CEVA®
The DSP Powerhouse

CEVA-Toolbox™

Debugger Reference Guide Vol-III (TCL)

Rev 15.1.0

August 2015

DSP Division

Documentation Control

History Table:

Version	Date	Description	Remarks
10.0	01/12/2012	First tracked version	Added on 24/6/2013
10.0.5	06/02/2014	Updated	
10.1	09/03/2014	Updated	
15.1.0	11/08/2015	Minor updates	

Disclaimer and Proprietary Information Notice

The information contained in this document is subject to change without notice and does not represent a commitment on any part of CEVA®, Inc. CEVA®, Inc. and its subsidiaries make no warranty of any kind with regard to this material, including, but not limited to implied warranties of merchantability and fitness for a particular purpose whether arising out of law, custom, conduct or otherwise.

While the information contained herein is assumed to be accurate, CEVA®, Inc. assumes no responsibility for any errors or omissions contained herein, and assumes no liability for special, direct, indirect or consequential damage, losses, costs, charges, claims, demands, fees or expenses, of any nature or kind, which are incurred in connection with the furnishing, performance or use of this material.

This document contains proprietary information, which is protected by U.S. and international copyright laws. All rights reserved. No part of this document may be reproduced, photocopied, or translated into another language without the prior written consent of CEVA, Inc.

CEVA®, CEVA-XC™, CEVA-XC321™, CEVA-XC323™, CEVA-Xtend™, CEVA-XC4000™, CEVA-XC4100™, CEVA-XC4200™, CEVA-XC4210™, CEVA-XC4400™, CEVA-XC4410™, CEVA-XC4500™, CEVA-TeakLite™, CEVA-TeakLite-II™, CEVA-TeakLite-III™, CEVA-TL3210™, CEVA-TL3211™, CEVA-TeakLite-4™, CEVA-TL410™, CEVA-TL411™, CEVA-TL420™, CEVA-TL421™, CEVA-Quark™, CEVA-Teak™, CEVA-X™, CEVA-X1620™, CEVA-X1622™, CEVA-X1641™, CEVA-X1643™, Xpert-TeakLite-II™, Xpert-Teak™, CEVA-XS1100A™, CEVA-XS1200™, CEVA-XS1200A™, CEVA-TLS100™, Mobile-Media™, CEVA-MM1000™, CEVA-MM2000™, CEVA-SP™, CEVA-VP™, CEVA-MM3000™, CEVA-MM3100™, CEVA-MM3101™, CEVA-XM™, CEVA-XM4™, CEVA-X2™ CEVA-Audio™, CEVA-HD-Audio™, CEVA-VoP™, CEVA-Bluetooth™, CEVA-SATA™, CEVA-SAS™, CEVA-Toolbox™, SmartNcode™ are trademarks of CEVA, Inc.

All other product names are trademarks or registered trademarks of their respective owners

Support

CEVA® makes great efforts to provide a user-friendly software and hardware development environment. Along with this, CEVA® provides comprehensive documentation, enabling users to learn and develop applications on their own. Due to the complexities involved in the development of DSP applications that may be beyond the scope of the documentation, an on-line Technical Support Service (support@ceva-dsp.com) has been established. This service includes useful tips and provides fast and efficient help, assisting users to quickly resolve development problems.

How to Get Technical Support:

FAQs: Visit our web site http://www.ceva-dsp.com or your company's protected page on the CEVA® website for the latest answers to frequently asked questions.

Application Notes: Visit our website http://www.ceva-dsp.com or your company's protected page on the CEVA website for the latest application notes.

Email: Use CEVA's central support email address support@ceva-dsp.com. Your email will be forwarded automatically to the relevant support engineers and tools developers who will provide you with the most professional support in order to help you resolve any problem.

License Keys: Please refer any License Key requests or problems to sdtkeys@ceva-dsp.com. Refer to ***SDT Installation & Licensing Scheme Guide*** for SDT license keys installation information

Email: support@ceva-dsp.com
Visit us at: www.ceva-dsp.com

List of Sales and Support Centers

Israel	USA	Ireland	Sweden
2 Maskit Street P.O.Box 2068 Herzelia 46120 Israel **Tel:** +972 9 961 3700 **Fax:** +972 9 961 3800	1943 Landings Drive Mountain View, CA 94043 USA **Tel:** +1-650-417-7923 **Fax:** +1-650-417-7924	Segrave House 19/20 Earlsfort Terrace 3rd Floor Dublin 2 Ireland **Tel**: +353 1 237 3900 **Fax**: +353 1 237 3923	Klarabergsviadukten 70 Box 70396 107 24 Stockholm, Sweden **Tel**: +46(0)8 506 362 24 **Fax**: +46(0)8 506 362 20
China (Shanghai)	**China (Beijing)**	**China Shenzhen**	**Hong Kong**
Room 517, No. 1440 Yan An Road (C) Shanghai 200040 China **Tel:** +86-21-22236789 **Fax:** +86 21 22236800	Rm 503, Tower C, Raycom InfoTech Park No.2, Kexueyuan South Road, Haidian District Beijing 100190, China **Tel:** +86-10 5982 2285 **Fax:** +86-10 5982 2284	2702-09 Block C Tiley Central Plaza II Wenxin 4th Road, Nanshan District Shenzhen 518054 **Tel:** +86-755-86595012	Level 43, AIA Tower, 183 Electric Road, North Point Hong Kong **Tel:** +852-39751264 **:**
South Korea	**Taiwan**	**Japan**	**France**
#478, Hyundai Arion, 147, Gumgok-Dong, Bundang-Gu, Sungnam-Si, Kyunggi-Do, 463-853, Korea **Tel**: +82-31-704-4471 **Fax**:+82-31-704-4479	Room 621 No.1, Industry E, 2nd Rd Hsinchu, Science Park Hsinchu 300 Taiwan R.O.C **Tel**: +886 3 5798750 **Fax**: +886 3 5798750	3014 Shinoharacho Kasho Bldg. 4/F Kohoku-ku Yokohama, Kanagawa 222-0026 Japan **Tel:** +81 045-430-3901 **Fax:** +81 045-430-3904	RivieraWaves S.A.S 400, avenue Roumanille Les Bureaux Green Side 5, Bât 6 06410 Biot - Sophia Antipolis, France **Tel:** +33 4 83 76 06 00 **Fax:** +33 4 83 76 06 01

Table of Contents

List of Tables

19. TCL Command Language

The Debugger Command Language Interpreter (DCLI) is extended with the Tool Command Language (TCL) . The user can decide to work either in the regular Debugger CLI Command Mode or in an enhanced TCL Command Mode. When in TCL Mode, the programmer can still use the regular CLI commands. Using the TCL language enables the creation of complex and structural script files for testing.

Following is a description of the TCL language and its use.
TCL is widely used interpreted script language developed by Dr. John Ousterhout at the University of California, Berkeley, and now developed and maintained by Scriptics.
Various reference and training material onTCL are available online.
Current documents describes the TCL varian integrated in CEVA DCLI tool.

19.1 Description of the TCL language

TCL stands for Tool Command Language. It is really two things: a scripting language, and an interpreter for that language, designed to be easy to embed into the user application. TCL and its associated X Windows Toolkit (Tk), were designed and crafted by Professor John Ousterhout of the University of California, Berkeley. The user can find these packages on the Internet, (as explained later), and use them freely in his application, even if it is commercial. The TCL Interpreter has been ported from UNIX to PC environments.

As a scripting language, TCL is similar to other UNIX shell languages such as the Bourne Shell (sh), the C Shell (csh), the Korn Shell (ksh), and Perl.

Shell programs let the user execute other programs. They provide enough programmability (variables, control flow, and procedures) to let him build complex scripts that assemble existing programs into a new tool tailored for his needs. Shells are wonderful for automating routine chores.

It is the ability to easily add a TCL Interpreter to the application that sets it apart from other shells. TCL fills the role of an extension language that is used to configure and customize applications. There is no need to invent a command language for a new user application, or struggle to provide some sort of user-programmability for a user tool. Instead, by adding a TCL Interpreter, the user can structure his application as a set of primitive operations that can be composed by a script to best suit the needs of his users. It also allows other programs to have programmable control over the user application, leading to suites of applications that work well together.

19.2 How to Enter and Exit the TCL Mode

To enter TCL Mode, the *start tcl* CLI command should be issued to the Debugger. This command initiates TCL Mode for the Debugger, in which each command entered is interpreted as a TCL command.

When in TCL Mode, the user can still use the CLI commands, in addition to the TCL commands. However, as some CLI commands have the same syntax as other TCL commands, the Debugger differentiates between CLI and TCL commands by the prefix **deb** for any Debugger CLI command.

Let us demonstrate this point. Some commands which have a different meaning in TCL and CLI are: set, continue, break and exit.

For example:
The command “set break C:0005” is used in CLI to set a breaking point at the code memory address C:0005. However, it can also be referred as a TCL command that sets the value of the variable "break" to the string "c:0005".

The solution is that, when in TCL Mode, the user should supply the deb prefix for any Debugger CLI command.

```
start tcl
set j 5 ; here, the command sets variable j to be “5”
foreach i { 1 2 3 } ; execute the following 3 times
{
        puts "i=$i"
        set j [expr $i+7]
        puts "j=$j"
        deb set break c:000$i ; ; here, we set a breakpoint (prefix by "deb")
}
deb stop tcl ; stop tcl is a Debugger command, thus prefixed by "deb"
```

In this example, the Debugger's CLI command "set break", will be executed three times, each time substituting a different value for $i :

set break c:0001

set break c:0002

set break c:0003

Notes:

- Before sending the **deb** commands to the Debugger, TCL performs all variables substitutions if needed.
- To exit TCL Mode, the *deb stop tcl* CLI command should be used.

19.3 Interchanging Between Debugger's Variables and TCL Variables

As a result of adding the TCL Utility, we have two different spaces of variables: the Debugger's variables (inside the Debugger's data/code memory and registers) and the TCL variables as defined in the script file.

Therefore, there should be a way to interchange between these two variable sets. Let's look at the following example:

We are in TCL Mode. We would like to check the value of a data memory variable D:xxxx in an "if" statement, or define a loop counter according to this value. This memory variable is defined in run-time. However, we cannot simply put D:xxxx inside a TCL command, as TCL will interpret D:xxxx as a simple string.

The interchange between the Debugger's variables and the TCL variables is achieved with the **deb** command prefix. The **deb** command prefix can return a string value to the TCL context, in the same way other TCL commands return string values. Any of the Debugger's query commands, which provide a value (such as: ? and *) may be used in the context of TCL scripts and substitution mechanism. The ? and * query commands are used to evaluate the numeric expression of a register or a memory address respectively. For example, **?r0** yields the contents of register r0, and ***MySeg.MyOffset** yields the value address MySeg.MyOffset in the memory address. This approach gives us all the flexibility of TCL combined with all the power of the existing CLI command set.

For the following examples, assume that we are in TCL Mode :

(1) Setting TCL variables by registers and data memory values.

The TCL command "**set j [deb *d:5]**" will set j to the suitable data memory contents (it will actually read from the hardware in Emulation Mode). First TCL gets this command, then calls the Debugger's command "*d: 5". The Debugger puts the "*d: 5" command to its CLI queue, and then

parses this command and executes it. Execution results in writing the appropriate value to the Command Window as usual. This same string (which was written to the Command Window) is returned by the Debugger to the TCL Interpreter. TCL substitutes "[deb *d: 5]" with the returned value and sets j to this value.

The TCL command "**set i [deb? r0]**" will set i to r0's contents, in the same way.

In a more complicated example, a data memory value is multiplied by a TCL variable "i" and the product is assigned to another TCL variable, k. Let's assume that the contents of MySeg.MyOffset is: 3.

```
set i 3 ; i = 3
set k [expr $i*[deb *MySeg.MyOffset]]
```

After executing these TCL commands, the value of k is 9.

(2) Setting Debugger's variables by TCL variable values.

If we want to use a TCL variable for setting a Debuggers variable, it is a straightforward procedure. For example, this TCL sequence:

```
set i 5
deb d:0=$i
```

Results in the "d: 0=5" CLI command.

