complete cc 'c/-I/d/'

demonstrates c-type completion. Any word following `cc' and beginning with `-I' is completed as a directory. `-I' is not taken as part of the directory because we used lowercase c. Different lists are useful with different commands.

> complete alias 'p/1/a/'

> complete man 'p/*/c/'

> complete set 'p/1/s/'

> complete true 'p/1/x:Truth has no options./'

These complete words following `alias' with aliases, `man' with commands, and `set' with shell variables. `true' doesn't have any options, so x does nothing when completion is attempted and prints `Truth has no options.' when completion choices are listed.

Note that the man example, and several other examples below, could just as well have used 'c/*' or 'n/*' as 'p/*'.

Words can be completed from a variable evaluated at completion time,

> complete ftp 'p/1/$hostnames/'

> set hostnames = (rtfm.mit.edu tesla.ee.cornell.edu)

> ftp [^D]

rtfm.mit.edu tesla.ee.cornell.edu

> ftp [^C]

> set hostnames = (rtfm.mit.edu tesla.ee.cornell.edu

uunet.uu.net)

> ftp [^D]

rtfm.mit.edu tesla.ee.cornell.edu uunet.uu.net

or from a command run at completion time:

> complete kill 'p/*/`ps | awk \{print\ \$1\}`/'

> kill -9 [^D]

23113 23377 23380 23406 23429 23529 23530 PID

Note that the complete command does not itself quote its argu? ments, so the braces, space and `$' in `{print $1}' must be quoted explicitly.

One command can have multiple completions:

> complete dbx 'p/2/(core)/' 'p/*/c/'

completes the second argument to `dbx' with the word `core' and all other arguments with commands. Note that the positional completion is specified before the next-word completion. Be? cause completions are evaluated from left to right, if the next-word completion were specified first it would always match and the positional completion would never be executed. This is a common mistake when defining a completion.

The select pattern is useful when a command takes files with only particular forms as arguments. For example,

> complete cc 'p/*/f:*.[cao]/'

completes `cc' arguments to files ending in only `.c', `.a', or `.o'. select can also exclude files, using negation of a glob-pattern as described under Filename substitution. One might use

> complete rm 'p/*/f:^*.{c,h,cc,C,tex,1,man,l,y}/'

to exclude precious source code from `rm' completion. Of course, one could still type excluded names manually or over?

ride the completion mechanism using the complete-word-raw or list-choices-raw editor commands (q.v.).

The `C', `D', `F' and `T' lists are like `c', `d', `f' and `t' respectively, but they use the select argument in a different way: to restrict completion to files beginning with a particu? lar path prefix. For example, the Elm mail program uses `=' as an abbreviation for one's mail directory. One might use

```
> complete elm c@=@F:$HOME/Mail/@
```

to complete `elm -f =' as if it were `elm -f ~/Mail/'. Note that we used `@' instead of `/' to avoid confusion with the se? lect argument, and we used `$HOME' instead of `~' because home directory substitution works at only the beginning of a word. suffix is used to add a nonstandard suffix (not space or `/' for directories) to completed words.

```
> complete finger 'c/*@/$hostnames/' 'p/1/u/@'
```

completes arguments to `finger' from the list of users, appends an `@', and then completes after the `@' from the `hostnames' variable. Note again the order in which the completions are specified.

Finally, here's a complex example for inspiration:

```
> complete find \
'n/-name/f/' 'n/-newer/f/' 'n/-{,n}cpio/f/' \
?n/-exec/c/' 'n/-ok/c/' 'n/-user/u/' \
'n/-group/g/' 'n/-fstype/(nfs 4.2)/' \
'n/-type/(b c d f l p s)/' \
?c/-/(name newer cpio ncpio exec ok user \
group fstype type atime ctime depth inum \
ls mtime nogroup nouser perm print prune \
size xdev)/' \
'p/*/d/'
```

This completes words following `-name', `-newer', `-cpio' or `ncpio' (note the pattern which matches both) to files, words following `-exec' or `-ok' to commands, words following `user'

and `group' to users and groups respectively and words follow?
ing `-fstype' or `-type' to members of the given lists. It
also completes the switches themselves from the given list
(note the use of c-type completion) and completes anything not
otherwise completed to a directory. Whew.
Remember that programmed completions are ignored if the word
being completed is a tilde substitution (beginning with `~') or
a variable (beginning with `$'). See also the uncomplete
builtin command.

continue

Continues execution of the nearest enclosing while or foreach.
The rest of the commands on the current line are executed.

default:

Labels the default case in a switch statement. It should come
after all case labels.

dirs [-l] [-n|-v]

dirs -S|-L [filename] (+)

dirs -c (+)

The first form prints the directory stack. The top of the
stack is at the left and the first directory in the stack is
the current directory. With -l, `~' or `~name' in the output
is expanded explicitly to home or the pathname of the home di?
rectory for user name. (+) With -n, entries are wrapped before
they reach the edge of the screen. (+) With -v, entries are
printed one per line, preceded by their stack positions. (+)
If more than one of -n or -v is given, -v takes precedence. -p
is accepted but does nothing.
With -S, the second form saves the directory stack to filename
as a series of cd and pushd commands. With -L, the shell
sources filename, which is presumably a directory stack file
saved by the -S option or the savedirs mechanism. In either
case, dirsfile is used if filename is not given and ~/.cshdirs
is used if dirsfile is unset.

Note that login shells do the equivalent of `dirs -L' on startup and, if savedirs is set, `dirs -S' before exiting. Be? cause only ~/.tcshrc is normally sourced before ~/.cshdirs, dirsfile should be set in ~/.tcshrc rather than ~/.login.

The last form clears the directory stack.

echo [-n] word ...

Writes each word to the shell's standard output, separated by spaces and terminated with a newline. The echo_style shell variable may be set to emulate (or not) the flags and escape sequences of the BSD and/or System V versions of echo; see echo(1).

echotc [-sv] arg ... (+)

Exercises the terminal capabilities (see termcap(5)) in args. For example, 'echotc home' sends the cursor to the home posi? tion, 'echotc cm 3 10' sends it to column 3 and row 10, and 'echotc ts 0; echo "This is a test."; echotc fs' prints "This is a test." in the status line.

If arg is 'baud', 'cols', 'lines', 'meta' or 'tabs', prints the value of that capability ("yes" or "no" indicating that the terminal does or does not have that capability). One might use this to make the output from a shell script less verbose on slow terminals, or limit command output to the number of lines on the screen:

```
> set history=`echotc lines`
> @ history--
```

Termcap strings may contain wildcards which will not echo cor? rectly. One should use double quotes when setting a shell variable to a terminal capability string, as in the following example that places the date in the status line:

```
> set tosl="`echotc ts 0`"
> set frsl="`echotc fs`"
> echo -n "$tosl";date; echo -n "$frsl"
```

With -s, nonexistent capabilities return the empty string

rather than causing an error. With -v, messages are verbose.

else

end

endif

endsw See the description of the foreach, if, switch, and while statements below.

eval arg ...

Treats the arguments as input to the shell and executes the re? sulting command(s) in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, because parsing occurs before these substitutions. See tset(1) for a sample use of eval.

exec command

Executes the specified command in place of the current shell.

exit [expr]

The shell exits either with the value of the specified expr (an expression, as described under Expressions) or, without expr, with the value 0.

fg [%job ...]

Brings the specified jobs (or, without arguments, the current job) into the foreground, continuing each if it is stopped. job may be a number, a string, `', `%', `+' or `-' as described under Jobs. See also the run-fg-editor editor command.

filetest -op file ... (+)

Applies op (which is a file inquiry operator as described under File inquiry operators) to each file and returns the results as a space-separated list.

foreach name (wordlist)

...

end Successively sets the variable name to each member of wordlist and executes the sequence of commands between this command and the matching end. (Both foreach and end must appear alone on separate lines.) The builtin command continue may be used to

continue the loop prematurely and the builtin command break to terminate it prematurely. When this command is read from the terminal, the loop is read once prompting with `foreach? ' (or prompt2) before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

getspath (+)

Prints the system execution path. (TCF only)

getxvers (+)

Prints the experimental version prefix. (TCF only)

glob wordlist

Like echo, but the `-n' parameter is not recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word

word is filename and command-substituted to yield a string of the form `label'. The shell rewinds its input as much as pos? sible, searches for a line of the form `label:', possibly pre? ceded by blanks or tabs, and continues execution after that line.

hashstat

Prints a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding exec's). An exec is attempted for each component of the path where the hash function indicates a possible hit, and in each component which does not begin with a `/'.

On machines without vfork(2), prints only the number and size of hash buckets.

history [-hTr] [n]

history -S|-L|-M [filename] (+)

history -c (+)

The first form prints the history event list. If n is given

only the n most recent events are printed or saved. With -h, the history list is printed without leading numbers. If -T is specified, timestamps are printed also in comment form. (This can be used to produce files suitable for loading with 'history -L' or 'source -h'.) With -r, the order of printing is most recent first rather than oldest first.

With -S, the second form saves the history list to filename. If the first word of the savehist shell variable is set to a number, at most that many lines are saved. If the second word of savehist is set to `merge', the history list is merged with the existing history file instead of replacing it (if there is one) and sorted by time stamp. (+) Merging is intended for an environment like the X Window System with several shells in si? multaneous use. If the second word of savehist is `merge' and the third word is set to `lock', the history file update will be serialized with other shell sessions that would possibly like to merge history at exactly the same time.

With -L, the shell appends filename, which is presumably a his? tory list saved by the -S option or the savehist mechanism, to the history list. -M is like -L, but the contents of filename are merged into the history list and sorted by timestamp. In either case, histfile is used if filename is not given and ~/.history is used if histfile is unset. `history -L' is ex? actly like 'source -h' except that it does not require a file? name.