19.4 Summary of TCL Language Syntax

The following rules define the syntax and semantics of TCL Language:

[1] A TCL script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

[2] A command is evaluated in two steps. First, the TCL Interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command; then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or TCL script. Different commands interpret their words differently.

[3] Words of a command are separated by white space (except for new lines, which are command separators).

[4] If the first character of a word is a double-quote (``"'') then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

[5] If the first character of a word is an open brace (``{'') then the word is terminated by the matching close brace (``}''). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

[6] If a word contains an open bracket (``['') then TCL performs command substitution. To do this it invokes the TCL interpreter recursively to process the characters following the open bracket as a TCL script. The script may contain any number of commands and must be terminated by a close bracket (``]''). The result of the script (i.e. the result of its last command) is substituted in the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

[7] If a word contains a dollar-sign (``$'') then TCL performs a variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitutes may take any of the following forms:

$name

Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

$name(index)

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

${name}

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces. There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

[8] If a backslash (``\'') appears within a word, then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing.
Table 19-1 below lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

Table 19-1: TCL Backslash Sequences

Backslash Sequence	Operation
\a	Audible alert (bell) (0x7)
\b	Backspace (0x8).
\f	Form feed (0xc)
\n	New Line (0xa)
\r	Carriage-return (0xd)
\t	Tab (0x9)
\v	Vertical tab (0xb)
\<newline>whiteSpace	

A single space character replaces the backslash, newline, and all white space after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

\\

Backslash (``\'').

\ooo

The digits ooo (one, two, or three of them) give the octal value of the character.

\xhh

The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present. Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

[9] If a hash character (``#'') appears at a point where TCL is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

[10] Each character is processed exactly once by the TCL Interpreter as part of creating the words of a command. For example, if variable substitution occurs, then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the TCL interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

[11] Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

[12] Environment variable can be evaluated and set by using the **env** TCL command, e.g.
set c $env(OAKTOOLS)
if {$c=="E:\TOOLS\OAK"} { ...DO_SOMETHING… }
set env(PORTNUM) 340
deb start emu $env(PORTNUM) ==> 'deb start emu 340'

19.5 Some Practical Examples For Using TCL

Example 1: Set register to the code label

```
start tcl
scan [deb ?S_Main.Reset] "C:%x" m
```

Following this command, the TCL variable m contains a numerical value without "C:", so the user can set any Debugger register (or data location) to this value; for example:

```
deb pc=$m
deb r0=$m
deb d:20=$m
```

The user can also use any arithmetic operation on this value; for example:

```
deb r1=[expr $m+5]
deb r4=[expr $m*2]
```

The user can also use this technique to extract values from any Debugger output in the form C:xxxx or D:xxxx; for example:

```
scan [deb ?S_Main.Begin] "C:%x" b
scan [deb ?S_Main.Stop] "C:%x" s
deb pc=$b
while { [deb ?pc] < $s } {
deb step }
```

Example 2: How to generate hexadecimal values, expected by some of the Debugger's CLI commands?

When one of the parameters of a CLI command is a data or code address or address range, it is expected to have a hexadecimal value. The following example fills the data memory range from D:0000 to D:001F with the values 0 to 0x1F.

```
for { set i 0 } { $i < 0x20 } { incr i 1 } {
```

```
deb d:[format "%x" $i]=$i }
```

Example 3: How to use run time variables in the Debugger's commands? Pay attention to the format of the Debugger's **copy** command. The user can supply an additional parameter: the offset inside the file (beginning from 0).

Please note, the user should write **\[** instead of **[,** and **\]** instead of **]** while inside TCL, because these symbols (**[** and **]**) are reserved for TCL. The results of this loop will be written to files f0.dat, f1.dat ,..., f179.dat.

```
scan [deb ?TSPEECH.BeforeEncoding] "C:%x" m
deb go TSPEECH.BeforeEncoding
for { set i 0 } { $i < 180 } { incr i 1 } {
deb pc = $m
deb copy Test_src.asc \[d:0,80\] [expr $i*80]
deb go TSPEECH.AfterEncoding
deb copy \[d:86AF,11\] f$i.dat }
deb stop tcl
```

19.6 The TCL Commands

- append
- array
- break
- case
- catch
- cd
- close
- concat
- continue
- eof
- error
- eval
- exec
- exit
- expr
- file
- flush
- for
- foreach
- format
- gets
- glob
- global
- history
- if
- incr
- info
- join
- lappend
- library
- lindex
- linsert
- list
- llength
- lrange
- lreplace
- lsearch
- lsearch
- lsort

- open
- pid
- proc
- puts
- pwd
- read
- regexp
- regsub
- rename
- return
- scan
- seek
- set
- source
- split
- string
- switch
- TCL TildeSubst
- tclsh
- tclvars
- tell
- time
- trace
- unknown
- unset
- uplevel
- upvar
- while

19.6.1 append

append - Append to variable

(TCL Commands)

SYNOPSIS: append varName value ?value value ...?

DESCRIPTION: Append all of the value arguments to the current value of variable varName.
If varName doesn't exist, it is given a value equal to the concatenation of all the value arguments.
This command provides an efficient way to build up long variables incrementally.
For example, append a $b" is much more efficient than ``set a ab" if $a is long.

19.6.2 array

(TCL Commands)

Manipulate array variables

SYNOPSIS: array option arrayName ?arg arg ...?

DESCRIPTION: This command performs one of several operations on the variable given by arrayName. ArrayName must be the name of an existing array variable. The option argument determines what action is carried out by the command. The legal options (which may be abbreviated) are:

array anymore arrayName searchId

Returns 1 if there are any more elements left to be processed in an array search, 0 if all elements have already been returned. SearchId indicates which search on arrayName to check, and must be the return value from a previous invocation of array start search. This option is particularly useful if an array has an element with an empty name, since the return value from array nextelement won't indicate whether the search has been completed.

array donesearch arrayName searchId

This command terminates an array search and destroys all the states associated with this search. SearchId indicates which search on arrayName to destroy and must be the return value from a previous invocation of array startsearch. Returns an empty string.

array names arrayName

Returns a list containing the names of all of the elements in the array. If there are no elements in the array, then an empty string is returned.

array nextelement arrayName searchId

Returns the name of the next element in arrayName, or an empty string if all elements of

arrayName have already been returned in this search. The searchId argument identifies the search, and must be the return value of an array startsearch command.

Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if array donesearch had been invoked; this will cause array nextelement operations to fail for those searches.

array size arrayName

Returns a decimal string giving the number of elements in the array.

array startsearch arrayName

This command initializes an element-by-element search through the array given by arrayName, such that invocations of the array nextelement command will return the names of the individual elements in the array. When the search has been completed, the array donesearch command should be invoked. The return value is a search identifier that must be used in array nextelement and array donesearch commands; it allows multiple searches simultaneously for the same array.

19.6.3 break

break - Abort looping command

(TCL Commands)

SYNOPSIS: break

DESCRIPTION: This command may be invoked only inside the body of a looping command such as **for**, **for each** or **while**. It returns a TCL_BREAK code to signal the innermost containing loop command to return immediately.

19.6.4 case

case- Evaluate one of several scripts, depending on a given value

(TCL Commands)

SYNOPSIS:

case string ?in? patList body ?patList body ...?
case string ?in? {patList body ?patList body ...?}

DESCRIPTION: Note: the case command is obsolete and is supported only for backward compatibility. At some point in the future it may be removed entirely. The user should use the switch command instead. The case command matches a string against each of the patList arguments in order. Each patList argument is a list of one or more patterns. If any of these patterns matches the string, then case evaluates the following body argument by passing it recursively to the TCL Interpreter and returns the result of that evaluation. Each patList argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild cards described under string match. If a patList argument is the default, the corresponding body will be evaluated if no patList matches string. If no patList argument matches string and no default is given, then the case command returns an empty string. Two syntaxes are provided for the patList and body arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument. The argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line case commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the patList arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different from the first form in some cases.

19.6.5 catch

catch - Evaluate script and trap exceptional returns

(TCL Commands)

SYNOPSIS: catch script ?varName?

DESCRIPTION: The catch command may be used to prevent errors from aborting command interpretation. Catch calls the TCL Interpreter recursively to execute script, and always returns a TCL_OK code, regardless of any errors that might occur while executing script. The return value from catch is a decimal string giving the code returned by the TCL Interpreter after executing script. This will be 0 (TCL_OK) if there were no errors in script; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see tcl.h for the definitions of code values). If the varName argument is given, then it gives the name of a variable; catch will set the variable to the string returned from script (either a result or an error message).

19.6.6 cd

cd - Change working directory

(TCL Commands)

SYNOPSIS: cd ?dirName?

DESCRIPTION: Change the current working directory to dirName, or to the home directory (as specified in the HOME environment variable) if dirName is not given. If dirName starts with a tilde, then tilde-expansion is done as described for Tcl_TildeSubst. Returns an empty string.

19.6.7 close

close – Close an open file

(TCL Commands)

SYNOPSIS: close fileId

DESCRIPTION: Closes the file given by fileId. FileId must be the return value from a previous invocation of the open command; after this command, it should not be used anymore. If fileId refers to a command pipeline instead of a file, it then waits for the children to complete. The normal result of this command is an empty string, but errors are returned if there are problems in closing the file or waiting for children to complete.

19.6.8 concat

concat - Join lists together

(TCL Commands)

SYNOPSIS: concat ?arg arg ...?

DESCRIPTION: This command treats each argument as a list and concatenates the arguments into a single list. It also eliminates leading and trailing spaces in the args and adds a single separator space between args. It permits any number of arguments. For example, the command concatenates a b {c d e} {f {g h}}, will return a b c d e f {g h} as its result. If no args are supplied, the result is an empty string.

19.6.9 continue

continue - Skip to the next iteration of a loop

(TCL Commands)

SYNOPSIS: continue

DESCRIPTION: This command may be invoked only inside the body of a looping command such as for or foreach or while. It returns a TCL_CONTINUE code to signal the innermost containing loop command to skip the remainder of the loop's body but continue with the next iteration of the loop.

19.6.10 eof

eof - Check for end-of-file condition on open file

(TCL Commands)

SYNOPSIS: eof fileId

DESCRIPTION: Returns 1 if an end-of-file condition has occurred on fileId, 0 otherwise. FileId must be the return value from a previous call to open, or it may be stdin, stdout, or stderr to refer to one of the standard I/O channels.

19.6.11 error

error - Generate an error

(TCL Commands)

SYNOPSIS: error message ?info? ?code?

DESCRIPTION: Returns a TCL_ERROR code, which causes command interpretation to be unwound. Message is a string that is returned to the application to indicate what went wrong. If the info argument is provided and is non-empty, it is used to initialize the global variable errorInfo. errorInfo is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the TCL interpreter adds information to errorInfo. If the info argument is present, it is used to initialize errorInfo and the first increment of unwind information will not be added by the TCL interpreter. In other words, the command containing the error command will not appear in errorInfo; in its place will be info.

This feature is most useful in conjunction with the catch command: if a caught error cannot be handled successfully, info can be used to return a stack trace that reflects the original point of occurrence of the error:

catch {...} errMsg

set savedInfo $errorInfo...
error $errMsg

$savedInfo

If the code argument is present, then its value is stored in the errorCode global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see the section BUILT-IN VARIABLES below for information on the proper format for the variable. If the code argument is not present, then errorCode is automatically reset to ``NONE'' by the TCL Interpreter as part of processing the error generated by the command.

19.6.12 eval

eval - Evaluate a TCL script

(TCL Commands)

SYNOPSIS: eval arg ?arg ...?

DESCRIPTION: Eval takes one or more arguments, which together comprise a TCL script containing one or more commands. Eval concatenates all its arguments in the same fashion as the **concat** command, passes the concatenated string to the TCL Interpreter recursively, and returns the result of that evaluation (or any error generated by it).

19.6.13 exec

exec - Invoke subprocess(es)

(TCL Commands)

SYNOPSIS: exec ?switches? arg ?arg ...?

DESCRIPTION: This command treats its arguments as the specification of one or more subprocesses to execute. The arguments take the form of a standard shell pipeline where each arg becomes one word of a command, and each distinct command becomes a subprocess. If the initial arguments to exec start with - then they are treated as command-line switches and are not part of the pipeline specification. The following switches are currently supported:

-keepnewline Retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.