Note that login shells do the equivalent of `history -L' on startup and, if savehist is set, `history -S' before exiting. Because only ~/.tcshrc is normally sourced before ~/.history, histfile should be set in ~/.tcshrc rather than ~/.login.

If histlit is set, the first and second forms print and save the literal (unexpanded) form of the history list.

The last form clears the history list.

hup [command] (+)

With command, runs command such that it will exit on a hangup signal and arranges for the shell to send it a hangup signal when the shell exits. Note that commands may set their own re? sponse to hangups, overriding hup. Without an argument, causes the non-interactive shell only to exit on a hangup for the re? mainder of the script. See also Signal handling and the nohup builtin command.

if (expr) command

If expr (an expression, as described under Expressions) evalu? ates true, then command is executed. Variable substitution on command happens early, at the same time it does for the rest of the if command. command must be a simple command, not an alias, a pipeline, a command list or a parenthesized command list, but it may have arguments. Input/output redirection oc? curs even if expr is false and command is thus not executed; this is a bug.

if (expr) then

...

else if (expr2) then

...

else

...

endif If the specified expr is true then the commands to the first else are executed; otherwise if expr2 is true then the commands to the second else are executed, etc. Any number of else-if pairs are possible; only one endif is needed. The else part is likewise optional. (The words else and endif must appear at the beginning of input lines; the if must appear alone on its input line or after an else.)

inlib shared-library ... (+)

Adds each shared-library to the current environment. There is no way to remove a shared library. (Domain/OS only)

jobs [-l]

Lists the active jobs. With -l, lists process IDs in addition to the normal information. On TCF systems, prints the site on which each job is executing.

kill [-s signal] %job|pid ...

kill -l The first and second forms sends the specified signal (or, if none is given, the TERM (terminate) signal) to the specified jobs or processes. job may be a number, a string, `', `%', `+' or `-' as described under Jobs. Signals are either given by number or by name (as given in /usr/include/signal.h, stripped of the prefix `SIG'). There is no default job; saying just `kill' does not send a signal to the current job. If the sig? nal being sent is TERM (terminate) or HUP (hangup), then the job or process is sent a CONT (continue) signal as well. The third form lists the signal names.

limit [-h] [resource [maximum-use]]

Limits the consumption by the current process and each process it creates to not individually exceed maximum-use on the speci? fied resource. If no maximum-use is given, then the current limit is printed; if no resource is given, then all limitations are given. If the -h flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceil? ing on the values of the current limits. Only the super-user may raise the hard limits, but a user may lower or raise the current limits within the legal range.

Controllable resources currently include (if supported by the OS):

cputime

the maximum number of cpu-seconds to be used by each process

filesize

the largest single file which can be created

datasize

the maximum growth of the data+stack region via sbrk(2)

beyond the end of the program text

stacksize

the maximum size of the automatically-extended stack re?

gion

coredumpsize

the size of the largest core dump that will be created

memoryuse

the maximum amount of physical memory a process may have

allocated to it at a given time

NOTE: Changing this value has no effect. Support has

been removed from Linux kernel v2.6 and newer.

vmemoryuse

the maximum amount of virtual memory a process may have

allocated to it at a given time (address space)

vmemoryuse

the maximum amount of virtual memory a process may have

allocated to it at a given time

heapsize

the maximum amount of memory a process may allocate per

brk() system call

descriptors or openfiles

the maximum number of open files for this process

pseudoterminals

the maximum number of pseudo-terminals for this user

kqueues

the maximum number of kqueues allocated for this process

concurrency

the maximum number of threads for this process

memorylocked

the maximum size which a process may lock into memory

using mlock(2)

maxproc

the maximum number of simultaneous processes for this

user id

maxthread

the maximum number of simultaneous threads (lightweight processes) for this user id

threads

the maximum number of threads for this process

sbsize the maximum size of socket buffer usage for this user

swapsize

the maximum amount of swap space reserved or used for this user

maxlocks

the maximum number of locks for this user

posixlocks

the maximum number of POSIX advisory locks for this user

maxsignal

the maximum number of pending signals for this user

maxmessage

the maximum number of bytes in POSIX mqueues for this user

maxnice

the maximum nice priority the user is allowed to raise mapped from [19...-20] to [0...39] for this user

maxrtprio

the maximum realtime priority for this user maxrttime

the timeout for RT tasks in microseconds for this user.

maximum-use may be given as a (floating point or integer) num? ber followed by a scale factor. For all limits other than cputime the default scale is `k' or `kilobytes' (1024 bytes); a scale factor of `m' or `megabytes' or `g' or `gigabytes' may also be used. For cputime the default scaling is `seconds', while `m' for minutes or `h' for hours, or a time of the form `mm:ss' giving minutes and seconds may be used.

If maximum-use is `unlimited', then the limitation on the

specified resource is removed (this is equivalent to the un? limit builtin command).

For both resource names and scale factors, unambiguous prefixes of the names suffice.

log (+) Prints the watch shell variable and reports on each user indi? cated in watch who is logged in, regardless of when they last logged in. See also watchlog.

login Terminates a login shell, replacing it with an instance of /bin/login. This is one way to log off, included for compati? bility with sh(1).

logout Terminates a login shell. Especially useful if ignoreeof is set.

ls-F [-switch ...] [file ...] (+)

Lists files like `ls -F', but much faster. It identifies each type of special file in the listing with a special character:

/ Directory

* Executable

Block device

% Character device

| Named pipe (systems with named pipes only)

= Socket (systems with sockets only)

@ Symbolic link (systems with symbolic links only)

+ Hidden directory (AIX only) or context dependent (HP/UX only)

: Network special (HP/UX only)

If the listlinks shell variable is set, symbolic links are identified in more detail (on only systems that have them, of course):

@ Symbolic link to a non-directory

> Symbolic link to a directory

& Symbolic link to nowhere

listlinks also slows down ls-F and causes partitions holding files pointed to by symbolic links to be mounted.

If the listflags shell variable is set to `x', `a' or `A', or any combination thereof (e.g., `xA'), they are used as flags to ls-F, making it act like `ls -xF', `ls -Fa', `ls -FA' or a com? bination (e.g., `ls -FxA'). On machines where `ls -C' is not the default, ls-F acts like `ls -CF', unless listflags contains an `x', in which case it acts like `ls -xF'. ls-F passes its arguments to ls(1) if it is given any switches, so `alias ls ls-F' generally does the right thing.

The ls-F builtin can list files using different colors depend? ing on the filetype or extension. See the color shell variable and the LS_COLORS environment variable.

migrate [-site] pid|%jobid ... (+)

migrate -site (+)

The first form migrates the process or job to the site speci? fied or the default site determined by the system path. The second form is equivalent to `migrate -site $$': it migrates the current process to the specified site. Migrating the shell itself can cause unexpected behavior, because the shell does not like to lose its tty. (TCF only)

newgrp [-] [group] (+)

Equivalent to `exec newgrp'; see newgrp(1). Available only if the shell was so compiled; see the version shell variable.

nice [+number] [command]

Sets the scheduling priority for the shell to number, or, with? out number, to 4. With command, runs command at the appropri? ate priority. The greater the number, the less cpu the process gets. The super-user may specify negative priority by using `nice -number ...'. Command is always executed in a sub-shell, and the restrictions placed on commands in simple if statements apply.

nohup [command]

With command, runs command such that it will ignore hangup sig? nals. Note that commands may set their own response to

hangups, overriding nohup. Without an argument, causes the non-interactive shell only to ignore hangups for the remainder of the script. See also Signal handling and the hup builtin command.

notify [%job ...]

Causes the shell to notify the user asynchronously when the status of any of the specified jobs (or, without %job, the cur? rent job) changes, instead of waiting until the next prompt as is usual. job may be a number, a string, `', `%', `+' or `-' as described under Jobs. See also the notify shell variable.

onintr [-|label]

Controls the action of the shell on interrupts. Without argu? ments, restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the termi? nal command input level. With `-', causes all interrupts to be ignored. With label, causes the shell to execute a `goto la? bel' when an interrupt is received or a child process termi? nates because it was interrupted.

onintr is ignored if the shell is running detached and in sys? tem startup files (see FILES), where interrupts are disabled anyway.

popd [-p] [-l] [-n|-v] [+n]

Without arguments, pops the directory stack and returns to the new top directory. With a number `+n', discards the n'th entry in the stack.

Finally, all forms of popd print the final directory stack, just like dirs. The pushdsilent shell variable can be set to prevent this and the -p flag can be given to override pushdsi? lent. The -l, -n and -v flags have the same effect on popd as on dirs. (+)

printenv [name] (+)

Prints the names and values of all environment variables or, with name, the value of the environment variable name.

pushd [-p] [-l] [-n|-v] [name|+n]

Without arguments, exchanges the top two elements of the direc?
tory stack. If pushdtohome is set, pushd without arguments
does `pushd ~', like cd. (+) With name, pushes the current
working directory onto the directory stack and changes to name.
If name is `-' it is interpreted as the previous working direc?
tory (see Filename substitution). (+) If dunique is set, pushd
removes any instances of name from the stack before pushing it
onto the stack. (+) With a number `+n', rotates the nth ele?
ment of the directory stack around to be the top element and
changes to it. If dextract is set, however, `pushd +n' ex?
tracts the nth directory, pushes it onto the top of the stack
and changes to it. (+)
Finally, all forms of pushd print the final directory stack,
just like dirs. The pushdsilent shell variable can be set to
prevent this and the -p flag can be given to override pushdsi?
lent. The -l, -n and -v flags have the same effect on pushd as
on dirs. (+)

rehash Causes the internal hash table of the contents of the directo?
ries in the path variable to be recomputed. This is needed if
the autorehash shell variable is not set and new commands are
added to directories in path while you are logged in. With au?
torehash, a new command will be found automatically, except in
the special case where another command of the same name which
is located in a different directory already exists in the hash
table. Also flushes the cache of home directories built by
tilde expansion.

repeat count command

The specified command, which is subject to the same restric?
tions as the command in the one line if statement above, is ex?
ecuted count times. I/O redirections occur exactly once, even
if count is 0.

rootnode //nodename (+)

Changes the rootnode to //nodename, so that `/' will be inter?
preted as `//nodename'. (Domain/OS only)

sched (+)

sched [+]hh:mm command (+)

sched -n (+)

The first form prints the scheduled-event list. The sched shell variable may be set to define the format in which the scheduled-event list is printed. The second form adds command to the scheduled-event list. For example,

```
> sched 11:00 echo It\'s eleven o\'clock.
```

causes the shell to echo `It's eleven o'clock.' at 11 AM. The time may be in 12-hour AM/PM format

```
> sched 5pm set prompt='[%h] It\'s after 5; go home: >'
```

or may be relative to the current time:

```
> sched +2:15 /usr/lib/uucp/uucico -r1 -sother
```

A relative time specification may not use AM/PM format. The third form removes item n from the event list:

```
> sched
    1  Wed Apr  4 15:42  /usr/lib/uucp/uucico -r1 -sother
    2  Wed Apr  4 17:00  set prompt=[%h] It's after 5;  go
home: >
> sched -2
> sched
    1  Wed Apr  4 15:42  /usr/lib/uucp/uucico -r1 -sother
```

A command in the scheduled-event list is executed just before the first prompt is printed after the time when the command is scheduled. It is possible to miss the exact time when the com? mand is to be run, but an overdue command will execute at the next prompt. A command which comes due while the shell is waiting for user input is executed immediately. However, nor? mal operation of an already-running command will not be inter? rupted so that a scheduled-event list element may be run.

This mechanism is similar to, but not the same as, the at(1)

command on some Unix systems. Its major disadvantage is that it may not run a command at exactly the specified time. Its major advantage is that because sched runs directly from the shell, it has access to shell variables and other structures. This provides a mechanism for changing one's working environ? ment based on the time of day.

set

set name ...

set name=word ...

set [-r] [-f|-l] name=(wordlist) ... (+)

set name[index]=word ...

set -r (+)

set -r name ... (+)

set -r name=word ... (+)

The first form of the command prints the value of all shell variables. Variables which contain more than a single word print as a parenthesized word list. The second form sets name to the null string. The third form sets name to the single word. The fourth form sets name to the list of words in wordlist. In all cases the value is command and filename ex? panded. If -r is specified, the value is set read-only. If -f or -l are specified, set only unique words keeping their order. -f prefers the first occurrence of a word, and -l the last. The fifth form sets the index'th component of name to word; this component must already exist. The sixth form lists only the names of all shell variables that are read-only. The sev? enth form makes name read-only, whether or not it has a value. The eighth form is the same as the third form, but make name read-only at the same time.

These arguments can be repeated to set and/or make read-only multiple variables in a single set command. Note, however, that variable expansion happens for all arguments before any setting occurs. Note also that `=' can be adjacent to both

name and word or separated from both by whitespace, but cannot be adjacent to only one or the other. See also the unset builtin command.

setenv [name [value]]

Without arguments, prints the names and values of all environment variables. Given name, sets the environment variable name to value or, without value, to the null string.

setpath path (+)

Equivalent to setpath(1). (Mach only)

setspath LOCAL|site|cpu ... (+)

Sets the system execution path. (TCF only)

settc cap value (+)

Tells the shell to believe that the terminal capability cap (as defined in termcap(5)) has the value value. No sanity checking is done. Concept terminal users may have to `settc xn no' to get proper wrapping at the rightmost column.

setty [-d|-q|-x] [-a] [[+|-]mode] (+)

Controls which tty modes (see Terminal management) the shell does not allow to change. -d, -q or -x tells setty to act on the `edit', `quote' or `execute' set of tty modes respectively; without -d, -q or -x, `execute' is used.

Without other arguments, setty lists the modes in the chosen set which are fixed on (`+mode') or off (`-mode'). The available modes, and thus the display, vary from system to system. With -a, lists all tty modes in the chosen set whether or not they are fixed. With +mode, -mode or mode, fixes mode on or off or removes control from mode in the chosen set. For example, `setty +echok echoe' fixes `echok' mode on and allows commands to turn `echoe' mode on or off, both when the shell is executing commands.

setxvers [string] (+)

Set the experimental version prefix to string, or removes it if string is omitted. (TCF only)

shift [variable]

Without arguments, discards argv[1] and shifts the members of argv to the left. It is an error for argv not to be set or to have less than one word as value. With variable, performs the same function on variable.

source [-h] name [args ...]

The shell reads and executes commands from name. The commands are not placed on the history list. If any args are given, they are placed in argv. (+) source commands may be nested; if they are nested too deeply the shell may run out of file de? scriptors. An error in a source at any level terminates all nested source commands. With -h, commands are placed on the history list instead of being executed, much like `history -L'.

stop %job|pid ...

Stops the specified jobs or processes which are executing in the background. job may be a number, a string, `', `%', `+' or `-' as described under Jobs. There is no default job; saying just `stop' does not stop the current job.

suspend Causes the shell to stop in its tracks, much as if it had been sent a stop signal with ^Z. This is most often used to stop shells started by su(1).

switch (string)

case str1:

...

breaksw

...

default:

...

breaksw

endsw Each case label is successively matched, against the specified string which is first command and filename expanded. The file metacharacters `*', `?' and `[...]' may be used in the case labels, which are variable expanded. If none of the labels

match before a `default' label is found, then the execution be?
gins after the default label. Each case label and the default
label must appear at the beginning of a line. The command
breaksw causes execution to continue after the endsw. Other?
wise control may fall through case labels and default labels as
in C. If no label matches and there is no default, execution
continues after the endsw.

telltc (+)

Lists the values of all terminal capabilities (see termcap(5)).

termname [terminal type] (+)

Tests if terminal type (or the current value of TERM if no ter?
minal type is given) has an entry in the hosts termcap(5) or
terminfo(5) database. Prints the terminal type to stdout and
returns 0 if an entry is present otherwise returns 1.

time [command]

Executes command (which must be a simple command, not an alias,
a pipeline, a command list or a parenthesized command list) and
prints a time summary as described under the time variable. If
necessary, an extra shell is created to print the time statis?
tic when the command completes. Without command, prints a time
summary for the current shell and its children.

umask [value]

Sets the file creation mask to value, which is given in octal.
Common values for the mask are 002, giving all access to the
group and read and execute access to others, and 022, giving
read and execute access to the group and others. Without
value, prints the current file creation mask.

unalias pattern

Removes all aliases whose names match pattern. `unalias *'
thus removes all aliases. It is not an error for nothing to be
unaliased.

uncomplete pattern (+)

Removes all completions whose names match pattern. `uncomplete

*' thus removes all completions. It is not an error for noth?
ing to be uncompleted.

unhash Disables use of the internal hash table to speed location of executed programs.

universe universe (+)
Sets the universe to universe. (Masscomp/RTU only)

unlimit [-hf] [resource]
Removes the limitation on resource or, if no resource is speci?
fied, all resource limitations. With -h, the corresponding
hard limits are removed. Only the super-user may do this.
Note that unlimit may not exit successful, since most systems
do not allow descriptors to be unlimited. With -f errors are
ignored.

unset pattern
Removes all variables whose names match pattern, unless they
are read-only. `unset *' thus removes all variables unless
they are read-only; this is a bad idea. It is not an error for
nothing to be unset.

unsetenv pattern
Removes all environment variables whose names match pattern.
`unsetenv *' thus removes all environment variables; this is a
bad idea. It is not an error for nothing to be unsetenved.

ver [systype [command]] (+)
Without arguments, prints SYSTYPE. With systype, sets SYSTYPE
to systype. With systype and command, executes command under
systype. systype may be `bsd4.3' or `sys5.3'. (Domain/OS
only)

wait The shell waits for all background jobs. If the shell is in?
teractive, an interrupt will disrupt the wait and cause the
shell to print the names and job numbers of all outstanding
jobs.

warp universe (+)
Sets the universe to universe. (Convex/OS only)

watchlog (+)

An alternate name for the log builtin command (q.v.). Avail?able only if the shell was so compiled; see the version shell variable.

where command (+)

Reports all known instances of command, including aliases, builtins and executables in path.

which command (+)

Displays the command that will be executed by the shell after substitutions, path searching, etc. The builtin command is just like which(1), but it correctly reports tcsh aliases and builtins and is 10 to 100 times faster. See also the which-command editor command.

while (expr)

...

end Executes the commands between the while and the matching end while expr (an expression, as described under Expressions) evaluates non-zero. while and end must appear alone on their input lines. break and continue may be used to terminate or continue the loop prematurely. If the input is a terminal, the user is prompted the first time through the loop as with fore?ach.

Special aliases (+)

If set, each of these aliases executes automatically at the indicated time. They are all initially undefined.

beepcmd Runs when the shell wants to ring the terminal bell.

cwdcmd Runs after every change of working directory. For example, if the user is working on an X window system using xterm(1) and a re-parenting window manager that supports title bars such as twm(1) and does

> alias cwdcmd 'echo -n "^[]2;${HOST}:$cwd ^G"'

then the shell will change the title of the running xterm(1) to be the name of the host, a colon, and the full current working

directory. A fancier way to do that is

> alias cwdcmd 'echo -n

"^[]2;${HOST}:$cwd^G^[]1;${HOST}^G"'

This will put the hostname and working directory on the title bar but only the hostname in the icon manager menu.

Note that putting a cd, pushd or popd in cwdcmd may cause an infinite loop. It is the author's opinion that anyone doing so will get what they deserve.

jobcmd Runs before each command gets executed, or when the command changes state. This is similar to postcmd, but it does not print builtins.