Marks the end of switches. The argument following this one will be treated as the first arg even if it starts with a -.

If an arg (or pair of args) has one of the forms described below, then it is used by exec to control the flow of input and output among the subprocess(es). Such arguments will not be passed to the subprocess(es). In forms such as ``< fileName" fileName may either be in a separate argument from ``<" or in the same argument with no intervening space (i.e.``<fileName"). | Separates distinct commands in the pipeline. The standard output of the preceding command will be piped into the standard input of the next command.

|& Separates distinct commands in the pipeline. Both standard output and standard error of the preceding command will be piped into the standard input of the next command. This form of redirection overrides forms such as 2> and >&.

< fileName: The file named by fileName is opened and used as the standard input for the first command in the pipeline.

<@ fileId: FileId must be the identifier for an open file, such as the return value from a previous call to open. It is used as the standard input for the first command in the pipeline. FileId must have been opened for reading.

<< value: Value is passed to the first command as its standard input.

> fileName: Standard output from the last command is redirected to the file named fileName, overwriting its previous contents. FileNameStandard error from all commands in the pipeline is redirected to the file namedfileName, overwriting its previous contents.

>& fileName: Both standard output from the last command and standard errors from all commands are redirected to the file named fileName, overwriting its previous contents.

>> fileName
Standard output from the last command is redirected to the file named fileName, appending to it rather than overwriting it.

> fileName
Standard error from all commands in the pipeline is redirected to the file named fileName, appending to it rather than overwriting it.

>>& fileName
Both standard output from the last command and standard errors from all commands are redirected to the file named fileName, appending to it rather than overwriting it.

>@ fileId

FileId must be the identifier for an open file, such as the return value from a previous call to open. Standard output from the last command is redirected to fileId's file, which must have been opened for writing.

@ fileId
FileId must be the identifier for an open file, such as the return value from a previous call to open. Standard errors from all commands in the pipeline are redirected to fileId's file. The file must have been opened for writing.

>&@ fileId

FileId must be the identifier for an open file, such as the return value from a previous call to open.

Both standard output from the last command and standard errors from all commands are redirected to fileId's file. The file must have been opened for writing. If standard output has not been redirected, then the exec command returns the standard output from the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then exec will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the errorCode variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file and that standard error isn't redirected, then exec will return an error; the error message will include the pipeline's standard output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is normally deleted from the result or error message. This is consistent with other TCL return values, which do not normally end with newlines. However, if -keepnewline is specified then the trailing newline is retained.

If standard input isn't redirected with ``<" or ``<<" or ``<@" then the standard nput for the first command in the pipeline is taken from the application's current standard input.

If the last arg is ``&" then the pipeline will be executed in background. In this case the exec command will return a list whose elements are the process identifiers for all of the subprocesses in the pipeline. The standard output from the last command in the pipeline will go to the application's standard output if it hasn't been redirected, and error output from all of the commands in the pipeline will go to the application's standard error file unless redirected.

The first word in each command is taken as the command name; tilde-substitution is performed on it, and if the result contains no slashes then the directories in the PATH environment variable are searched for an executable by the given name. If the name contains a slash then it must refer to an executable reachable from the current directory. No ``glob" expansion or other shell-like substitutions are performed on the arguments to commands.

19.6.14 exit

exit- End the application

(TCL Commands)

SYNOPSIS: exit ?returnCode?

DESCRIPTION: Terminate the process, returning returnCode to the system as the exit status. If returnCode isn't specified then it defaults to 0.

19.6.15 expr

expr - Evaluate an expression

(TCL Commands)

SYNOPSIS: expr arg ?arg arg ...?

DESCRIPTION: Concatenates args (adding separator spaces between them), evaluates the result as a TCL expression, and returns the value. The operators permitted in TCL expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression expr 8.2 + 6 evaluates to 14.2. TCL expressions differ from C expressions in the way that operands are specified. Also, TCL expressions support non-numeric operands and string comparisons.

OPERANDS

A TCL expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is 0), or in hexadecimal (if the first two characters of the operand are 0x). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the ``f", ``F", ``l", and ``L" suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

[1] As a numeric value, either integer or floating-point.

[2] As a TCL variable, using standard $ notation. The variable's value will be used as the operand.

[3] As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand.

[4] As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.

[5] As a TCL command enclosed in brackets. The command will be executed and its result will be used as the operand.

[6] As a mathematical function whose arguments have any of the above forms for operands, such as ``sin($x)". See below for a list of defined functions.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents. For some examples of simple expressions, suppose the variable a has the value 3 and the variable b has the value 6. Then the command on the left side of each of the lines below will produce the value on the right side of the line:

```
expr 3.1 + $a 6.1
expr 2 + "$a.$b" 5.6
expr 4*[llength "6 2"] 8
expr {{word one} < "word $a"} 0
```

OPERATORS

The valid operators are listed below, grouped in decreasing order of precedence:

- ~ !

Unary minus, bit-wise NOT, logical NOT. None of these operands may be appliedto string operands, and bit-wise NOT may be applied only to integers.

* / %

Multiply, divide, remainder. None of these operands may be applied to stringoperands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.

+ -

Add and subtract. Valid for any numeric operands.

<< >>

Left and right shift. Valid for integer operands only.

< > <= >=

Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

== !=

Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

&

Bit-wise AND. Valid for integer operands only.

^

Bit-wise exclusive OR. Valid for integer operands only.

|

Bit-wise OR. Valid for integer operands only.

&&

Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).

||

Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).

x?y:z

If-then-else, as in C. If x evaluates to non-zero, then the result is the value of y.

Otherwise the result is the value of z. The x operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level. For example, the command expr 4*2 < 7 returns 0.

The &&, ||, and ?: operators have ``lazy evaluation'', just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example: in the command expr {$v ? [a] : [b]} only one of [a] or [b] will actually be evaluated, depending on the value of $v. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the TCL parser will evaluate both [a] and [b] before invoking the expr command.

MATH FUNCTIONS

TCL supports the following mathematical functions in expressions:

acos cos hypot sinh

asin cosh log sqrt

atan exp log10 tan

atan2 floor pow tanh

ceil fmod sin

Each of these functions invokes the math library function of the same name; see the manual entries for the library functions for details on what they do. TCL also implements the following functions for conversions between integers and floating-point numbers: abs (arg)

Returns the absolute value of arg. Arg may be either integer or floating-point, and the result is returned in the same form: double (arg). If arg is a floating value, it returns arg; otherwise it converts arg to floating-point and returns the converted value; int (arg). If arg is an integer value, it returns arg, otherwise it converts arg to integer by truncation and returns the converted value; round (arg). If arg is an integer value, it returns arg, otherwise it converts arg to an integer by

rounding and returns the converted value. In addition to these predefined functions, applications may define additional functions using Tcl_CreateMathFunc().

TYPES, OVERFLOW, AND PRECISION

All internal computations involving integers are done with the C type long and all internal computations involving floating-point are done with the C type double. When converting a string to floating-point, exponent overflow is detected and results in a TCL error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable. Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetical computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example, expr 5 / 4returns 1, while expr 5 / 4.0, expr 5 / ([string length "abcd"] + 0.0) both return 1.25. Floating-point values are always returned with a ``." or an ``e" so that they will not look like integer values. For example, expr 20.0/5.0 returns ``4.0", not ``4". The global variable tcl_precision determines the number of significant digits that are retained when floating values are converted to strings (except that trailing zeroes are omitted). If tcl_precision is unset, then 6 digits of precision are used. To retain all of the significant bits of an IEEE floating-point number set tcl_precision to 17; if a value is converted to string with 17 digits of precision and then converted back to binary for some later calculation, the resulting binary value is guaranteed to be identical to the original one.

STRING OPERATIONS

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C sprintf format character sign %d for integers and %g for floating-point values. For example, the commands:

expr {"0x03" > "2"}

expr {"0y" < "0x12"}

Both return 1. The first comparison is done using integer comparison, and the second is done by using string comparison; after the second operand is converted to the string ``18".

19.6.16 file

file - Manipulate file names and attributes:

(TCL Commands)

SYNOPSIS: file option name ?arg arg ...?

DESCRIPTION: This command provides several options for operations on a file's name or attributes. Name is the name of a file; if it starts with a tilde, then tilde substitution is performed before executing the command (see the manual entry for Tcl_TildeSubst for details). Option indicates what to do with the file name. Any unique abbreviation for option is acceptable. The valid options are:

File atime name:

Returns a decimal string giving the time at which file name was last accessed. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried, then an error is generated.

File dirname name:

Returns all of the characters in name up to but not including the last slash character. If there are no slashes in name, then it returns ``.''. If the last slash in name is its first character, then it returns ``/''.

File executable name:

Returns 1 if file name is executable by the current user, 0 otherwise.

File exists name:

Returns 1 if file name exists and the current user has search privileges for the directories leading to it, 0 otherwise.

File extension name:
Returns all of the characters in name after and including the last dot in name. If there is no dot in the name, then it returns the empty string.

File is directory name:
Returns 1 if file name is a directory, 0 otherwise.

File is file name:
Returns 1 if file name is a regular file, 0 otherwise.

File is at name varName:

Same as the start option (see below) except that it uses the lstat kernel call instead of stat. This means that if name refers to a symbolic link, the information returned in varName is for the link rather than the file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the start option.

File time name:

Returns a decimal string giving the time at which file name was last modified. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

File owned name:
Returns 1 if file name is owned by the current user, 0 otherwise.

File readable name:
Returns 1 if file name is readable by the current user, 0 otherwise.

File readlink name:

Returns the value of the symbolic link given by name (i.e. the name of the file itpoints to). If name isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

File rootname name:

Returns all of the characters in name up to but not including the last ``." character in the name. If name doesn't contain a dot, then it returns name.

File size name:

Returns a decimal string giving the size of file name in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

File stat name varName:

Invokes the stat kernel call on name, and uses the variable given by varName to hold information returned from the kernel call. VarName is treated as an array variable, and the following elements of that variable are set: atime, ctime, dev, gid, ino, mode, mtime, nlink, size, type, uid. Each element except type is a decimal string with the value of the corresponding field from the stat return structure. Refer to the manual entry for stat for details on the meanings of the values. The type element gives the type of the file in the same form returned by the command file type. This command returns an empty string.

File tail name

Returns all of the characters in name after the last slash. If name contains no slashes then it returns name.

File type name:

Returns a string giving the type of file name, which will be one of file, directory, characterSpecial, blockSpecial, fifo, link, or socket.

File writable name:

Returns 1 if file name is writable by the current user, 0 otherwise.

19.6.17 flush

flush - Flush buffered output for a file

(TCL Commands)

SYNOPSIS: flush fileId

DESCRIPTION: Flushes any output that has been buffered for fileId. FileId must be the return value from a previous call to open, or it may be stdout or stderr to access one of the standard I/O streams; it must refer to a file that was opened for writing. The command returns an empty string.

19.6.18 foreach

for each - Iterate over all elements in a list

(TCL Commands)

SYNOPSIS: foreach varname list body

DESCRIPTION: In this command varname is the name of a variable, list is a list of values to assign to varname, and body is a TCL script. For each element of list (in order from left to right), foreach assigns the contents of the field to varname as if the lindex command had been used to extract the field, then calls the TCL Interpreter to execute body. The break and continue statements may be invoked inside body, with the same effect as in the for command. Foreach returns an empty string.

19.6.19 for

for - "For" loop

(TCL Commands)

SYNOPSIS: for start test next body

DESCRIPTION: For is a looping command, similar in structure to the C for statement. The start, next, and body arguments must be TCL command strings, and test is an expression string. The for command first invokes the TCL Interpreter to execute start. Then it repeatedly evaluates test as an expression; if the result is non-zero it invokes the TCL interpreter on body, then invokes the TCL Interpreter on next, then repeats the loop. The command terminates when test evaluates to 0. If a continue command is invoked within body then any remaining commands in the current execution of body are skipped; processing continues by invoking the TCL Interpreter on next, then evaluating test, and so on. If a break command is invoked within body or next, then the for command will return immediately. The operation of break and continue are similar to the corresponding statements in C. For returns an empty string.

19.6.20 format

format - Format a string in the style of sprintf

(TCL Commands)

SYNOPSIS: format formatString ?arg arg ...?