> alias jobcmd 'echo -n "^[]2\;\!#:q^G"'

then executing vi foo.c will put the command string in the xterm title bar.

helpcommand

Invoked by the run-help editor command. The command name for which help is sought is passed as sole argument. For example, if one does

> alias helpcommand '\!:1 --help'

then the help display of the command itself will be invoked, using the GNU help calling convention. Currently there is no easy way to account for various calling conventions (e.g., the customary Unix `-h'), except by using a table of many commands.

periodic

Runs every tperiod minutes. This provides a convenient means for checking on common but infrequent changes such as new mail. For example, if one does

> set tperiod = 30

> alias periodic checknews

then the checknews(1) program runs every 30 minutes. If peri? odic is set but tperiod is unset or set to 0, periodic behaves like precmd.

precmd Runs just before each prompt is printed. For example, if one

does

> alias precmd date

then date(1) runs just before the shell prompts for each com?
mand. There are no limits on what precmd can be set to do, but
discretion should be used.

postcmd Runs before each command gets executed.

> alias postcmd 'echo -n "^[]2\;\!#:q^G"'

then executing vi foo.c will put the command string in the
xterm title bar.

shell Specifies the interpreter for executable scripts which do not
themselves specify an interpreter. The first word should be a
full path name to the desired interpreter (e.g., `/bin/csh' or
`/usr/local/bin/tcsh').

Special shell variables

The variables described in this section have special meaning to the
shell.

The shell sets addsuffix, argv, autologout, csubstnonl, command,
echo_style, edit, gid, group, home, loginsh, oid, path, prompt,
prompt2, prompt3, shell, shlvl, tcsh, term, tty, uid, user and version
at startup; they do not change thereafter unless changed by the user.
The shell updates cwd, dirstack, owd and status when necessary, and
sets logout on logout.
The shell synchronizes group, home, path, shlvl, term and user with the
environment variables of the same names: whenever the environment vari?
able changes the shell changes the corresponding shell variable to
match (unless the shell variable is read-only) and vice versa. Note
that although cwd and PWD have identical meanings, they are not syn?
chronized in this manner, and that the shell automatically converts be?
tween the different formats of path and PATH.

addsuffix (+)

If set, filename completion adds `/' to the end of directories
and a space to the end of normal files when they are matched
exactly. Set by default.

afsuser (+)

If set, autologout's autolock feature uses its value instead of the local username for kerberos authentication.

ampm (+)

If set, all times are shown in 12-hour AM/PM format.

anyerror (+)

This variable selects what is propagated to the value of the status variable. For more information see the description of the status variable below.

argv The arguments to the shell. Positional parameters are taken from argv, i.e., `$1' is replaced by `$argv[1]', etc. Set by default, but usually empty in interactive shells.

autocorrect (+)

If set, the spell-word editor command is invoked automatically before each completion attempt.

autoexpand (+)

If set, the expand-history editor command is invoked automati? cally before each completion attempt. If this is set to only? history, then only history will be expanded and a second com? pletion will expand filenames.

autolist (+)

If set, possibilities are listed after an ambiguous completion. If set to `ambiguous', possibilities are listed only when no new characters are added by completion.

autologout (+)

The first word is the number of minutes of inactivity before automatic logout. The optional second word is the number of minutes of inactivity before automatic locking. When the shell automatically logs out, it prints `auto-logout', sets the vari? able logout to `automatic' and exits. When the shell automati? cally locks, the user is required to enter his password to con? tinue working. Five incorrect attempts result in automatic lo? gout. Set to `60' (automatic logout after 60 minutes, and no

locking) by default in login and superuser shells, but not if the shell thinks it is running under a window system (i.e., the DISPLAY environment variable is set), the tty is a pseudo-tty (pty) or the shell was not so compiled (see the version shell variable). Unset or set to `0' to disable automatic logout. See also the afsuser and logout shell variables.

autorehash (+)

If set, the internal hash table of the contents of the directo? ries in the path variable will be recomputed if a command is not found in the hash table. In addition, the list of avail? able commands will be rebuilt for each command completion or spelling correction attempt if set to `complete' or `correct' respectively; if set to `always', this will be done for both cases.

backslash_quote (+)

If set, backslashes (`\') always quote `\', `'', and `"'. This may make complex quoting tasks easier, but it can cause syntax errors in csh(1) scripts.

catalog The file name of the message catalog. If set, tcsh use `tcsh.${catalog}' as a message catalog instead of default `tcsh'.

cdpath A list of directories in which cd should search for subdirecto? ries if they aren't found in the current directory.

cdtohome (+)

If not set, cd requires a directory name, and will not go to the home directory if it's omitted. This is set by default.

color If set, it enables color display for the builtin ls-F and it passes --color=auto to ls. Alternatively, it can be set to only ls-F or only ls to enable color to only one command. Set? ting it to nothing is equivalent to setting it to (ls-F ls).

colorcat

If set, it enables color escape sequence for NLS message files. And display colorful NLS messages.

command (+)

If set, the command which was passed to the shell with the -c flag (q.v.).

compat_expr (+)

If set, the shell will evaluate expressions right to left, like the original csh.

complete (+)

If set to `igncase', the completion becomes case insensitive. If set to `enhance', completion ignores case and considers hyphens and underscores to be equivalent; it will also treat periods, hyphens and underscores (`.', `-' and `_') as word separators. If set to `Enhance', completion matches uppercase and underscore characters explicitly and matches lowercase and hyphens in a case-insensitive manner; it will treat periods, hyphens and underscores as word separators.

continue (+)

If set to a list of commands, the shell will continue the listed commands, instead of starting a new one.

continue_args (+)

Same as continue, but the shell will execute:

echo `pwd` $argv > ~/.<cmd>_pause; %<cmd>

correct (+)

If set to `cmd', commands are automatically spelling-corrected. If set to `complete', commands are automatically completed. If set to `all', the entire command line is corrected.

csubstnonl (+)

If set, newlines and carriage returns in command substitution are replaced by spaces. Set by default.

cwd The full pathname of the current directory. See also the dirstack and owd shell variables.

dextract (+)

If set, `pushd +n' extracts the nth directory from the directory stack rather than rotating it to the top.

dirsfile (+)

The default location in which `dirs -S' and `dirs -L' look for a history file. If unset, ~/.cshdirs is used. Because only ~/.tcshrc is normally sourced before ~/.cshdirs, dirsfile should be set in ~/.tcshrc rather than ~/.login.

dirstack (+)

An array of all the directories on the directory stack. `$dirstack[1]' is the current working directory, `$dirstack[2]' the first directory on the stack, etc. Note that the current working directory is `$dirstack[1]' but `=0' in directory stack substitutions, etc. One can change the stack arbitrarily by setting dirstack, but the first element (the current working directory) is always correct. See also the cwd and owd shell variables.

dspmbyte (+)

Has an effect iff 'dspm' is listed as part of the version shell variable. If set to `euc', it enables display and editing EUC-kanji(Japanese) code. If set to `sjis', it enables display and editing Shift-JIS(Japanese) code. If set to `big5', it enables display and editing Big5(Chinese) code. If set to `utf8', it enables display and editing Utf8(Unicode) code. If set to the following format, it enables display and editing of original multi-byte code format:

> set dspmbyte = 0000....(256 bytes)....0000

The table requires just 256 bytes. Each character of 256 char? acters corresponds (from left to right) to the ASCII codes 0x00, 0x01, ... 0xff. Each character is set to number 0,1,2 and 3. Each number has the following meaning:

0 ... not used for multi-byte characters.

1 ... used for the first byte of a multi-byte character.

2 ... used for the second byte of a multi-byte character.

3 ... used for both the first byte and second byte of a multi-byte character.

Example:

If set to `001322', the first character (means 0x00 of the ASCII code) and second character (means 0x01 of ASCII code) are set to `0'. Then, it is not used for multi-byte characters. The 3rd character (0x02) is set to '1', indicating that it is used for the first byte of a multi-byte character. The 4th character(0x03) is set '3'. It is used for both the first byte and the second byte of a multi-byte character. The 5th and 6th characters (0x04,0x05) are set to '2', indicating that they are used for the second byte of a multi-byte character.

The GNU fileutils version of ls cannot display multi-byte file? names without the -N (--literal) option. If you are using this version, set the second word of dspmbyte to "ls". If not, for example, "ls-F -l" cannot display multi-byte filenames.

Note:

This variable can only be used if KANJI and DSPMBYTE has been defined at compile time.

dunique (+)

If set, pushd removes any instances of name from the stack be? fore pushing it onto the stack.

echo If set, each command with its arguments is echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, because these substitutions are then done selectively. Set by the -x command line option.

echo_style (+)

The style of the echo builtin. May be set to

bsd Don't echo a newline if the first argument is `-n'; the default for csh.

sysv Recognize backslashed escape sequences in echo strings.

both Recognize both the `-n' flag and backslashed escape se? quences; the default for tcsh.

none Recognize neither.

Set by default to the local system default. The BSD and System V options are described in the echo(1) man pages on the appro? priate systems.

edit (+)

If set, the command-line editor is used. Set by default in in? teractive shells.

editors (+)

A list of command names for the run-fg-editor editor command to match. If not set, the EDITOR (`ed' if unset) and VISUAL (`vi' if unset) environment variables will be used instead.

ellipsis (+)

If set, the `%c'/`%.' and `%C' prompt sequences (see the prompt shell variable) indicate skipped directories with an ellipsis (`...') instead of `/<skipped>'.

euid (+)

The user's effective user ID.

euser (+)

The first matching passwd entry name corresponding to the ef? fective user ID.

fignore (+)

Lists file name suffixes to be ignored by completion.

filec In tcsh, completion is always used and this variable is ignored by default. If edit is unset, then the traditional csh comple? tion is used. If set in csh, filename completion is used.

gid (+) The user's real group ID.

globdot (+)

If set, wild-card glob patterns will match files and directo? ries beginning with `.' except for `.' and `..'

globstar (+)

If set, the `**' and `***' file glob patterns will match any string of characters including `/' traversing any existing sub- directories. (e.g. `ls **.c' will list all the .c files in the current directory tree). If used by itself, it will match

zero or more sub-directories (e.g. `ls /usr/include/**/time.h' will list any file named `time.h' in the /usr/include directory tree; whereas `ls /usr/include/**time.h' will match any file in the /usr/include directory tree ending in `time.h'). To pre? vent problems with recursion, the `**' glob-pattern will not descend into a symbolic link containing a directory. To over? ride this, use `***'

group (+)

The user's group name.

highlight

If set, the incremental search match (in i-search-back and i-search-fwd) and the region between the mark and the cursor are highlighted in reverse video.