INTRODUCTION: This command generates a formatted string in the same way as the ANSI C sprintf procedure (it uses sprintf in its implementation). FormatString indicates how to format the result, using % conversion character signs as in sprintf, and additional arguments, if any, provide values to be substituted in the result. The return value from format is the formatted string.

DETAILS ON FORMATTING

The command operates by scanning formatString from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If the character is a % then it is not copied to the result string. Instead, the characters following the % character are treated as a conversion character sign. The conversion character sign controls the conversion of the next successive arg to a particular format and the result is appended to the result string in place of the conversion character sign. If there are multiple conversion character signs in the format string, then each one controls the conversion of one additional arg.

The format command must be given enough args to meet the needs of all of the conversion character signs in formatString. Each conversion character sign may contain up to six different parts:

an XPG3 position indicator, a set of flags, a minimum field width, a precision, a length modifier, and a conversion character.

Any of these fields may be omitted except for the conversion character. The fields that are present must appear in the order given above. The paragraphs below discuss each of these fields in turn. If the % is followed by a decimal number and a $, as in ``%2$d", then the value to convert is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first arg. If the conversion character sign requires multiple

arguments because of * characters in the indicator then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional indicators. If there are any positional indicators in formatString then all of the indicators must be positional.

The second portion of a conversion character sign may contain any of the following flag characters, in any order:

-

Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).

+

Specifies that a number should always be printed with a sign, even if positive.

space

Specifies that a space should be added to the beginning of the number if the first character isn't a sign.

0

Specifies that the number should be padded on the left with zeroes instead of spaces.

#

Requests an alternate output form. For o and O conversions, it guarantees that the first digit is always 0. For x or X conversions, 0x or 0X (respectively) will be added to the beginning of the result unless it is zero. For all floating-point conversions (e, E, f, g, and G) it guarantees that the result always has a decimal point. For g and G conversions, it specifies that trailing zeroes should not be removed. The third portion of a conversion character sign is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the minimum field width, then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the 0 and - flags may be used to specify padding

with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as * rather than a number, then the next argument to the format command determines the minimum field width; it must be a numeric string.

The fourth portion of a conversion character sign is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions. For e, E, and f conversions, it specifies the number of digits to appear to the right of the decimal point. For g and G conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omittedunless the # flag has been specified). For integer conversions, it specifies a minimum number of digits to print (leading zeroes will be added if necessary). For s conversions, it specifies the maximum number of characters to be printed; if the string is longer than this, then thetrailing characters will be dropped. If the precision is specified with * rather than a number, then the next argument to the format command determines the precision; it must be a numeric string. The fourth part of a conversion character sign is a length modifier, which must be h or l. If it is h it specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful. The l modifier is ignored.

The last thing in a conversion character sign is an alphabetic character that determines what kind of conversion to perform. The following conversion characters are currently supported:

d

Convert integer to signed decimal string.

u

Convert integer to unsigned decimal string.

i

Convert integer to signed decimal string; the integer may either be in decimal, in

octal (with a leading 0) or in hexadecimal (with a leading 0x).

o

Convert integer to unsigned octal string.

x or X

Convert integer to unsigned hexadecimal string, using digits ``0123456789abcdef"

for x and ``0123456789ABCDEF" for X).

c

Convert integer to the 8-bit character it represents.

s

No conversion; just insert string.

f

Convert floating-point number to signed decimal string of the form xx.yyy, where the number of ‘y’s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.

e or E

Convert floating-point number to scientific notation in the form x.yyye\(+-zz, where the number of y's is determined by the precision (default: 6). If the precision is 0 then no decimal point is output. If the E form is used then E is printed instead of e.

g or G

If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for %e or %E. Otherwise convert as for %f. Trailing zeroes and a trailing decimal point are omitted.

%
No conversion: just insert %.

For the numerical conversions the argument being converted must be an integer or floating-point string; format converts the argument to binary and then converts it back to a string according to the conversion character sign.

DIFFERENCES FROM ANSI SPRINTF

The behavior of the format command is the same as the ANSI C sprintf procedure except for the following differences:

[1]
%p and %n character signs are not currently supported.
[2]
For %c conversions the argument must be a decimal string, which will then be converted to the corresponding character value.
[3]
The l modifier is ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the l modifier were present (i.e. type double is used for the internal representation). If the h modifier is specified then integer values are truncated to short before conversion.

19.6.21 gets

gets - Read a line from a file

(TCL Commands)

SYNOPSIS: gets fileId ?varName?

DESCRIPTION: This command reads the next line from the file given by fileId and discards the terminating newline character. If varName is specified, then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then -1 is returned and varName is set to an empty string. If varName is not specified, then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so eof may have to be used to determine what really happened. If the last character in the file is not a newline character, then gets behaves as if there were an additional newline character at the end of the file. FileId must be stdin or the return value from a previous call to open; it must refer to a file that was opened for reading. Any existing end-of-file or error condition on the file is cleared at the beginning of the gets command.

19.6.22 glob

glob - Return names of files that match patterns

(TCL Commands)

SYNOPSIS: glob ?switches? pattern ?pattern ...?

DESCRIPTION: This command performs file name ``globbing'' in a fashion similar to the csh shell. It returns a list of the files whose names match any of the pattern arguments.

If the initial arguments to glob start with - then they are treated as switches. The following switches are currently supported:

-no complain

Allows an empty list to be returned without error; without this switch an error is returned if the result list would be empty.

--

Marks the end of switches. The argument following this one will be treated as a pattern even if it starts with a -.

The pattern arguments may contain any of the following special characters:

?

Matches any single character.

*

Matches any sequence of zero or more characters.

[chars]

Matches any single character in chars. If chars contains a sequence of the form a-b then any character between a and b (inclusive) will match.

\x

Matches the character x.

{a,b,...}

Matches any of the strings a, b, etc.

As with csh, a ``.'' at the beginning of a file's name or just after a ``/'' must be matched explicitly or with a {} construct. In addition, all ``/'' characters must be matched explicitly. If the first character

in a pattern is ``~'' then it refers to the home directory for the user whose name follows the ``~''. If the ``~'' is followed immediately by ``/'' then the value of the HOME environment variable is used. The glob command differs from csh globbing in two ways. First, it does not sort its result list (use the lsort command if the user wants the list sorted). Second, glob only returns the names of files that actually exist; in csh no check for existence is made unless a pattern contains a ?, *, or [] construct.

19.6.23 global

global - Access global variables

(TCL Commands)

SYNOPSIS: global varname ?varname ...?

DESCRIPTION: This command is ignored unless a TCL procedure is being interpreted. If so, then it declares the given varnames to be global variables rather than local ones. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the varnames will refer to the global variable by the same name.

19.6.24 history

history - Manipulate the history list

(TCL Commands)

SYNOPSIS: history ?option? ?arg arg ...?

DESCRIPTION: The history command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an ``event''. When specifying an event to the history command, the following forms may be used:

[1]
A number: if positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (-1 refers to the previous event, -2 to the one before that, and so on).

[2]
A string: selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the string match command. The history command can take any of the following forms:
history Same as history info, described below.

history add command ?exec?
Adds the command argument to the history list as a new event. If exec is specified (or abbreviated) then the command is also executed and its result is returned. If exec isn't specified, then an empty string is returned as a result, history change newValue ?event? replaces the value recorded for an event with newValue. Event specifies the event to replace, and defaults to the current event (not event -1). This command is intended for use in commands that implement new forms of history substitution and wish to replace the current event (which invokes the substitution) with the command created through substitution. The return value is an empty string.

history event ?event?

Returns the value of the event given by event. Event defaults to -1. This command causes history revision to occur: see below for details.

history info ?count?

Returns a formatted string (intended for humans to read) giving the event number and contents for each of the events in the history list except the current event. If count is specified then only the most recent count events are returned.

history keep count
This command may be used to change the size of the history list to count events. Initially, 20 events are retained in the history list. This command returns an emptystring.

history nextid
Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

history redo ?event?
Re-executes the command indicated by event and return its result. Event defaults to -1. This command results in history revision: see below for details.

history substitute old new ?event?
Retrieves the command given by event (-1 by default), replaces any occurrences of old by new in the command (only simple character equality is supported; no wild cards), executes the resulting command, and returns the result of that execution. This command results in a history revision: see below for details.

history words selector ?event?
Retrieves from the command given by event (-1 by default) the words given by selector, and returns those words in a string separated by spaces. The selector argument has three forms. If it is a single number, then it selects the word given by that number (0 for the command name,1 for its first argument, and so on). If it consists of two

numbers separated by a dash, then it selects all the arguments between those two. Otherwise,

selector is treated as a pattern; all words matching that pattern (in the sense of string match) are returned. In the numeric forms $ may be used to select the last word of a command. For example, suppose the most recent command in the history list is format {%s is %d years old} Alice [expr $ageInMonths/12]

Below are some history commands and the results they would produce:

history words $ [expr $ageInMonths/12]
history words 1-2 {%s is %d years old} Alice
history words *a*o* {%s is %d years old} [expr $ageInMonths/12]
History words results in history revision: see below for details.

HISTORY REVISION

The history options **event**, **redo**, **substitute** and **words** result in **``history revision''**. When one of these options is invoked, the current event is modified to eliminate the history command and replace it with the result of the history command. For example, suppose that the most recent command in the history list is:

set a [expr $b+2] and suppose that the next command invoked is one of those on the left side of the table below. The command actually recorded in the history event will be the corresponding one on the right side of the table.

history redo set a [expr $b+2]
history s a b set b [expr $b+2]
set c [history w 2] set c [expr $b+2]

History revision is needed because event indicators such as -1 are only valid at a particular time. Indeed, when more events have been added to the history list, a different event indicator would be needed. History revision occurs even when history is invoked indirectly from the current event (e.g. a user types a command that invokes a TCL procedure that invokes history); the top-level command whose execution eventually resulted in a history command is replaced. If the user wishes to invoke commands like history words without history revision, he can use history event to save the current history event and then use history change to restore it later.

19.6.25 if

if - Execute scripts conditionally

(TCL Commands)

SYNOPSIS: if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?

DESCRIPTION: The if command evaluates expr1 as an expression (in the same way that expr evaluates its argument). The value of the expression must be a Boolean (a numeric value, where 0 is false and anything is true, or a string value such as true or yes for true and false or no for false); if it is true then body1 is executed by passing it to the TCL interpreter. Otherwise expr2 is evaluated as an expression and if it is true then body2 is executed, and so on. If none of the expressions evaluates to true, then bodyN is executed. The then and else arguments are optional ``noise words" to make the command easier to read. There may be any number of elseif clauses, including zero. BodyN may also be omitted as long as else is omitted too. The return value from the command is the result of the body script that was executed, or an empty string, if none of the expressions was non-zero and there was no bodyN.

19.6.26 incr

incr - Increment the value of a variable

(TCL Commands)

SYNOPSIS: incr varName ?increment?

DESCRIPTION: Increments the value stored in the variable whose name is varName. The value of the variable must be an integer. If increment is supplied, then its value (which must be an integer) is added to the value of variable varName; otherwise 1 is added to varName. The new value is stored as a decimal string in variable varName and is also returned as result.

19.6.27 info

info - Return information about the state of the TCL interpreter

(TCL Commands)

SYNOPSIS: info option ?arg arg ...?

DESCRIPTION: This command provides information about various internals of the TCL interpreter. The legal options (which may be abbreviated) are:

info args procname

Returns a list containing the names of the arguments to procedure procname, in rder. Procname must be the name of a TCL command procedure.

info body procname

Returns the body of procedure procname. Procname must be the name of a TCL command procedure.

info cmdcount
Returns a count of the total number of commands that have been invoked in this interpreter.

info commands ?pattern?

If a pattern is not specified, it returns a list of names of all the TCL commands, including both the built-in commands written in C and the command procedures defined using the proc command. If a pattern is specified, only those names matching the pattern are returned. Matching is determined using the same rules as for string matches.

info complete command

Returns 1 if command is a complete TCL command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command does not appear to be complete, then 0 is returned. This command is typically used in line- oriented input environments to allow users to type in commands that span multiple lines. If the command isn't complete, the script can delay evaluating it until additional lines have been typed to complete the command. info default procname arg varname Procname must be the name of a TCL command procedure and arg must

be the name of an argument to that procedure. If arg doesn't have a default value then the command returns 0. Otherwise it returns 1 and places the default value of arg in variable varname.

info exists varName Returns 1 if the variable named varName exists in the current context (either as a global or local variable), returns 0 otherwise.

info globals ?pattern?

If a pattern isn't specified, it returns a list of all the names of the currently-defined global variables. If a pattern is specified, only those names matching the pattern are returned.

Matching is determined using the same rules as for string match.

info level ?number?

If a number is not specified, this command returns a number giving the stack level of the invoking procedure, or 0 if the command is invoked at top-level. If a number is specified, then the result is a list consisting of the name and arguments for the procedure call at level number on the stack. If the number is positive, then it selects a particular stack level (1 refers to the top-most active procedure, 2 to the procedure it called, and so on); otherwise it gives a level relative to the current level (0 refers to the current procedure, -1 to its caller, and so on). See the uplevel command for more information on what stack levels mean.

info library

Returns the name of the library directory in which standard TCL scripts are stored.

The default value for the library is compiled into TCL, but it may be overridden by setting the TCL_LIBRARY environment variable. If there is no TCL_LIBRARY variable and no compiled-in value, then an error is generated. See the library manual entry for details of the facilities provided by the TCL script library. Normally each application will have its own application-specific script library in addition to the TCL script
library. We suggest that each application set a global variable with a name like $app_library (where app is the application's name) to hold the location of that application's library directory.

info locals ?pattern?

If a pattern isn't specified, it returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the global and upvar commands will not be returned. If a pattern is specified, only those names matching the pattern are returned. Matching is determined using the same rules as for string matches.

info patchlevel

Returns a decimal integer giving the current patch level for TCL. The patch level is incremented for each new release or patch, and it uniquely identifies an official version of TCL.

info procs ?pattern?

If pattern isn't specified, it returns a list of all the names of TCL command procedures. If a pattern is specified, only those names matching the pattern are returned. Matching is determined using the same rules as for string matches.

info script

If a TCL script file is currently being evaluated (i.e. there is a call to Tcl_EvalFile active or there is an active invocation of the source command), then this command returns the name of the innermost file being processed. Otherwise the command returns an empty string.

info tclversion

Returns the version number for this version of TCL in the form x.y, where changes to x represent major changes with probable incompatibilities and changes to y represent small enhancements and bug fixes that retain backward compatibility.

info vars ?pattern?

If a pattern isn't specified, it returns a list of all the names of currently-visible variables, including both locals and currently-visible globals. If a pattern is specified, only those names matching the pattern are returned. Matching is determined using the same rules as for string matches.

19.6.28 join

join - Create a string by joining together list elements

(TCL Commands)

SYNOPSIS: join list ?joinString?

DESCRIPTION: The list argument must be a valid TCL list. This command returns the string formed by joining all of the elements of list together with joinString separating each adjacent pair of elements. The joinString argument defaults to a space character.

19.6.29 lappend

lappend - Append list elements onto a variable

(TCL Commands)

SYNOPSIS: lappend varName value ?value value ...?

DESCRIPTION: This command treats the variable given by varName as a list and appends each of the value arguments to that list as a separate element, with spaces between elements. If varName doesn't exist, it is created as a list with elements given by the value arguments. Lappend is similar to append, except that the values are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists. For example, ``lappend a $b" is much more efficient than ``set a [concat $a [list $b]]" when $a is long.

19.6.30 library

library - standard library of TCL procedures

(TCL Commands)

SYNOPSIS: auto_execok cmd

auto_load cmd
auto_mkindex dir pattern pattern ...
auto_reset
parray arrayName
unknown cmd ?arg arg ...?

INTRODUCTION: TCL includes a library of TCL procedures for commonly-needed functions. The procedures defined in the TCL Library are generic ones suitable for use by many different applications. The location of the TCL Library is returned by the info library command. In addition to the TCL Library, each application will normally have its own library of support procedures as well. The location of this library is normally given by the value of the $app_library global variable, where app is the name of the application. For example, the location of the Tk Library is kept in the variable $tk_library. To access the procedures in the TCL Library, an application should source the file init.tcl in the library; for example, with the TCL command source [info library]/init.tcl. This will define the unknown procedure and arrange for the other procedures to be loaded on-demand using the auto-load mechanism defined
below.

COMMAND PROCEDURES

The following procedures are provided in the TCL Library:

auto_execok cmd

Determines whether there is an executable file by the name cmd. This command examines the directories in the current search path (given by the PATH environment variable) to see if there is an executable file named cmd in any of those directories. If so, it returns 1; if not it returns 0. Auto_exec remembers information about previous searches in an array named auto_execs; this avoids the path search in future calls for the same cmd. The command auto_reset may be used to force auto_execok to forget its cached information.

auto_load cmd

This command attempts to load the definition for a TCL command named cmd. To do this, it searches an auto-load path, which is a list of one or more directories. The auto-load path is given by the global variable $auto_path if it exists. If there is no $auto_path variable, then the TCLLIBPATH environment variable is used, if it exists. Otherwise the auto-load path consists of just the TCL library directory. Within each directory in the auto-load path there must be a file tclIndex that describes one or more commands defined in that directory and a script to evaluate to load each of the commands. The tclIndex file should be generated with the auto_mkindex command. If cmd is found in an index file, then the appropriate script is evaluated to create the command. The auto_load command returns 1 if cmd was successfully created. The command returns 0 if there was no index entry for cmd or if the script didn't actually define cmd (e.g. because index information is out of date). If an error occurs while processing the script, then that error is returned. Auto_load only reads the index information once and saves it in the array auto_index; future calls to auto_load check for cmd in the array rather than re-reading the index files. The cached index information may be deleted with the command auto_reset. This will force the next auto_load command to reload the index database from disk.

auto_mkindex dir pattern ...

Generates an index suitable for use by auto_load. The command searches dir for all files whose names match any of the pattern arguments (matching is done with the glob command), generates an index of all the TCL command procedures defined in all the matching files, and stores the

index information in a file named tclIndex in dir. For example, the command auto_mkindex foo *.tcl will read all the .tcl files in subdirectory foo and generate a new index file foo/tclIndex. Auto_mkindex parses the TCL scripts in a relatively unsophisticated way: if any line contains the word proc as its first characters then it is assumed to be a procedure definition and the next word of the line is taken as the procedure's name. Procedure definitions that don't appear in this way (e.g. they have spaces before the proc) will not be indexed.

auto_reset

Destroys all the information cached by auto_execok and auto_load. This information will be re-read from disk the next time it is needed. Auto_reset also deletes any procedures listed in the auto-load index, so that fresh copies of them will be loaded the next time that they are used. Parray arrayName Prints on standard output the names and values of all the elements in the array arrayName. ArrayName must be an array accessible to the caller of parray. It may be either local or global.

unknown cmd ?arg arg ...?

This procedure is invoked automatically by the TCL Interpreter whenever the name of a command doesn't exist. The unknown procedure receives as its arguments the name and arguments of the missing command. Unknown first calls auto_load to load the command. If this succeeds, then it executes the original command with its original arguments. If the auto-load fails then unknown calls auto_execok to see if there is an executable file by the name cmd. If so, it invokes the TCL exec command with cmd and all the args as arguments. If cmd cannot be auto-executed, unknown checks to see if the command was invoked at top-level and outside of any script. If so, then unknown takes two additional steps. First, it sees if cmd has one of the following three forms: !!, !event, or ^old^new?^?. If so, unknown carries out history substitution in the same way that csh would for these constructs. Second, and last, unknown checks to see if cmd is a unique abbreviation for an existing TCL command. If so, it expands the command name and executes the command with the original arguments. If none of the above efforts has been able to execute the command, unknown generates an error return. If the global variable auto_noload is defined, then the auto-load step is skipped. If the global variable auto_noexec is defined, the auto-exec step is

skipped. Under normal circumstances the return value from unknown is the return value from the command that was eventually executed.

VARIABLES

The following global variables are defined or used by the procedures in the TCL library:

auto_execs: used by auto_execok to record information about whether particular commands exist as executable files.

auto_index: used by auto_load to save the index information read from disk.

auto_noexec: if set to any value, unknown will not attempt to auto-exec any commands.

auto_noload: if set to any value, unknown will not attempt to auto-load any commands.

auto_path: if set, it must contain a valid TCL list giving directories to search during auto-load operations.

env(TCL_LIBRARY): if set, it specifies the location of the directory containing library scripts (the value of this variable will be returned by the command info library). If this variable isn't set, a default value is used.

env(TCLLIBPATH): if set, it must contain a valid TCL list giving directories to search during auto-load operations. This variable is only used if auto_path is not defined.unknown_active: this variable is set by unknown to indicate that it is active. It is used to detect errors where unknown recourses on itself infinitely. The variable is unset before unknown returns.

19.6.31 lindex

lindex - Retrieve an element from a list

(TCL Commands)

SYNOPSIS: lindex list index

DESCRIPTION: This command treats list as a TCL list and returns the index'th element from it (0 refers to the first element of the list). In extracting the element, lindex observes the same rules concerning braces and quotes and backslashes as the TCL command interpreter; however, variable substitution and command substitution do not occur. If the index is negative or greater than or equal to the number of elements in value, then an empty string is returned.

19.6.32 linsert

linsert - Insert elements into a list

(TCL Commands)

SYNOPSIS: linsert list index element ?element element ...?

DESCRIPTION: This command produces a new list from list by inserting all of the element arguments just before the indexth element of list. Each element argument will become a separate element of the new list. If index is less than or equal to zero, then the new elements are inserted at the beginning of the list. If index is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

19.6.33 list

list - Create a list

(TCL Commands)

SYNOPSIS: list ?arg arg ...?

DESCRIPTION: This command returns a list comprised of all the args, or an empty string if no args are specified. Braces and backslashes are added as necessary, so that the index command may be used on the result to re-extract the original arguments, and also so that eval may be used to execute the resulting list, with arg1 comprising the command's name and the other args comprising its arguments. List produces slightly different results than concat: concat removes one level of grouping before forming the list, while list works directly from the original arguments. For example: the command list a b {c d e} {f {g h}} will return

a b {c d e} {f {g h}}
while concat with the same arguments will return

a b c d e f {g h}.

19.6.34 llength

llength - Count the number of elements in a list

(TCL Commands)

SYNOPSIS: llength list

DESCRIPTION: Treats list as a list and returns a decimal string giving the number of elements in it.

19.6.35 lrange

lrange - Return one or more adjacent elements from a list

(TCL Commands)

SYNOPSIS: lrange list first last

DESCRIPTION: List must be a valid TCL list. This command will return a new list consisting of elements first through last, inclusive. Last may be typed as end (or any abbreviation of it) to refer to the last element of the list. If first is less than zero, it is treated as if it were zero. If last is greater than or equal to the number of elements in the list, then it is treated as if it were end. If first is greater than last, then an empty string is returned. Note: ``lrange list first first'' does not always produce the same result as ``lindex list first'' (although it often does for simple fields that aren't enclosed in braces); it does, however, produce exactly the same results as ``list [lindex list first]''.

19.6.36 lreplace

lreplace - Replace elements in a list with new elements

(TCL Commands)

SYNOPSIS: lreplace list first last ?element ...?

DESCRIPTION: Lreplace returns a new list formed by replacing one or more elements of list with the element arguments. First gives the index in list the first element to be replaced. If first is less than zero, then it refers to the first element of list (the element indicated by first must exist in the list). Last gives the index in list of the last element to be replaced (it must be greater than or equal to first). Last may be end (or any abbreviation of it) to indicate that all elements between first and the end of the list should be replaced. The element arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each element argument will become a separate element of the list. If no element arguments are specified, then the elements between first and last are simply deleted.

19.6.37 lsearch

lsearch - See if a list contains a particular element

(TCL Commands)

SYNOPSIS: lsearch ?mode? list pattern

DESCRIPTION: This command searches the elements of list to see if one of them matches a pattern. If so, the command returns the index of the first matching element. If not, the command returns -1. The mode argument indicates how the elements of the list are to be matched against the pattern and it must have one of the following values:

-exact

The list element must contain exactly the same string as pattern.