Highlighting requires more frequent terminal writes, which in? troduces extra overhead. If you care about terminal perfor? mance, you may want to leave this unset.

histchars

A string value determining the characters used in History sub? stitution (q.v.). The first character of its value is used as the history substitution character, replacing the default char? acter `!'. The second character of its value replaces the character `^' in quick substitutions.

histdup (+)

Controls handling of duplicate entries in the history list. If set to `all' only unique history events are entered in the his? tory list. If set to `prev' and the last history event is the same as the current command, then the current command is not entered in the history. If set to `erase' and the same event is found in the history list, that old event gets erased and the current one gets inserted. Note that the `prev' and `all' options renumber history events so there are no gaps.

histfile (+)

The default location in which `history -S' and `history -L'

look for a history file. If unset, ~/.history is used. hist?
file is useful when sharing the same home directory between
different machines, or when saving separate histories on dif?
ferent terminals. Because only ~/.tcshrc is normally sourced
before ~/.history, histfile should be set in ~/.tcshrc rather
than ~/.login.

histlit (+)

If set, builtin and editor commands and the savehist mechanism
use the literal (unexpanded) form of lines in the history list.
See also the toggle-literal-history editor command.

history The first word indicates the number of history events to save.
The optional second word (+) indicates the format in which his?
tory is printed; if not given, `%h\t%T\t%R\n' is used. The
format sequences are described below under prompt; note the
variable meaning of `%R'. Set to `100' by default.

home Initialized to the home directory of the invoker. The filename
expansion of `~' refers to this variable.

ignoreeof

If set to the empty string or `0' and the input device is a
terminal, the end-of-file command (usually generated by the
user by typing `^D' on an empty line) causes the shell to print
`Use "exit" to leave tcsh.' instead of exiting. This prevents
the shell from accidentally being killed. Historically this
setting exited after 26 successive EOF's to avoid infinite
loops. If set to a number n, the shell ignores n - 1 consecu?
tive end-of-files and exits on the nth. (+) If unset, `1' is
used, i.e., the shell exits on a single `^D'.

implicitcd (+)

If set, the shell treats a directory name typed as a command as
though it were a request to change to that directory. If set
to verbose, the change of directory is echoed to the standard
output. This behavior is inhibited in non-interactive shell
scripts, or for command strings with more than one word.

Changing directory takes precedence over executing a like-named command, but it is done after alias substitutions. Tilde and variable expansions work as expected.

inputmode (+)

If set to `insert' or `overwrite', puts the editor into that input mode at the beginning of each line.

killdup (+)

Controls handling of duplicate entries in the kill ring. If set to `all' only unique strings are entered in the kill ring. If set to `prev' and the last killed string is the same as the current killed string, then the current string is not entered in the ring. If set to `erase' and the same string is found in the kill ring, the old string is erased and the current one is inserted.

killring (+)

Indicates the number of killed strings to keep in memory. Set to `30' by default. If unset or set to less than `2', the shell will only keep the most recently killed string. Strings are put in the killring by the editor commands that delete (kill) strings of text, e.g. backward-delete-word, kill-line, etc, as well as the copy-region-as-kill command. The yank edi? tor command will yank the most recently killed string into the command-line, while yank-pop (see Editor commands) can be used to yank earlier killed strings.

listflags (+)

If set to `x', `a' or `A', or any combination thereof (e.g., `xA'), they are used as flags to ls-F, making it act like `ls -xF', `ls -Fa', `ls -FA' or a combination (e.g., `ls -FxA'): `a' shows all files (even if they start with a `.'), `A' shows all files but `.' and `..', and `x' sorts across instead of down. If the second word of listflags is set, it is used as the path to `ls(1)'.

listjobs (+)

If set, all jobs are listed when a job is suspended. If set to `long', the listing is in long format.

listlinks (+)

If set, the ls-F builtin command shows the type of file to which each symbolic link points.

listmax (+)

The maximum number of items which the list-choices editor com? mand will list without asking first.

listmaxrows (+)

The maximum number of rows of items which the list-choices edi? tor command will list without asking first.

loginsh (+)

Set by the shell if it is a login shell. Setting or unsetting it within a shell has no effect. See also shlvl.

logout (+)

Set by the shell to `normal' before a normal logout, `auto? matic' before an automatic logout, and `hangup' if the shell was killed by a hangup signal (see Signal handling). See also the autologout shell variable.

mail A list of files and directories to check for incoming mail, op? tionally preceded by a numeric word. Before each prompt, if 10 minutes have passed since the last check, the shell checks each file and says `You have new mail.' (or, if mail contains multi? ple files, `You have new mail in name.') if the filesize is greater than zero in size and has a modification time greater than its access time.

If you are in a login shell, then no mail file is reported un? less it has been modified after the time the shell has started up, to prevent redundant notifications. Most login programs will tell you whether or not you have mail when you log in.

If a file specified in mail is a directory, the shell will count each file within that directory as a separate message, and will report `You have n mails.' or `You have n mails in

name.' as appropriate. This functionality is provided primar? ily for those systems which store mail in this manner, such as the Andrew Mail System.

If the first word of mail is numeric it is taken as a different mail checking interval, in seconds.

Under very rare circumstances, the shell may report `You have mail.' instead of `You have new mail.'

matchbeep (+)

If set to `never', completion never beeps. If set to `no? match', it beeps only when there is no match. If set to `am? biguous', it beeps when there are multiple matches. If set to `notunique', it beeps when there is one exact and other longer matches. If unset, `ambiguous' is used.

nobeep (+)

If set, beeping is completely disabled. See also visiblebell.

noclobber

If set, restrictions are placed on output redirection to insure that files are not accidentally destroyed and that `>>' redi? rections refer to existing files, as described in the In? put/output section.

noding If set, disable the printing of `DING!' in the prompt time specifiers at the change of hour.

noglob If set, Filename substitution and Directory stack substitution (q.v.) are inhibited. This is most useful in shell scripts which do not deal with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nokanji (+)

If set and the shell supports Kanji (see the version shell variable), it is disabled so that the meta key can be used.

nonomatch

If set, a Filename substitution or Directory stack substitution (q.v.) which does not match any existing files is left un? touched rather than causing an error. It is still an error for

the substitution to be malformed, e.g., `echo [' still gives an error.

nostat (+)

A list of directories (or glob-patterns which match directo? ries; see Filename substitution) that should not be stat(2)ed during a completion operation. This is usually used to exclude directories which take too much time to stat(2), for example /afs.

notify If set, the shell announces job completions asynchronously. The default is to present job completions just before printing a prompt.

oid (+) The user's real organization ID. (Domain/OS only)

owd (+) The old working directory, equivalent to the `-' used by cd and pushd. See also the cwd and dirstack shell variables.

padhour If set, enable the printing of padding '0' for hours, in 24 and 12 hour formats. E.G.: 07:45:42 vs. 7:45:42.

parseoctal

To retain compatibily with older versions numeric variables starting with 0 are not interpreted as octal. Setting this variable enables proper octal parsing.

path A list of directories in which to look for executable commands. A null word specifies the current directory. If there is no path variable then only full path names will execute. path is set by the shell at startup from the PATH environment variable or, if PATH does not exist, to a system-dependent default some? thing like `(/usr/local/bin /usr/bsd /bin /usr/bin .)'. The shell may put `.' first or last in path or omit it entirely de? pending on how it was compiled; see the version shell variable. A shell which is given neither the -c nor the -t option hashes the contents of the directories in path after reading ~/.tcshrc and each time path is reset. If one adds a new command to a directory in path while the shell is active, one may need to do a rehash for the shell to find it.

printexitvalue (+)

If set and an interactive program exits with a non-zero status,

the shell prints `Exit status'.

prompt The string which is printed before reading each command from

the terminal. prompt may include any of the following format?

ting sequences (+), which are replaced by the given informa?

tion:

%/ The current working directory.

%~ The current working directory, but with one's home direc?

tory represented by `~' and other users' home directories

represented by `~user' as per Filename substitution.

`~user' substitution happens only if the shell has already

used `~user' in a pathname in the current session.

%c[[0]n], %.[[0]n]

The trailing component of the current working directory, or

n trailing components if a digit n is given. If n begins

with `0', the number of skipped components precede the

trailing component(s) in the format `/<skipped>trailing'.

If the ellipsis shell variable is set, skipped components

are represented by an ellipsis so the whole becomes

`...trailing'. `~' substitution is done as in `%~' above,

but the `~' component is ignored when counting trailing

components.

%C Like %c, but without `~' substitution.

%h, %!, !

The current history event number.

%M The full hostname.

%m The hostname up to the first `.'.

%S (%s)

Start (stop) standout mode.

%B (%b)

Start (stop) boldfacing mode.

%U (%u)

Start (stop) underline mode.

%t, %@

The time of day in 12-hour AM/PM format.

%T Like `%t', but in 24-hour format (but see the ampm shell variable).

%p The `precise' time of day in 12-hour AM/PM format, with seconds.

%P Like `%p', but in 24-hour format (but see the ampm shell variable).

\c c is parsed as in bindkey.

^c c is parsed as in bindkey.

%% A single `%'.

%n The user name.

%N The effective user name.

%j The number of jobs.

%d The weekday in `Day' format.

%D The day in `dd' format.

%w The month in `Mon' format.

%W The month in `mm' format.

%y The year in `yy' format.

%Y The year in `yyyy' format.

%l The shell's tty.

%L Clears from the end of the prompt to end of the display or the end of the line.

%$ Expands the shell or environment variable name immediately after the `$'.

%# `>' (or the first character of the promptchars shell vari? able) for normal users, `#' (or the second character of promptchars) for the superuser.

%{string%}

Includes string as a literal escape sequence. It should be used only to change terminal attributes and should not move the cursor location. This cannot be the last sequence in

prompt.

%? The return code of the command executed just before the prompt.

%R In prompt2, the status of the parser. In prompt3, the corrected string. In history, the history string.

`%B', `%S', `%U' and `%{string%}' are available in only eight-bit-clean shells; see the version shell variable.

The bold, standout and underline sequences are often used to distinguish a superuser shell. For example,

> set prompt = "%m [%h] %B[%@]%b [%/] you rang? "
tut [37] [2:54pm] [/usr/accts/sys] you rang? _

If `%t', `%@', `%T', `%p', or `%P' is used, and noding is not set, then print `DING!' on the change of hour (i.e, `:00' minutes) instead of the actual time.

Set by default to `%# ' in interactive shells.

prompt2 (+)

The string with which to prompt in while and foreach loops and after lines ending in `\'. The same format sequences may be used as in prompt (q.v.); note the variable meaning of `%R'.

Set by default to `%R? ' in interactive shells.

prompt3 (+)

The string with which to prompt when confirming automatic spelling correction. The same format sequences may be used as in prompt (q.v.); note the variable meaning of `%R'. Set by default to `CORRECT>%R (y|n|e|a)? ' in interactive shells.

promptchars (+)

If set (to a two-character string), the `%#' formatting sequence in the prompt shell variable is replaced with the first character for normal users and the second character for the superuser.

pushdtohome (+)

If set, pushd without arguments does `pushd ~', like cd.

pushdsilent (+)

If set, pushd and popd do not print the directory stack.

recexact (+)

If set, completion completes on an exact match even if a longer match is possible.

recognize_only_executables (+)

If set, command listing displays only files in the path that are executable. Slow.

rmstar (+)

If set, the user is prompted before `rm *' is executed.

rprompt (+)

The string to print on the right-hand side of the screen (after the command input) when the prompt is being displayed on the left. It recognizes the same formatting characters as prompt. It will automatically disappear and reappear as necessary, to ensure that command input isn't obscured, and will appear only if the prompt, command input, and itself will fit together on the first line. If edit isn't set, then rprompt will be printed after the prompt and before the command input.

savedirs (+)

If set, the shell does `dirs -S' before exiting. If the first word is set to a number, at most that many directory stack en? tries are saved.

savehist

If set, the shell does `history -S' before exiting. If the first word is set to a number, at most that many lines are saved. (The number should be less than or equal to the number history entries; if it is set to greater than the number of history settings, only history entries will be saved) If the second word is set to `merge', the history list is merged with the existing history file instead of replacing it (if there is one) and sorted by time stamp and the most recent events are retained. If the second word of savehist is `merge' and the third word is set to `lock', the history file update will be

serialized with other shell sessions that would possibly like to merge history at exactly the same time. (+)

sched (+)

The format in which the sched builtin command prints scheduled events; if not given, `%h\t%T\t%R\n' is used. The format se? quences are described above under prompt; note the variable meaning of `%R'.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of Builtin and non-builtin command execution.) Initialized to the (system-dependent) home of the shell.

shlvl (+)

The number of nested shells. Reset to 1 in login shells. See also loginsh.

status The exit status from the last command or backquote expansion, or any command in a pipeline is propagated to status. (This is also the default csh behavior.) This default does not match what POSIX mandates (to return the status of the last command only). To match the POSIX behavior, you need to unset anyerror.

If the anyerror variable is unset, the exit status of a pipe? line is determined only from the last command in the pipeline, and the exit status of a backquote expansion is not propagated to status.

If a command terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `1', all other builtin commands return status `0'.

symlinks (+)

Can be set to several different values to control symbolic link (`symlink') resolution:

If set to `chase', whenever the current directory changes to a directory containing a symbolic link, it is expanded to the real name of the directory to which the link points. This does

not work for the user's home directory; this is a bug.

If set to `ignore', the shell tries to construct a current di?
rectory relative to the current directory before the link was
crossed. This means that cding through a symbolic link and
then `cd ..'ing returns one to the original directory. This
affects only builtin commands and filename completion.

If set to `expand', the shell tries to fix symbolic links by
actually expanding arguments which look like path names. This
affects any command, not just builtins. Unfortunately, this
does not work for hard-to-recognize filenames, such as those
embedded in command options. Expansion may be prevented by
quoting. While this setting is usually the most convenient, it
is sometimes misleading and sometimes confusing when it fails
to recognize an argument which should be expanded. A compro?
mise is to use `ignore' and use the editor command normalize-
path (bound by default to ^X-n) when necessary.

Some examples are in order. First, let's set up some play di?
rectories:

```
> cd /tmp
> mkdir from from/src to
> ln -s from/src to/dst
```

Here's the behavior with symlinks unset,

```
> cd /tmp/to/dst; echo $cwd
/tmp/to/dst
> cd ..; echo $cwd
/tmp/from
```

here's the behavior with symlinks set to `chase',

```
> cd /tmp/to/dst; echo $cwd
/tmp/from/src
> cd ..; echo $cwd
/tmp/from
```

here's the behavior with symlinks set to `ignore',

```
> cd /tmp/to/dst; echo $cwd
```

/tmp/to/dst

> cd ..; echo $cwd

/tmp/to

and here's the behavior with symlinks set to `expand'.

> cd /tmp/to/dst; echo $cwd

/tmp/to/dst

> cd ..; echo $cwd

/tmp/to

> cd /tmp/to/dst; echo $cwd

/tmp/to/dst

> cd ".."; echo $cwd

/tmp/from

> /bin/echo ..

/tmp/to

> /bin/echo ".."

..

Note that `expand' expansion 1) works just like `ignore' for builtins like cd, 2) is prevented by quoting, and 3) happens before filenames are passed to non-builtin commands.

tcsh (+)

The version number of the shell in the format `R.VV.PP', where `R' is the major release number, `VV' the current version and `PP' the patchlevel.

term The terminal type. Usually set in ~/.login as described under Startup and shutdown.

time If set to a number, then the time builtin (q.v.) executes auto? matically after each command which takes more than that many CPU seconds. If there is a second word, it is used as a format string for the output of the time builtin. (u) The following sequences may be used in the format string:

%U The time the process spent in user mode in cpu seconds.

%S The time the process spent in kernel mode in cpu seconds.

%E The elapsed (wall clock) time in seconds.

%P The CPU percentage computed as (%U + %S) / %E.

%W Number of times the process was swapped.

%X The average amount in (shared) text space used in Kbytes.

%D The average amount in (unshared) data/stack space used in
    Kbytes.

%K The total space used (%X + %D) in Kbytes.

%M The maximum memory the process had in use at any time in
    Kbytes.

%F The number of major page faults (page needed to be brought
    from disk).

%R The number of minor page faults.

%I The number of input operations.

%O The number of output operations.

%r The number of socket messages received.

%s The number of socket messages sent.

%k The number of signals received.

%w The number of voluntary context switches (waits).

%c The number of involuntary context switches.

Only the first four sequences are supported on systems without
BSD resource limit functions. The default time format is `%Uu
%Ss %E %P %X+%Dk %I+%Oio %Fpf+%Ww' for systems that support re?
source usage reporting and `%Uu %Ss %E %P' for systems that do
not.

Under Sequent's DYNIX/ptx, %X, %D, %K, %r and %s are not avail?
able, but the following additional sequences are:

%Y The number of system calls performed.

%Z The number of pages which are zero-filled on demand.

%i The number of times a process's resident set size was in?
    creased by the kernel.

%d The number of times a process's resident set size was de?
    creased by the kernel.

%l The number of read system calls performed.

%m The number of write system calls performed.

%p The number of reads from raw disk devices.

%q The number of writes to raw disk devices.

and the default time format is `%Uu %Ss %E %P %I+%Oio %Fpf+%Ww'. Note that the CPU percentage can be higher than 100% on multi-processors.

tperiod (+)

The period, in minutes, between executions of the periodic special alias.

tty (+) The name of the tty, or empty if not attached to one.

uid (+) The user's real user ID.

user The user's login name.

verbose If set, causes the words of each command to be printed, after history substitution (if any). Set by the -v command line option.

version (+)

The version ID stamp. It contains the shell's version number (see tcsh), origin, release date, vendor, operating system and machine (see VENDOR, OSTYPE and MACHTYPE) and a comma-separated list of options which were set at compile time. Options which are set by default in the distribution are noted.

8b The shell is eight bit clean; default

7b The shell is not eight bit clean

wide The shell is multibyte encoding clean (like UTF-8)

nls The system's NLS is used; default for systems with NLS

lf Login shells execute /etc/csh.login before instead of after /etc/csh.cshrc and ~/.login before instead of after ~/.tcshrc and ~/.history.

dl `.' is put last in path for security; default

nd `.' is omitted from path for security

vi vi(1)-style editing is the default rather than emacs(1)-style

dtr Login shells drop DTR when exiting

bye bye is a synonym for logout and log is an alternate name

for watchlog

al autologout is enabled; default

kan Kanji is used if appropriate according to locale set?

tings, unless the nokanji shell variable is set

sm The system's malloc(3) is used

hb The `#!<program> <args>' convention is emulated when exe?

cuting shell scripts

ng The newgrp builtin is available

rh The shell attempts to set the REMOTEHOST environment

variable

afs The shell verifies your password with the kerberos server

if local authentication fails. The afsuser shell vari?

able or the AFSUSER environment variable override your

local username if set.