-glob

Pattern is a glob-style pattern which is matched against each list element using the same rules as the string match command.

-regexp

Pattern is treated as a regular expression and matched against each list element using the same rules as the regexp command.

If mode is omitted then it defaults to -glob.

19.6.38 lsort

lsort - Sort the elements of a list

(TCL Commands)

SYNOPSIS: lsort ?switches? list

DESCRIPTION: This command sorts the elements of list, returning a new list in sorted order. By default ASCII sorting is used with the result returned in increasing order. However, any of the following switches may be specified before list to control the sorting process (unique abbreviations are accepted):
-ascii

Use string comparison with ASCII collation order. This is the default.

-integer

Convert list elements to integers and use integer comparison.

-real

Convert list elements to floating-point values and use floating comparison.

-command command

Use command as a comparison command. To compare two elements, evaluate a TCL script consisting of command with the two elements appended as additional arguments. The script should return an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively.

-increasing

Sort the list in increasing order (``smallest" items first). This is the default.

-decreasing
Sort the list in decreasing order (``largest" items first).

19.6.39 open

open - Open a file

(TCL Commands)

SYNOPSIS: open fileName ?access? ?permissions?

DESCRIPTION: This command opens a file and returns an identifier that may be used in future invocations of commands such as read, puts, and close. FileName gives the name of the file to open; if it starts with a tilde then tilde substitution is performed as described for Tcl_TildeSubst. If the first character of fileName is ``|'' then the remaining characters of fileName are treated as a command pipeline to invoke, in the same style as for exec. In this case, the identifier returned by open may be used to write to the command's input pipe or read from its output pipe.

The access argument indicates the way in which the file (or command pipeline) is to be accessed. It may take two forms, either a string in the form that would be passed to the fopen library procedure or a list of POSIX access flags. It defaults to ``r''. In the first form access may have any of the following values:

r Open the file for reading only; the file must already exist.
r+ Open the file for both reading and writing; the file must already exist.
w Open the file for writing only. Truncate it if it exists. If it doesn't exist, create a new file.
w+ Open the file for reading and writing. Truncate it if it exists. If it doesn't exist, create a new file.
a Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.
a+ Open the file for reading and writing. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.
In the second form, access consists of a list of any of the following flags, all of which have the standard POSIX meanings. One of the flags must be either RDONLY, WRONLY or RDWR.

RDONLY
Open the file for reading only.

WRONLY
Open the file for writing only.

RDWR

Open the file for both reading and writing.
APPEND
Set the file pointer to the end of the file prior to each write.

CREAT
Create the file if it doesn't already exists (This access option must be included if the file does not exist).

EXCL
If CREAT is specified also, an error is returned if the file already exists.

NOCTTY
If the file is a terminal device, this flag prevents the file from becoming the controlling terminal of the process.

NONBLOCK
Prevents the process from blocking while opening the file. For details refer to the system documentation on the open system call's O_NONBLOCK flag.

TRUNC
If the file exists, it is truncated to zero length. If a new file is created as part of opening it, permissions (an integer) is used to set the permissions for the new file in conjunction with the process's file mode creation mask. Permissions defaults to 0666. If a file is opened for both reading and writing, then seek must be invoked between a read and a write, or vice versa (this restriction does not apply to command pipelines opened with open). When fileName specifies a command pipeline and a write-only access is used, then standard output from the pipeline is directed to the current standard output unless overridden by the command. When fileName specifies a command pipeline and a read-only access is used, then standard input from the pipeline is taken from the current standard input unless overridden by the command.

19.6.40 pid

pid - Retrieve process id(s)

(TCL Commands)

SYNOPSIS: pid ?fileId?

DESCRIPTION: If the fileId argument is given, then it should normally refer to a process pipeline created with the open command. In this case the pid command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if fileId refers to an open file that isn't a process pipeline. If no fileId argument is given then pid returns the process identifier of the current process. All process identifiers are returned as decimal strings.

19.6.41 proc

proc - Create a TCL procedure

(TCL Commands)

SYNOPSIS: proc name args body

DESCRIPTION: The proc command creates a new TCL procedure named name, replacing any existing command or procedure there may have been by that name. Whenever the new command is invoked, the contents of body will be executed by the TCL interpreter. Args specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument indicator is also a list with either one or two fields. If there is only a single field in the indicator then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value.
When name is invoked a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name args, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to args are combined into a list (as if the list command had been used); this combined value is assigned to the local variable args. When body is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the global command or the upvar command. The proc command returns an empty string. When a procedure is invoked, the procedure's return value is the value specified in a return command. If the procedure doesn't execute an explicit return, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

19.6.42 puts

puts - Write to a file

(TCL Commands)

SYNOPSIS: puts ?-nonewline? ?fileId? string

DESCRIPTION: Writes the characters given by string to the file given by fileId. FileId must have been the return value from a previous call to open, or it may be stdout or stderr to refer to one of the standard I/O channels; it must refer to a file that was opened for writing. If no fileId is specified then it defaults to stdout. Puts normally outputs a newline character after string, but this feature may be suppressed by specifying the -nonewline switch. Output to files is buffered internally by TCL; the flush command may be used to force buffered characters to be output.

19.6.43 pwd

pwd - Return the current working directory

(TCL Commands)

SYNOPSIS: pwd

DESCRIPTION: Returns the path name of the current working directory.

19.6.44 read

read - Read from a file

(TCL Commands)

SYNOPSIS:

read ?-nonewline? fileId

read fileId numBytes

DESCRIPTION: In the first form, all of the remaining bytes are read from the file given by fileId; they are returned as the result of the command. If the -nonewline switch is specified, then the last character of the file is discarded if it is a newline. In the second form, the extra argument specifies how many bytes to read; exactly that many bytes will be read and returned, unless there are fewer than numBytes bytes left in the file; in this case, all the remaining bytes are returned. FileId must be stdin or the return value from a previous call to open; it must refer to a file that was opened for reading. Any existing end-of-file or error condition on the file is cleared at the beginning of the read command.

19.6.45 regexp

regexp - Match a regular expression against a string

(TCL Commands)

SYNOPSIS: regexp ?switches? exp string ?matchVar? ?subMatchVar subMatchVar ...?

DESCRIPTION: Determines whether the regular expression exp matches part or all of string and returns 1 if it does, 0 if it doesn't. If additional arguments are specified after string, then they are treated as the names of variables in which to return information about which part(s) of string matched exp. MatchVar will be set to the range of string that matched all of exp. The first subMatchVar will contain the characters in string that matched the leftmost parenthesized subexpression within exp, the next subMatchVar will contain the characters that matched the next parenthesized subexpression to the right in exp, and so on.

If the initial arguments to regexp start with - then they are treated as switches. The following switches are currently supported:

-nocase

Causes upper-case characters in string to be treated as lower case during the matching process.

-indices

Changes what is stored in the subMatchVars. Instead of storing the matching characters from string, each variable will contain a list of two decimal strings giving the indices in string of the first and last characters in the matching range of characters.

--

Marks the end of switches. The argument following this one will be treated as exp even if it starts with a -.

If there are more subMatchVar's than parenthesized subexpressions within exp, or if a particular subexpression in exp doesn't match the string (e.g. because it was in a portion of the expression that wasn't matched), then the corresponding subMatchVar will be set to ``-1 -1" if -indices have been specified; to an empty string otherwise.

REGULAR EXPRESSIONS

Regular expressions are implemented using Henry Spencer's package, and much of the description of regular expressions below is copied verbatim from his manual entry. A regular expression is zero or more branches, separated by ``|". It matches anything that matches one of the branches. A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by ``*", ``+", or ``?". An atom followed by ``*" matches a sequence of 0 or more matches of the atom. An atom followed by ``+" matches a sequence of 1 or more matches of the atom. An atom followed by ``?" matches a match of the atom, or the null string. An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), ``." (matching any single character), ``^" (matching the null string at the beginning of the input string), ``$" (matching the null string at the end of the input string), a ``\" followed by a single character (matching that character), or a single character with no other significance (matching that character).

A range is a sequence of characters enclosed in ``[]". It normally matches any single character from the sequence. If the sequence begins with ``^", it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by ``-", this is shorthand for the full list of ASCII characters between them (e.g. ``[0-9]" matches any decimal digit). To include a literal ``]" in the sequence, make it the first character (following a possible ``^"). To include a literal ``-", make it the first or last character.

CHOOSING AMONG ALTERNATIVE MATCHES

In general there may be more than one way to match a regular expression to an input string. For example, consider the command regexp (a*)b* aabaaabb x y

Considering only the rules given so far, x and y could end up with the values aabb and aa, aaab and aaa, ab and a, or any of several other combinations. To resolve this potential ambiguity regexp selects from among alternatives using the rule ``first then longest". In other words, it considers the possible matches in order, working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones.

More specifically, the following rules apply in decreasing order of priority:

[1] If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

[2] If a regular expression contains | operators then the left-most matching sub-expression is selected.

[3] In *, +, and ? constructs, longer matches are selected in preference to shorter ones.

[4] In sequences of expression components the components are considered from left to right. In the example from above, (a*)b* matches aab: the (a*) portion of the pattern is matched first and it consumes the leading aa; then the b* portion of the pattern consumes the next b. Or, consider the following example: regexp (ab|a)(b*)c abc x y z. After this command x will be abc, y will be ab, and z will be an empty string. Rule 4 specifies that (ab|a) gets first shot at the input string and Rule 2 specifies that the ab sub-expression is checked before the a sub-expression. Thus the b has already been claimed before the (b*) component is checked and (b*) must match an empty string.

19.6.46 regsub

regsub - Perform substitutions based on regular expression pattern matching

(TCL Commands)

SYNOPSIS: regsub ?switches? exp string subSpec varName

DESCRIPTION: This command matches the regular expression exp against string, and it copies string to the variable whose name is given by varName. The command returns 1 if there is a match and 0 if there is not a match. If there is a match, then while copying string to varName the portion of string that matched exp is replaced with subSpec. If subSpec contains a ``&''
or ``\0'', then it is replaced in the substitution with the portion of string that matched exp. If subSpec contains a ``\n'', where n is a digit between 1 and 9, then it is replaced in the substitution with the portion of string that matched the n^{th} parenthesized subexpression of exp. Additional backslashes may be used in subSpec to prevent special interpretation of ``&'' or ``\0'' or ``\n'' or backslash. The use of backslashes in subSpec tends to interact badly with the TCL parser's use of backslashes, so it is generally safest to enclose subSpec in braces if it includes backslashes. If the initial arguments to regexp start with - then they are treated as switches. The following switches are currently supported: -all. All ranges in string that match exp are found and substitution is performed for each of these ranges. Without this switch only the first matching range is found and substituted. If -all is specified, then ``&'' and ``\n'' sequences are handled for each substitution using the information from the corresponding match.

-nocase

Upper-case characters in string will be converted to lower-case before matching against exp; however, substitutions specified by subSpec use the original unconverted form of string.

--Marks the end of switches. The argument following this one will be treated as exp even if it starts with a -. See the manual entry for regexp for details on the interpretation of regular expressions.

19.6.47 rename

rename - Rename or delete a command

(TCL Commands)

SYNOPSIS: rename oldName newName

DESCRIPTION: Rename the command that used to be called oldName so that it is now called newName. If newName is an empty string then oldName is deleted. The rename command returns an empty string as result.

19.6.48 return

return - Return from a procedure

(TCL Commands)

SYNOPSIS: return ?-code code? ?-errorinfo info? ?-errorcode code? ?string?

DESCRIPTION: Return immediately from the current procedure (or top-level command or Source command), with string as the return value. If string is not specified then an empty string will be returned as result.

EXCEPTIONAL RETURNS

In the usual case where the -code option isn't specified, the procedure will return normally (its completion code will be TCL_OK). However, the -code option may be used to generate an exceptional return from the procedure. Code may have any of the following values: ok

Normal return: same as if the option is omitted. error

Error return: same as if the error command were used to terminate the procedure, except for handling of errorInfo and errorCode variables (see below).

return. The current procedure will return with a completion code of

TCL_RETURN

so that the procedure that invoked it will return also.

break the current procedure will return with a completion code of TCL_BREAK, which will terminate the innermost nested loop in the code that invoked the current procedure.