An administrator may enter additional strings to indicate dif?

ferences in the local version.

vimode (+)

If unset, various key bindings change behavior to be more

emacs(1)-style: word boundaries are determined by wordchars

versus other characters.

If set, various key bindings change behavior to be more

vi(1)-style: word boundaries are determined by wordchars versus

whitespace versus other characters; cursor behavior depends

upon current vi mode (command, delete, insert, replace).

This variable is unset by bindkey -e and set by bindkey -v.

vimode may be explicitly set or unset by the user after those

bindkey operations if required.

visiblebell (+)

If set, a screen flash is used rather than the audible bell.

See also nobeep.

watch (+)

A list of user/terminal pairs to watch for logins and logouts.

If either the user is `any' all terminals are watched for the

given user and vice versa. Setting watch to `(any any)' watches all users and terminals. For example,

set watch = (george ttyd1 any console $user any)

reports activity of the user `george' on ttyd1, any user on the console, and oneself (or a trespasser) on any terminal. Logins and logouts are checked every 10 minutes by default, but the first word of watch can be set to a number to check every so many minutes. For example,

set watch = (1 any any)

reports any login/logout once every minute. For the impatient, the log builtin command triggers a watch report at any time. All current logins are reported (as with the log builtin) when watch is first set.

The who shell variable controls the format of watch reports.

who (+) The format string for watch messages. The following sequences are replaced by the given information:

%n The name of the user who logged in/out.

%a The observed action, i.e., `logged on', `logged off' or `replaced olduser on'.

%l The terminal (tty) on which the user logged in/out.

%M The full hostname of the remote host, or `local' if the lo? gin/logout was from the local host.

%m The hostname of the remote host up to the first `.'. The full name is printed if it is an IP address or an X Window System display.

%M and %m are available on only systems that store the remote hostname in /etc/utmp. If unset, `%n has %a %l from %m.' is used, or `%n has %a %l.' on systems which don't store the re? mote hostname.

wordchars (+)

A list of non-alphanumeric characters to be considered part of a word by the forward-word, backward-word etc., editor com? mands. If unset, the default value is determined based on the

state of vimode: if vimode is unset, `*?_-.[]~=' is used as the default; if vimode is set, `_' is used as the default.

ENVIRONMENT

AFSUSER (+)

Equivalent to the afsuser shell variable.

COLUMNS The number of columns in the terminal. See Terminal manage?ment.

DISPLAY Used by X Window System (see X(1)). If set, the shell does not set autologout (q.v.).

EDITOR The pathname to a default editor. Used by the run-fg-editor editor command if the the editors shell variable is unset. See also the VISUAL environment variable.

GROUP (+)

Equivalent to the group shell variable.

HOME Equivalent to the home shell variable.

HOST (+)

Initialized to the name of the machine on which the shell is running, as determined by the gethostname(2) system call.

HOSTTYPE (+)

Initialized to the type of machine on which the shell is run?ning, as determined at compile time. This variable is obsolete and will be removed in a future version.

HPATH (+)

A colon-separated list of directories in which the run-help ed?itor command looks for command documentation.

LANG Gives the preferred character environment. See Native Language System support.

LC_CTYPE

If set, only ctype character handling is changed. See Native Language System support.

LINES The number of lines in the terminal. See Terminal management.

LS_COLORS

The format of this variable is reminiscent of the termcap(5)

file format; a colon-separated list of expressions of the form "xx=string", where "xx" is a two-character variable name. The variables with their associated defaults are:

no 0 Normal (non-filename) text

fi 0 Regular file

di 01;34 Directory

ln 01;36 Symbolic link

pi 33 Named pipe (FIFO)

so 01;35 Socket

do 01;35 Door

bd 01;33 Block device

cd 01;32 Character device

ex 01;32 Executable file

mi (none) Missing file (defaults to fi)

or (none) Orphaned symbolic link (defaults to ln)

lc ^[[Left code

rc m Right code

ec (none) End code (replaces lc+no+rc)

You need to include only the variables you want to change from the default.

File names can also be colorized based on filename extension. This is specified in the LS_COLORS variable using the syntax "*ext=string". For example, using ISO 6429 codes, to color all C-language source files blue you would specify "*.c=34". This would color all files ending in .c in blue (34) color.

Control characters can be written either in C-style-escaped no? tation, or in stty-like ^-notation. The C-style notation adds ^[for Escape, _ for a normal space character, and ? for Delete. In addition, the ^[escape character can be used to override the default interpretation of ^[, ^, : and =.

Each file will be written as <lc> <color-code> <rc> <filename> <ec>. If the <ec> code is undefined, the sequence <lc> <no> <rc> will be used instead. This is generally more convenient

to use, but less general. The left, right and end codes are provided so you don't have to type common parts over and over again and to support weird terminals; you will generally not need to change them at all unless your terminal does not use ISO 6429 color sequences but a different system.

If your terminal does use ISO 6429 color codes, you can compose the type codes (i.e., all except the lc, rc, and ec codes) from numerical commands separated by semicolons. The most common commands are:

0 to restore default color

1 for brighter colors

4 for underlined text

5 for flashing text

30 for black foreground

31 for red foreground

32 for green foreground

33 for yellow (or brown) foreground

34 for blue foreground

35 for purple foreground

36 for cyan foreground

37 for white (or gray) foreground

40 for black background

41 for red background

42 for green background

43 for yellow (or brown) background

44 for blue background

45 for purple background

46 for cyan background

47 for white (or gray) background

Not all commands will work on all systems or display devices.

A few terminal programs do not recognize the default end code properly. If all text gets colorized after you do a directory listing, try changing the no and fi codes from 0 to the numeri?

cal codes for your standard fore- and background colors.

MACHTYPE (+)

The machine type (microprocessor class or machine model), as determined at compile time.

NOREBIND (+)

If set, printable characters are not rebound to self-insert-command. See Native Language System support.

OSTYPE (+)

The operating system, as determined at compile time.

PATH A colon-separated list of directories in which to look for exe? cutables. Equivalent to the path shell variable, but in a dif? ferent format.

PWD (+) Equivalent to the cwd shell variable, but not synchronized to it; updated only after an actual directory change.

REMOTEHOST (+)

The host from which the user has logged in remotely, if this is the case and the shell is able to determine it. Set only if the shell was so compiled; see the version shell variable.

SHLVL (+)

Equivalent to the shlvl shell variable.

SYSTYPE (+)

The current system type. (Domain/OS only)

TERM Equivalent to the term shell variable.

TERMCAP The terminal capability string. See Terminal management.

USER Equivalent to the user shell variable.

VENDOR (+)

The vendor, as determined at compile time.

VISUAL The pathname to a default full-screen editor. Used by the run-fg-editor editor command if the the editors shell variable is unset. See also the EDITOR environment variable.

FILES

/etc/csh.cshrc Read first by every shell. ConvexOS, Stellix and Intel use /etc/cshrc and NeXTs use /etc/cshrc.std. A/UX,

AMIX, Cray and IRIX have no equivalent in csh(1), but read this file in tcsh anyway. Solaris 2.x does not have it either, but tcsh reads /etc/.cshrc. (+)

/etc/csh.login Read by login shells after /etc/csh.cshrc. ConvexOS, Stellix and Intel use /etc/login, NeXTs use /etc/login.std, Solaris 2.x uses /etc/.login and A/UX, AMIX, Cray and IRIX use /etc/cshrc.

~/.tcshrc (+) Read by every shell after /etc/csh.cshrc or its equivalent.

~/.cshrc Read by every shell, if ~/.tcshrc doesn't exist, after /etc/csh.cshrc or its equivalent. This manual uses `~/.tcshrc' to mean `~/.tcshrc or, if ~/.tcshrc is not found, ~/.cshrc'.

~/.history Read by login shells after ~/.tcshrc if savehist is set, but see also histfile.

~/.login Read by login shells after ~/.tcshrc or ~/.history. The shell may be compiled to read ~/.login before instead of after ~/.tcshrc and ~/.history; see the version shell variable.

~/.cshdirs (+) Read by login shells after ~/.login if savedirs is set, but see also dirsfile.

/etc/csh.logout Read by login shells at logout. ConvexOS, Stellix and Intel use /etc/logout and NeXTs use /etc/logout.std. A/UX, AMIX, Cray and IRIX have no equivalent in csh(1), but read this file in tcsh anyway. Solaris 2.x does not have it either, but tcsh reads /etc/.logout. (+)

~/.logout Read by login shells at logout after /etc/csh.logout or its equivalent.

/bin/sh Used to interpret shell scripts not starting with a `#'.

/tmp/sh* Temporary file for `<<'.

/etc/passwd Source of home directories for `~name' substitutions.

The order in which startup files are read may differ if the shell was

so compiled; see Startup and shutdown and the version shell variable.

NEW FEATURES (+)

This manual describes tcsh as a single entity, but experienced csh(1) users will want to pay special attention to tcsh's new features.

A command-line editor, which supports emacs(1)-style or vi(1)-style key bindings. See The command-line editor and Editor commands.

Programmable, interactive word completion and listing. See Completion and listing and the complete and uncomplete builtin commands.

Spelling correction (q.v.) of filenames, commands and variables.

Editor commands (q.v.) which perform other useful functions in the mid? dle of typed commands, including documentation lookup (run-help), quick editor restarting (run-fg-editor) and command resolution (which-com? mand).

An enhanced history mechanism. Events in the history list are time-stamped. See also the history command and its associated shell vari? ables, the previously undocumented `#' event specifier and new modi? fiers under History substitution, the *-history, history-search-*, i-search-*, vi-search-* and toggle-literal-history editor commands and the histlit shell variable.

Enhanced directory parsing and directory stack handling. See the cd, pushd, popd and dirs commands and their associated shell variables, the description of Directory stack substitution, the dirstack, owd and sym? links shell variables and the normalize-command and normalize-path edi? tor commands.

Negation in glob-patterns. See Filename substitution.

New File inquiry operators (q.v.) and a filetest builtin which uses them.

A variety of Automatic, periodic and timed events (q.v.) including scheduled events, special aliases, automatic logout and terminal lock? ing, command timing and watching for logins and logouts.

Support for the Native Language System (see Native Language System sup? port), OS variant features (see OS variant support and the echo_style shell variable) and system-dependent file locations (see FILES).

Extensive terminal-management capabilities. See Terminal management.

New builtin commands including builtins, hup, ls-F, newgrp, printenv, which and where (q.v.).

New variables that make useful information easily available to the shell. See the gid, loginsh, oid, shlvl, tcsh, tty, uid and version shell variables and the HOST, REMOTEHOST, VENDOR, OSTYPE and MACHTYPE environment variables.

A new syntax for including useful information in the prompt string (see prompt), and special prompts for loops and spelling correction (see prompt2 and prompt3).

Read-only variables. See Variable substitution.

BUGS

When a suspended command is restarted, the shell prints the directory it started in if this is different from the current directory. This can be misleading (i.e., wrong) as the job may have changed directories internally.

Shell builtin functions are not stoppable/restartable. Command se? quences of the form `a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend `b', the shell will then immedi? ately execute `c'. This is especially noticeable if this expansion re? sults from an alias. It suffices to place the sequence of commands in ()'s to force it to a subshell, i.e., `(a ; b ; c)'.

Control over tty output after processes are started is primitive; per? haps this will inspire someone to work on a good virtual terminal in? terface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell proce? dures; shell procedures should be provided rather than aliases.

Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed any? where, to be combined with `|', and to be used with `&' and `;' meta? syntax.

foreach doesn't ignore here documents when looking for its end.

It should be possible to use the `:' modifiers on the output of command substitutions.

The screen update for lines longer than the screen width is very poor if the terminal cannot move the cursor up (i.e., terminal type `dumb').

HPATH and NOREBIND don't need to be environment variables.

Glob-patterns which do not use `?', `*' or `[]' or which use `{}' or `~' are not negated correctly.

The single-command form of if does output redirection even if the expression is false and the command is not executed.

ls-F includes file identification characters when sorting filenames and does not handle control characters in filenames well. It cannot be interrupted.

Command substitution supports multiple commands and conditions, but not cycles or backward gotos.

Report bugs at https://bugs.astron.com/, preferably with fixes. If you want to help maintain and test tcsh, add yourself to the mailing list in https://mailman.astron.com/.

THE T IN TCSH

In 1964, DEC produced the PDP-6. The PDP-10 was a later re-implementation. It was re-christened the DECsystem-10 in 1970 or so when DEC brought out the second model, the KI10.

TENEX was created at Bolt, Beranek & Newman (a Cambridge, Massachusetts think tank) in 1972 as an experiment in demand-paged virtual memory operating systems. They built a new pager for the DEC PDP-10 and created the OS to go with it. It was extremely successful in academia.

In 1975, DEC brought out a new model of the PDP-10, the KL10; they intended to have only a version of TENEX, which they had licensed from BBN, for the new box. They called their version TOPS-20 (their capitalization is trademarked). A lot of TOPS-10 users (`The OPerating System for PDP-10') objected; thus DEC found themselves supporting two incompatible systems on the same hardware--but then there were 6 on the PDP-11!

TENEX, and TOPS-20 to version 3, had command completion via a user-

code-level subroutine library called ULTCMD. With version 3, DEC moved all that capability and more into the monitor (`kernel' for you Unix types), accessed by the COMND% JSYS (`Jump to SYStem' instruction, the supervisor call mechanism [are my IBM roots also showing?]).

The creator of tcsh was impressed by this feature and several others of TENEX and TOPS-20, and created a version of csh which mimicked them.

LIMITATIONS

The system limits argument lists to ARG_MAX characters.

The number of arguments to a command which involves filename expansion is limited to 1/6th the number of characters allowed in an argument list.

Command substitutions may substitute no more characters than are al? lowed in an argument list.

To detect looping, the shell restricts the number of alias substitu? tions on a single line to 20.

SEE ALSO

csh(1), emacs(1), ls(1), newgrp(1), sh(1), setpath(1), stty(1), su(1), tset(1), vi(1), x(1), access(2), execve(2), fork(2), killpg(2), pipe(2), setrlimit(2), sigvec(2), stat(2), umask(2), vfork(2), wait(2), malloc(3), setlocale(3), tty(4), a.out(5), termcap(5), environ(7), termio(7), Introduction to the C Shell

VERSION

This manual documents tcsh 6.22.03 (Astron) 2020-11-18.

AUTHORS

William Joy

Original author of csh(1)

J.E. Kulp, IIASA, Laxenburg, Austria

Job control and directory stack features

Ken Greer, HP Labs, 1981

File name completion

Mike Ellis, Fairchild, 1983

Command name recognition/completion

Paul Placeway, Ohio State CIS Dept., 1983-1993

Command line editor, prompt routines, new glob syntax and numerous fixes and speedups

Karl Kleinpaste, CCI 1983-4

Special aliases, directory stack extraction stuff, login/logout watch, scheduled events, and the idea of the new prompt format

Rayan Zachariassen, University of Toronto, 1984

ls-F and which builtins and numerous bug fixes, modifications and speedups

Chris Kingsley, Caltech

Fast storage allocator routines

Chris Grevstad, TRW, 1987

Incorporated 4.3BSD csh into tcsh

Christos S. Zoulas, Cornell U. EE Dept., 1987-94

Ports to HPUX, SVR2 and SVR3, a SysV version of getwd.c, SHORT_STRINGS support and a new version of sh.glob.c

James J Dempsey, BBN, and Paul Placeway, OSU, 1988

A/UX port

Daniel Long, NNSC, 1988

wordchars

Patrick Wolfe, Kuck and Associates, Inc., 1988

vi mode cleanup

David C Lawrence, Rensselaer Polytechnic Institute, 1989

autolist and ambiguous completion listing

Alec Wolman, DEC, 1989

Newlines in the prompt

Matt Landau, BBN, 1989

~/.tcshrc

Ray Moody, Purdue Physics, 1989

Magic space bar history expansion

Mordechai ????, Intel, 1989

printprompt() fixes and additions

Kazuhiro Honda, Dept. of Computer Science, Keio University, 1989

Automatic spelling correction and prompt3

Per Hedeland, Ellemtel, Sweden, 1990-

Various bugfixes, improvements and manual updates

Hans J. Albertsson (Sun Sweden)

ampm, settc and telltc

Michael Bloom

Interrupt handling fixes

Michael Fine, Digital Equipment Corp

Extended key support

Eric Schnoebelen, Convex, 1990

Convex support, lots of csh bug fixes, save and restore of directory stack

Ron Flax, Apple, 1990

A/UX 2.0 (re)port

Dan Oscarsson, LTH Sweden, 1990

NLS support and simulated NLS support for non NLS sites, fixes

Johan Widen, SICS Sweden, 1990

shlvl, Mach support, correct-line, 8-bit printing

Matt Day, Sanyo Icon, 1990

POSIX termio support, SysV limit fixes

Jaap Vermeulen, Sequent, 1990-91

Vi mode fixes, expand-line, window change fixes, Symmetry port

Martin Boyer, Institut de recherche d'Hydro-Quebec, 1991

autolist beeping options, modified the history search to search for the whole string from the beginning of the line to the cursor.

Scott Krotz, Motorola, 1991

Minix port

David Dawes, Sydney U. Australia, Physics Dept., 1991

SVR4 job control fixes

Jose Sousa, Interactive Systems Corp., 1991

Extended vi fixes and vi delete command

Marc Horowitz, MIT, 1991

ANSIfication fixes, new exec hashing code, imake fixes, where

Bruce Sterling Woodcock, sterling@netcom.com, 1991-1995

ETA and Pyramid port, Makefile and lint fixes, ignoreeof=n addition,

and various other portability changes and bug fixes

Jeff Fink, 1992

complete-word-fwd and complete-word-back

Harry C. Pulley, 1992

Coherent port

Andy Phillips, Mullard Space Science Lab U.K., 1992

VMS-POSIX port

Beto Appleton, IBM Corp., 1992

Walking process group fixes, csh bug fixes, POSIX file tests, POSIX

SIGHUP

Scott Bolte, Cray Computer Corp., 1992

CSOS port

Kaveh R. Ghazi, Rutgers University, 1992

Tek, m88k, Titan and Masscomp ports and fixes. Added autoconf sup?

port.

Mark Linderman, Cornell University, 1992

OS/2 port

Mika Liljeberg, liljeber@kruuna.Helsinki.FI, 1992

Linux port

Tim P. Starrin, NASA Langley Research Center Operations, 1993

Read-only variables

Dave Schweisguth, Yale University, 1993-4

New man page and tcsh.man2html

Larry Schwimmer, Stanford University, 1993

AFS and HESIOD patches

Luke Mewburn, RMIT University, 1994-6

Enhanced directory printing in prompt, added ellipsis and rprompt.

Edward Hutchins, Silicon Graphics Inc., 1996

Added implicit cd.

Martin Kraemer, 1997

Ported to Siemens Nixdorf EBCDIC machine

Amol Deshpande, Microsoft, 1997

Ported to WIN32 (Windows/95 and Windows/NT); wrote all the missing library and message catalog code to interface to Windows.

Taga Nayuta, 1998

Color ls additions.

THANKS TO

Bryan Dunlap, Clayton Elwell, Karl Kleinpaste, Bob Manson, Steve Romig, Diana Smetters, Bob Sutterfield, Mark Verber, Elizabeth Zwicky and all the other people at Ohio State for suggestions and encouragement

All the people on the net, for putting up with, reporting bugs in, and suggesting new additions to each and every version

Richard M. Alderson III, for writing the `T in tcsh' section