CONTINUE

The current procedure will return with a completion code of TCL_CONTINUE, which will terminate the current iteration of the innermost nested loop in the code that invoked the current procedure.

VALUE

Value must be an integer; it will be returned as the completion code for the current

procedure. The code option is rarely used. It is provided so that procedures that implement new control structures can reflect exceptional conditions back to their callers.

Two additional options, -errorinfo and -errorcode, may be used to provide additional information during error returns. These options are ignored unless code is error. The -errorinfo option specifies an initial stack trace for the errorInfo variable; if it is not specified then the stack trace left in errorInfo will include the call to the procedure and higher levels on the stack but it will not include any information about the context of the error within the procedure. Typically the info value is supplied from the value left in errorInfo after a catch command trapped an error within the procedure. If the -errorcode option is specified then code provides a value for the errorCode variable. If the option is not specified then errorCode will default to NONE.

19.6.49 scan

scan - Parse string using conversion character signs in the style of sscanf

(TCL Commands)

SYNOPSIS: scan string format varName ?varName ...?

INTRODUCTION: This command parses fields from an input string in the same fashion as the ANSI C sscanf procedure and returns a count of the number of fields successfully parsed. String gives the input to be parsed and format indicates how to parse it, using % conversion character signs as in sscanf. Each varName gives the name of a variable; when a field is scanned from string the result is converted back into a string and assigned to the corresponding variable.

DETAILS ON SCANNING

Scan operates by scanning string and formatString together. If the next character in formatString is a blank or tab then it is ignored. Otherwise, if it isn't a % character then it must match the next non-white-space character of string. When a % is encountered in formatString, it indicates the start of a conversion character sign. A conversion character sign contains
three fields after the %: a *, which indicates that the converted value is to be discarded instead of assigned to a variable; a number indicating a maximum field width; and a conversion character. All of these fields are optional except for the conversion character. When scan finds a conversion character sign in formatString, it first skips any white-space characters in string. Then it converts the next input characters according to the conversion character sign and stores the result in the variable given by the next argument to scan. The following conversion characters are supported:

d The input field must be a decimal integer. It is read in and the value is stored in the variable as a decimal string.

o The input field must be an octal integer. It is read in and the value is stored in the variable as a decimal string.

x The input field must be a hexadecimal integer. It is read in and the value is stored in the variable as a decimal string.

c A single character is read in and its binary value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.

s The input field consists of all the characters up to the next white-space character; the characters are copied to the variable.

e or f or g The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly containing a decimal point, and an optional exponent consisting of an e or E followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point string.

[chars]

The input field consists of any number of characters in chars. The matching string is stored in the variable. If the first character between the brackets is a] then it is treated as part of chars rather than the closing bracket for the set.

[^chars]

The input field consists of any number of characters not in chars. The matching string is stored in the variable. If the character immediately following the ^ is a] then it is treated as part of the set rather than the closing bracket for the set.

The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion (i.e. as many decimal digits as possible for %d, as many octal digits as possible for %o, and so on). The input field for a given conversion terminates either when a white-space character is encountered or when the maximum field width has been reached, whichever comes first. If a * is present in the conversion character sign then no variable is assigned and the next scan argument is not consumed.

DIFFERENCES FROM ANSI SSCANF

The behavior of the scan command is the same as the behavior of the ANSI C sscanf procedure except for the following differences:

[1]
%p and %n conversion character signs are not currently supported.

[2]
For %c conversions, a single character value is converted to a decimal string, which is then assigned to the corresponding varName; no field width may be specified for this conversion.

[3]
The l, h, and L modifiers are ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the l modifier were present (i.e. type double is used for the internal representation).

19.6.50 seek

seek - Change the access position for an open file

(TCL Commands)

SYNOPSIS: seek fileId offset ?origin?

DESCRIPTION: Changes the current access position for fileId. FileId must have been the return value from a previous call to open, or it may be stdin, stdout, or stderr to refer to one of the standard I/O channels. The offset and origin arguments specify the position at which the next read or write will occur for fileId. Offset must be an integer (which may be negative) and origin must be one of the following:

start The new access position will be offset n bytes from the start of the file.

current The new access position will be offset n bytes from the current access position; a negative offset moves the access position backwards in the file.

end The new access position will be offset n bytes from the end of the file. A negative offset places the access position before the end-of-file, and a positive offset places the access position after the end-of-file. The origin argument defaults to start. This command returns an empty string.

19.6.51 set

set - Read and write variables

(TCL Commands)

SYNOPSIS: set varName ?value?

DESCRIPTION: Returns the value of variable varName. If value is specified, then sets the value of varName to value, creating a new variable if one does not already exist, and returns its value. If varName contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise varName refers to a scalar variable. If no procedure is active, then varName refers to a global variable. If a procedure is active, then varName refers to a parameter or localvariable of the procedure unless the global command has been invoked to declare varName to be global.

19.6.52 source

source - Evaluate a file as a TCL script

(TCL Commands)

SYNOPSIS: source fileName

DESCRIPTION: Read file fileName and pass the contents to the TCL Interpreter as a script to evaluate in the normal fashion. The return value from source is the return value of the last command executed from the file. If an error occurs in evaluating the contents of the file then the source command will return that error. If a return command is invoked from within the file then the remainder of the file will be skipped and the source command will return normally with the result from the return command. If fileName starts with a tilde, then it is tilde-substituted as described in the Tcl_TildeSubst manual entry.

19.6.53 split

split - Split a string into a proper TCL list

(TCL Commands)

SYNOPSIS: split string ?splitChars?

DESCRIPTION: Returns a list created by splitting string at each character that is in the splitChars argument. Each element of the result list will consist of the characters from string that lie between instances of the characters in splitChars. Empty list elements will be generated if string contains adjacent characters in splitChars, or if the first or last character of string is in splitChars. If splitChars is an empty string then each character of string becomes a separate element of the result list. SplitChars defaults to the standard white-space characters. For example:

split "comp.unix.misc" .

returns "comp unix misc" and

split "Hello world" {}

returns "H e l l o { } w o r l d".

19.6.54 string

string - Manipulate strings

(TCL Commands)

SYNOPSIS: string option arg ?arg ...?

DESCRIPTION: Performs one of several string operations, depending on the option selected. The legal options (which may be abbreviated) are:

string compare string1 string2

Perform a character-by-character comparison of strings string1 and string2 in the same way as the C strcmp procedure. Return -1, 0, or 1, depending on whether string1 is lexicographically less than, equal to, or greater than string2.

string first string1 string2.

Search string2 for a sequence of characters that exactly match the characters in string1. If found, return the index of the first character in the first such match within string2. If not found, return -1.

string index string charIndex Returns the charIndexth character of the string argument. A charIndex of 0 corresponds to the first character of the string. If charIndex is less than 0 or greater than or equal to the length of the string, then an empty string is returned.

string last string1 string2

Search string2 for a sequence of characters that exactly match the characters in string1. If found, return the index of the first character in the last such match within string2. If there is no match, then return -1.

string length string

Returns a decimal string giving the number of characters in string.

string match pattern string

Sees if the pattern matches the string; return 1 if it does, 0 if it doesn't. Matching is done in a

fashion
similar to that used by the C-shell. For the two strings to match, their contents must be identical except that the following special sequences may appear in the pattern:

*

Matches any sequence of characters in string, including a null string.

?

Matches any single character in string.

[chars]

Matches any character in the set given by chars. If a sequence of the form x-y appears in chars, then any character between x and y, inclusive, will match.

\x

Matches the single character x. This provides a way of avoiding the special interpretation of the characters *?[]\ in the pattern.

string range string first last

Returns a range of consecutive characters from string, starting with the character whose index is first and ending with the character whose index is last. An index of 0 refers to the first character of the string. Last may be end (or any abbreviation of it) to refer to the last character of the string. If first is less than zero then it is treated as if it were zero, and if last is greater than or equal to the length of the string then it is treated as if it were end. If first is greater than last then an empty string is returned.

string tolower string

Returns a value equal to string except that all upper case letters have been converted to lower case.

string toupper string

Returns a value equal to string except that all lower case letters have been converted to upper case.

string trim string ?chars?

Returns a value equal to string except that any leading or trailing characters from the set given by chars are removed. If chars is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimleft string ?chars?

Returns a value equal to string except that any leading characters from the set given by chars are removed. If chars is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimright string ?chars?

Returns a value equal to string except that any trailing characters from the set given by chars are removed. If chars is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

19.6.55 switch

switch - Evaluate one of several scripts, depending on a given value

(TCL Commands)

SYNOPSIS:

switch ?options? string pattern body ?pattern body ...?
switch ?options? string {pattern body ?pattern body ...?}

DESCRIPTION: The switch command matches its string argument against each of the pattern arguments in order. As soon as it finds a pattern that matches string, it evaluates the following body argument by passing it recursively to the TCL Interpreter and returns the result of that evaluation. If the last pattern argument is default then it matches anything. If no pattern argument matches string and no default is given, then the switch command returns an empty string.

If the initial arguments to switch start with - then they are treated as options. The following options are currently supported:

-exact

Use exact matching when comparing string to a pattern. This is the default.

-glob

When matching string to the patterns, use glob-style matching (i.e. the same as implemented by the string match command).

-regexp

When matching string to the patterns, use regular expression matching (i.e. the same as implemented by the regexp command).

--

Marks the end of options. The argument following this one will be treated as string even if it starts with a -.

Two syntaxes are provided for the pattern and body arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are

desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument. The argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line switch commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the pattern arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some
cases.

If a body is specified as ``-" it means that the body for the next pattern should also be used as the body for this pattern (if the next pattern also has a body of ``-" then the body after that is used, and so on). This feature makes it possible to share a single body among several patterns. Below are some examples of switch commands:

switch abc a - b {format 1} abc {format 2} default {format 3} will return 2,

switch -regexp aaab {^a.*b$ -b {format 1} a* {format2}default {format3}}willreturn 1, and
switch xyz {a-b{format 1}a*{format 2} default {format 3}} will return 3.

19.6.56 TCL TildeSubst

TCL TildeSubst - replace tilde with home directory in a file name

(TCL Commands)

SYNOPSIS:

```
#include <tcl.h>

char *Tcl_TildeSubst( interp, name, bufferPtr );
```

ARGUMENTS:

Tcl_Interp *interp (in)

Interpreter in which to report an error, if any.

char *name (in)

File name, which may start with a ``~".

Tcl_DString *bufferPtr ()

If needed, this dynamic string is used to store the new file name. At the time of the call it should be uninitialized or empty. The caller must eventually call Tcl_DStringFree to free up anything stored here.

DESCRIPTION: This utility procedure does tilde substation. If name doesn't start with a ``~" character, then the procedure returns name. If name does start with a tilde, then Tcl_TildeSubst returns a new string identical to name except that the first element of name is replaced with the location of the home directory for the given user. The substitution is carried out in the same way that it would be done by csh. If the tilde is followed immediately by a slash, then the $HOME environment variable is substituted. Otherwise the characters between the tilde and the next slash are taken as a user name, which is looked up in the password file; the user's home directory is retrieved from the password file and substituted. If Tcl_TildeSubst has to do tilde substitution then it uses the dynamic string at *bufferPtr to hold the new string it generates. After Tcl_TildeSubst returns, the caller must eventually invoke Tcl_DStringFree to free up any information placed in *bufferPtr. The caller need not know whether or not Tcl_TildeSubst actually used the string; Tcl_TildeSubst initializes *bufferPtr even if it doesn't use it, so the call to

Tcl_DStringFree will be safe in either case. If an error occurs (e.g. because there was no user by the given name) then NULL is returned and an error message will be left at interp->result. It is assumed that interp- >result has been initialized in the standard way when Tcl_TildeSubst is invoked.

19.6.57 tclsh

tclsh - Simple shell containing TCL interpreter

(TCL Commands)

SYNOPSIS: tclsh ?fileName arg arg ...?

DESCRIPTION: Tclsh is a shell-like application that reads TCL commands from its standard input or from a file and evaluates them. If invoked with no arguments then it runs interactively, reading TCL commands from standard input and printing command results and error messages to standard output. It runs until the exit command is invoked or until it reaches end-of-file on its standard input. If there exists a file. tclshrc in the home directory of the user, tclsh evaluates the file as a TCL script just before reading the first command from standard input.

SCRIPT FILES

If tclsh is invoked with arguments then the first argument is the name of a script file and any additional arguments are made available to the script as variables (see below). Instead of reading commands from standard input, tclsh will read TCL commands from the named file; tclsh will exit when it reaches the end of the file. There is no automatic evaluation of .tclshrc in this case, but the script file can always source it if desired. If the user creates a TCL script in a file
whose first line is #!/usr/local/bin/tclsh then he can invoke the script file directly from his shell if he marks the file as executable. This assumes that tclsh has been installed in the default location in /usr/local/bin; if it is installed somewhere else then the user must modify the above line to match.

VARIABLES

Tclsh sets the following TCL variables: argc

Contains a count of the number of arg arguments (0 if none), not including the name of the script file.

argv Contains a TCL list whose elements are the arg arguments, in order, or an empty string if there are no arg arguments.

argv0 Contains fileName if it was specified. Otherwise, contains the name by which tclsh was invoked.

tcl_interactive

Contains 1 if tclsh is running interactively (no fileName was specified and standard input is a terminal-like device), 0 otherwise.

PROMPTS

When tclsh is invoked interactively it normally prompts for each command with ``% ". The user can change the prompt by setting the variables tcl_prompt1 and tcl_prompt2. If variable tcl_prompt1 exists then it must consist of a TCL script in order to output a prompt; instead of outputting a prompt tclsh will evaluate the script in tcl_prompt1. The variable tcl_prompt2 is used in a similar way when a newline is typed but the current command is not yet complete; if tcl_prompt2 is not set, then no prompt is output for incomplete commands.

19.6.58 tclvars

tclvars - Variables used by TCL

(TCL Commands)

DESCRIPTION: The following global variables are created and managed automatically by the TCL Library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users. **env** The variable **env** is maintained by **TCL** as an array whose elements are the environment variables for the process. Reading an element will return the value of the corresponding environment variable. Setting an element of the array will modify the corresponding environment variable or create a new one if it does not already exist. Unsetting an element of **env** will remove the corresponding environment variable. Changes to the **env** array will affect the environment passed to children by commands like exec. If the entire env array is unset then TCL will stop monitoring **env** accesses and will not update environment variables. errorCode

After an error has occurred, this variable will be set to hold additional information about the error in a form that is easy to process with programs. errorCode consists of a TCL list with one or more elements. The first element of the list identifies a general class of erors, and determines the format of the rest of the list. The following formats for errorCode are used by the TCL core; individual applications may define additional formats.

ARITH code msg his format is used when an arithmetic error occurs (e.g. an attempt to divide by zero in the expr command). Code identifies the precise error and msg provides a human-readable description of the error. Code will be either

DIVZERO

(for an attempt to divide by zero), DOMAIN (if an argument is outside the domain of a function, such as acos(-3)), IOVERFLOW (for integer overflow),

OVERLFLOW (for a floating-point overflow), or UNKNOWN (if the cause of the error cannot be determined).

CHILDKILLED pid sigName msg

This format is used when a child process has been killed because of a signal. The second element of errorCode will be the process's identifier (in decimal). The third element will be the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file signal.h, such as

SIGPIPE.

The fourth element will be a short human-readable message describing the signal, such as ``write on pipe with no readers" for SIGPIPE.

CHILDSTATUS pid code

This format is used when a child process has exited with a non-zero exit status. The second element of errorCode will be the process's identifier (in decimal) and the third element will be the exit code returned by the process (also in decimal).

CHILDSUSP pid sigName msg

This format is used when a child process has been suspended because of a signal. The second element of errorCode will be the process's identifier, in decimal. The third element will be the symbolic name of the signal that caused the process to suspend; this will be one of the names from the include file signal.h, such as:

SIGTTIN. The fourth element will be a short human-readable message describing the signal, such as ``background tty read" for SIGTTIN.

NONE

This format is used for errors where no additional information is available for an error besides the message returned with the error. In these cases errorCode will consist of a list containing a single element whose contents are NONE.

POSIX errName msg

If the first element of errorCode is POSIX, then the error occurred during a

POSIX

kernel call. The second element of the list will contain the symbolic name of the

error that occurred, such as ENOENT; this will be one of the values defined in the include file errno.h. The third element of the list will be a human-readable message corresponding to errName, such as ``no such file or directory'' for the

ENOENT

case. To set errorCode, applications should use library procedures such as Tcl_SetErrorCode and Tcl_PosixError, or they may invoke the error command. If one of these methods hasn't been used, then the TCL interpreter will reset the variable to NONE after the next error.

errorInfo

After an error has occurred, this string will contain one or more lines identifying the TCL commands and procedures that were being executed when the most recent error occurred. Its contents take the form of a stack trace showing the various nested TCL commands that had been invoked at the time of the error.

tcl_precision

If this variable is set, it must contain a decimal number giving the number of significant digits to include when converting floating-point values to strings. If this variable is not set then 6 digits are included. 17 digits is ``perfect'' for IEEE floating-point in that it allows double-precision values to be converted to strings and ack to binary with no loss of precision.

19.6.59 tell

tell - Return current access position for an open file

(TCL Commands)

SYNOPSIS: tell fileId

DESCRIPTION: Returns a decimal string giving the current access position in fileId. FileId must have been the return value from a previous call to open, or it may be stdin, stdout, or stderr to refer to one of the standard I/O channels.

19.6.60 time

time - Time the execution of a script

(TCL Commands)

SYNOPSIS: time script ?count?

DESCRIPTION: This command will call the TCL Interpreter count times to evaluate script (or once if count is not specified). It will then return a string of the form 503 microseconds per iteration which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

19.6.61 trace

trace - Monitor variable accesses

(TCL Commands)

SYNOPSIS: trace option ?arg arg ...?

DESCRIPTION: This command causes TCL commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The legal options (which may be abbreviated) are:

trace variable name ops command
Arrange for command to be executed whenever variable name is accessed in one of the ways given by ops. Name may refer to a normal variable, an element of an array, or to an array as a whole (i.e. name may be just the name of an array, with no parenthesized index). If name refers to a whole array, then command is invoked whenever any element of the array is manipulated.

Ops indicates which operations are of interest, and consists of one or more of the following letters:

r

Invoke command whenever the variable is read.

w

Invoke command whenever the variable is written.

u

Invoke command whenever the variable is unset. Variables can be unset explicitly with the unset command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to command so that the actual command is as follows: command name1 name2 op.

Name1 and name2 give name(s) for the variable being accessed: if the variable is a scalar then name1 gives the variable's name and name2 is an empty string; if the variable is an array

element,
then name1 gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then name1 gives the array name and name2 is an empty string. Op indicates what operation is being performed on the variable, and is one of r, w, or u as defined above.

Command executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a TCL procedure, then command will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If command invokes a procedure (which it normally does) then the procedure will have to use upvar or uplevel if it wishes to access the traced variable. Note also that name1 may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the upvar command.

For read and write traces, command can modify the variable to affect the result of the traced operation. If command modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from command is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the trace command (this mechanism can be used to implement read-only variables, for example). For write traces, command is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, command will have to restore the old value of the variable.

While command is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by command will occur directly, without invoking command (or any other traces) again. However, if command unsets the variable then unset traces will be invoked. When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace

on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the trace vdelete command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array. This command returns an empty string.

trace vdelete name ops command

If there is a trace set on variable name with the operations and command given by ops and command, then the trace is removed, so that command will never again be invoked. Returns an empty string.

trace vinfo name

Returns a list containing one element for each trace currently set on variable name. Each element of the list is itself a list containing two elements, which are the ops and command associated with the trace. If name doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

19.6.62 unknown

unknown - Handle attempts to use non-existent commands

(TCL Commands)

SYNOPSIS: unknown cmdName ?arg arg ...?

DESCRIPTION: This command doesn't actually exist as part of TCL, but TCL will invoke it if it does exist. If the TCL Interpreter encounters a command name for which there is not a defined command, then TCL checks for the existence of a command named unknown. If there is no such command, then the interpreter returns an error. If the unknown command exists, then it is invoked with arguments consisting of the fully-substituted name and arguments for the original non-existent command. The unknown command typically does things like searching through library directories for a command procedure with the name cmdName, or expanding abbreviated command names to full-length, or automatically executing unknown commands as sub-processes. In some cases (such as expanding abbreviations) unknown will change the original command slightly and then (re-)execute it. The result of the unknown command is used as the result for the original non-existent command.

19.6.63 unset

unset - Delete variables

(TCL Commands)

SYNOPSIS: unset name ?name name ...?

DESCRIPTION: This command removes one or more variables. Each name is a variable name, specified in any of the ways acceptable to the set command. If a name refers to an element of an array, then that element is removed without affecting the rest of the array. If a name consists of an array name with no parenthesized index, then the entire array is deleted. The unset command returns an empty string as result. An error occurs if any of the variables doesn't exist, and any variables after the non-existent one are not deleted.

19.6.64 uplevel

uplevel - Execute a script in a different stack frame

(TCL Commands)

SYNOPSIS: uplevel ?level? arg ?arg ...?

DESCRIPTION: All of the arg arguments are concatenated as if they had been passed to concat; the result is then evaluated in the variable context indicated by level. Uplevel returns the result of that evaluation. If level is an integer then it gives a distance (up the procedure calling stack) to move before executing the command. If level consists of # followed by a number then the number gives an absolute level number. If level is omitted then it defaults to 1. Level cannot be defaulted if the first command argument starts with a digit or #. For example, suppose that procedure a was invoked from top-level, and that it called b, and that b called c. Suppose that c invokes the uplevel command. If level is 1 or #2 or omitted, then the command will be executed in the variable context of b. If level is 2 or #1 then the command will be executed in the variable context of a. If level is 3 or #0 then the command will be executed at top-level (only global variables will be visible). The uplevel command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose c invokes the command uplevel 1 {set x 43; d} where d is another TCL procedure. The set command will modify the variable x in b's context, and d will execute at level 3, as if called from b. If it in turn executes the command uplevel {set x 42} then the set command will modify the same variable x in b's context: the procedure c does not appear to be on the call stack when d is executing. The command ``info level'' may be used to obtain the level of the current procedure. Uplevel makes it possible to implement new control constructs as TCL procedures (for example, uplevel could be used to implement the while construct as a TCL procedure).

19.6.65 upvar

upvar - Create link to variable in a different stack frame

(TCL Commands)

SYNOPSIS: upvar ?level? otherVar myVar ?otherVar myVar ...?

DESCRIPTION: This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. Level may have any of the forms permitted for the uplevel command, and may be omitted if the first letter of the first otherVar isn't # or a digit (it defaults to 1). For each otherVar argument, upvar makes the variable by that name in the procedure frame given by level (or at global level, if level is #0) accessible in the current procedure by the name given in the corresponding myVar argument. The variable named by otherVar need not exist at the time of the call; it will be created the first time myVar is referenced, just like an ordinary variable. Upvar may only be invoked from within procedures. MyVar may not refer to an element of an array, but otherVar may refer to an array element. Upvar returns an empty string.

The upvar command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as TCL procedures. For example, consider the following procedure:

proc add2 name {upvar $name x set x [expr $x+2]}.

Add2 is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although add2 could have been implemented using uplevel instead of upvar, upvar makes it simpler for add2 to access the variable in the caller's procedure frame. If an upvar variable is unset (e.g. x in add2 above), the unset operation affects the variable it is linked to, not the upvar variable. There is no way to unset an upvar variable except by exiting the procedure in which it is defined. However, it is possible to retarget an upvar variable by executing another upvar command.

19.6.66 while

while - Execute script repeatedly as long as a condition is met

(TCL Commands)

SYNOPSIS: while test body

DESCRIPTION: The while command evaluates test as an expression (in the same way that expr evaluates its argument). The value of the expression must a proper boolean value; if it is a true value then body is executed by passing it to the TCL interpreter. Once body has been executed then test is evaluated again, and the process repeats until eventually test evaluates to a false boolean value. Continue commands may be executed inside body to terminate the current iteration of the loop, and break commands may be executed inside body to cause immediate termination of the while command. The while command always returns an empty string.

20. Index